Università degli Studi di Padova
Facoltà di Ingegneria

Finito di scrivere il giorno 4 ottobre 2014 utilizzando LaTeX 2ε

UNIVERSITÀ DEGLI STUDI DI PADOVA
FACOLTÀ DI INGEGNERIA

—

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

—

TESI DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

GESTIONE ED ELABORAZIONE DEI DATI DI ESERCIZIO PER IL MONITORAGGIO CONTINUO DI SISTEMI ELETTRICI PER LA PROPULSIONE NAVALE

RELATORE: CH.MO PROF. CARLO FERRARI
CORRELATORI: PROF.SSA ELEONORA DI MARIA,
ING. MAURIZIO RIZZI

LAUREANDO: NICOLA CORSO

ANNO ACCADEMICO 2013-2014

Ai miei genitori..

"640 K dovrebbero bastare a chiunque"

Bill Gates, parlando di Internet nel 1981

Indice

Sommario

Grazie alla sempre maggior copertura della rete Internet nel mondo e alla più ampia diffusione di componenti hardware embedded, qualsiasi dispositivo elettronico può accedere alla rete. Questo permette alle aziende di poter offrire ai propri clienti maggiori e sempre nuovi servizi. Il caso trattato da questo elaborato riguarda lo sviluppo di un sistema software per il monitoraggio continuo dei dati di esercizio di generatori elettrici installati sulle navi. Lo scopo è quello di migliorare le attività di manutenzione ed offrire nuovi servizi per la consultazione dello stato dei generatori ai propri clienti. Tutte le attività sono state svolte in collaborazione con l'azienda *Marelli Motori* di Arzignano, Vicenza. Il lavoro verrà presentato al *Middle East Electricity 2015*, la più importante manifestazione riguardante la produzione di energia elettrica.

Introduzione

La stesura di questa tesi di laurea nasce dall'esigenza concreta di una azienda di migliorare le attività di manutenzione dei propri componenti. La ditta produce principalmente generatori elettrici che vengono installati sulle navi per la loro propulsione. Ogni motore viene collegato ad un regolatore: una scheda di controllo per l'alimentazione e il settaggio di vari parametri del generatore. Al momento dell'installazione il regolatore viene configurato per operare secondo determinate grandezze elettriche.

Per ogni generatore in circolazione, le attività di manutenzione vengono programmate periodicamente. Per ciascun controllo un tecnico deve recarsi nella sala motori della nave, collegare un laptop al regolatore e verificare lo stato del componente. In particolare, si controllano la presenza di allarmi registrati dalla scheda segnalanti situazioni di funzionamento anomale. Se si considera che i generatori attualmente in circolazione sono circa 2000, e che sono sparsi in tutto il mondo, si capisce che questo modo di effettuare i controlli rappresenta un enorme spreco di risorse economiche e umane.

L'azienda ha necessità di migliorare il servizio di assistenza. Le operazioni di controllo dello stato dei motori devono essere *remotizzate* e non richiedere sempre l'intervento del tecnico sul posto. Anche alcune configurazioni dei componenti devono poter essere effettuate da remoto. Per permettere queste due operazioni, ogni regolatore *Marelli* in circolazione deve essere raggiungibile tramite una connessione Internet. Inoltre, i dati d'esercizio che vengono continuamente rilevati devono essere sempre consultabili.

Questo elaborato riguarda l'implementazione di un sistema in grado di rispondere a queste necessità. Per prima cosa si sono raccolte le informazioni e i requisiti aziendali circa il funzionamento dei vari dispositivi e delle dinamiche riguardanti la loro manutenzione. Nel Capitolo 1 vengono riportate le specifi-

che del progetto. Durante i colloqui aziendali sono emerse alcune esigenze di contorno da affiancare alla necessità principale sopra esposta. La creazione di uno storico dei dati di esercizio, non solo permette ai tecnici di avere informazioni aggiuntive per effettuare le manutenzioni, ma può essere utilizzato anche ad altri scopi. L'azienda ha individuato la possibilità di fornire ai propri clienti una applicazione che permetta di controllare l'esercizio dei generatori direttamente da uno smartphone. Questa funzionalità verrà presentata al *Middle East Electricity* che si terrà a febbraio a Dubai. La continua archiviazione dei dati di funzionamento potrà in futuro fornire supporto alla azienda tramite analisi statistiche. Le attività di *Data Mining* per esempio, possono contribuire a migliorare la produzione o la stessa progettazione del prodotto.

Il primo passo nella progettazione del sistema riguarda la costruzione del data base che farà da supporto all'intera applicazione. Nel Capitolo 2 vengono riportati i passi principali che hanno portato allo sviluppo della base di dati. I dati raccolti dalle analisi dei requisiti sono stati rielaborati per costruire un elenco conciso di *frasi filtrate* ed un *glossario dei termini*, utili nella realizzazione del data base. Tramite la *progettazione concettuale* si è ottenuto lo schema *ER*, utilizzato poi nella *progettazione logica* per ricavare il modello relazionale. Quest'ultimo è stato implementato tramite il linguaggio *SQL* nel *DBMS PostgreSQL*.

Nel Capitolo 3 viene trattata l'organizzazione dell'*hardware* e del *software* che compongono il sistema. L'architettura adottata è di tipo *client-server* a tre livelli. In questo capitolo, oltre al modello del sistema, si descrivono i protocolli di comunicazione che vengono utilizzati e l'organizzazione dei dati che vengono scambiati.

Essendomi concentrato nello sviluppo del software lato server, il capitolo successivo riguarda aspetti legati all'applicazione server. Viene riportato il diagramma di flusso che segue l'esecuzione del software e, per ciascuna richiesta, sono riportate le azioni da intraprendere. Ogni richiesta interagisce con il data base secondo le *query* illustrate in questo capitolo.

Per la consultazione dei dati rilevati dall'applicazione è stata implementata una applicazione Android. Il Capitolo 5 riguarda l'estensione al server per rispondere alle richieste generate dagli smartphone e presenta l'applicazione schermata dopo schermata come farebbe una guida per l'utente.

Infine, nel Capitolo 6 si trattano i possibili sviluppi futuri del progetto. Vengono presentate in questo capitolo funzionalità aggiuntive, alcune indispensabili, che il sistema dovrà avere quando sarà completamente ultimato.

Gli aspetti trattati sono legati alla sicurezza e alla scalabilità del sistema. Viene infine proposta una possibile architettura per i settaggi dei parametri da remoto.

Capitolo 1

Analisi dei requisiti

La raccolta delle informazioni rappresenta la parte più delicata nell'avvio di un progetto. È molto importante in questa fase prestare attenzione ad ogni singolo dettaglio riguardante le specifiche che l'applicazione dovrà possedere quando terminata. Una volta che lo sviluppo dell'intero sistema è avviato, un cambio di requisiti può compromettere gran parte del lavoro svolto. La raccolta delle informazioni presso la *Marelli Motori* ha avuto la durata di due mesi ed ha costituito la linea guida per tutte le scelte progettuali, soprattutto per la creazione del data base. Una base di dati rappresenta infatti una infrastruttura statica e immutabile nel tempo. Per la complessità dell'operazione, una volta entrato in funzione un data base viene modificato.

1.1 Il regolatore MEC-100

Come già detto l'azienda produce generatori elettrici installati su navi, destinati alla propulsione. Inoltre ogni motore è controllato da un regolatore. Le attività di diagnostica riguardano il generatore, ma solo indirettamente. È il regolatore infatti a interfacciarsi con il motore: le rilevazioni e tutte le attività di manutenzione vengono fatte attraverso quest'ultimo. Diverse tipologie di controllori vengono distribuite dall'azienda, ma il più diffuso è il regolatore *MEC-100*. Tutto lo sviluppo del architettura software fa riferimento a questo particolare modello di regolatore, tenendo però presente che in futuro ulteriori schede dovranno essere integrate all'interno del sistema. Le scelte attuate nello sviluppo dell'infrastruttura garantiscono la flessibilità necessaria per estendere l'applicativo ad altri regolatori. Nel seguito di questo elaborato il termine *MEC-100* verrà utilizzato come sinonimo di regolatore.

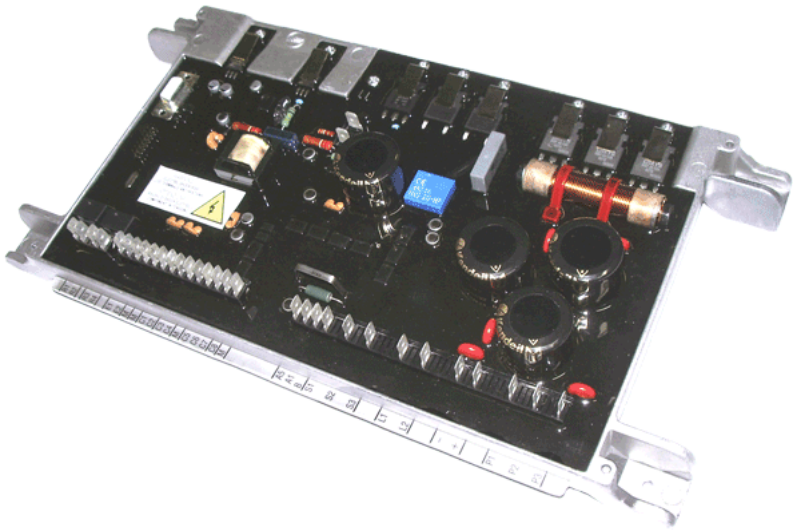

Figura 1.1: Il regolatore MEC-100

Il regolatore *MEC-100* è una scheda elettronica che viene installata su di un generatore elettrico. È dotato di interfaccia seriale *RS-232* tramite un connettore *DB-9* femmina. Per effettuare il settaggio del componente si deve utilizzare un software dedicato, distribuito dalla *Marelli Motori*. 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. Un tecnico, al momento dell'installazione della scheda, ne può modificare la configurazione di fabbrica, operazione che viene fatta collegando un computer attraverso un cavo seriale ed utilizzando sempre lo stesso software per interfacciarsi. Tramite il programma, il manutentore può leggere i valori dei parametri di configurazione del *MEC-100* e può rilevarne i dati di esercizio, ovvero i parametri elettrici con cui il regolatore sta interagendo con il generatore.

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`. Un *Alarm* che è settato ad `ON` rappresenta un'anomalia identificata dal codice dell'*Alarm*.

Se necessario il tecnico può modificare i valori di funzionamento settando dei nuovi parametri di configurazione.

Ogni componente *Marelli* ha perciò tre tipologie di parametri:

1. Parametri statici di lettura. Ne fanno parte i codici identificativi della scheda e le versioni *firmware*.

2. Parametri variabili di lettura. Sono le grandezze elettriche che rappresentano i valori di esercizio della scheda oppure dei *flag booleani* che indicano il verificarsi di un *Alarm*.

3. Parametri configurabili di scrittura/lettura. Sono i vari settaggi della scheda cioè quei parametri di configurazione che il tecnico può regolare.

1.2 La scheda Hachiko

Per poter monitorare i dati di esercizio, il *MEC-100* deve essere dotato di connettività Internet. Essendo la porta seriale l'unica a interfacciarsi con questo regolatore, si dovrà comunicare per mezzo di questa con un dispositivo capace di connettersi alla rete.

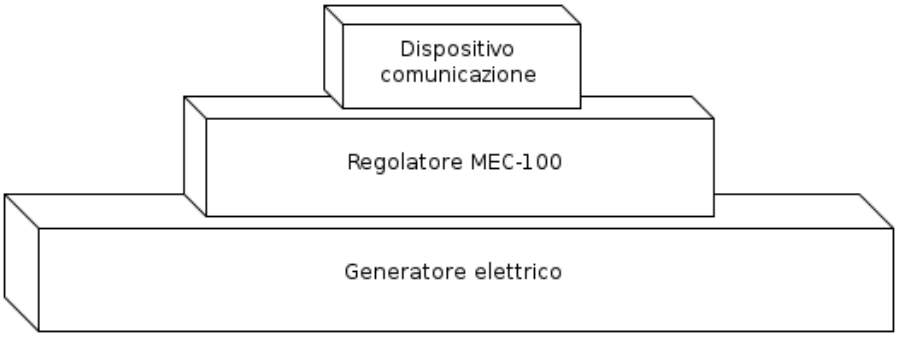

Figura 1.2: Connessione tra i dispositivi coinvolti. Il generatore elettrico è collegato al regolatore che ne garantisce il funzionamento. Il regolatore verrà collegato tramite porta seriale ad un dispositivo connesso alla rete Internet. Questo deve essere programmato per gestire il recupero dei dati di esercizio e inviarli ad un elaboratore.

Dopo i diversi incontri con i fornitori si è deciso di utilizzare per questo scopo la scheda *Hachiko* fornita da *Silica*. Tra le varie scelte questa rappresentava la più adatta per diversi motivi. Innanzitutto la scheda è dotata di sistema operativo *Linux* basato sul progetto *Yocto*. Quest'ultimo gode del supporto di aziende importanti nel settore ed ha una community molto numerosa. Utilizzare prodotti mantenuti nel tempo è fondamentale per un'azienda. Anche la stessa *Silica* offre supporto alla *Marelli*. Altre schede invece non garantivano

alcuna assistenza. I componenti principali della scheda *Hachiko* sono integrati all'interno della board, rendendola più robusta alle sollecitazioni esterne dovute alla difficile condizione ambientale in cui il regolatore opera. Inoltre, diventa quasi indispensabile l'utilizzo di un sistema operativo. Quest'ultimo ha reso più semplice lo sviluppo dell'applicativo che la scheda ospita. La connessione Internet viene resa trasparente all'applicazione e la scrittura di un *driver* per la comunicazione seriale tra le schede è semplificata rispetto all'utilizzo di una scheda senza *OS*. La scheda inoltre è dotata di uno schermo *touchscreen* da 7 pollici.

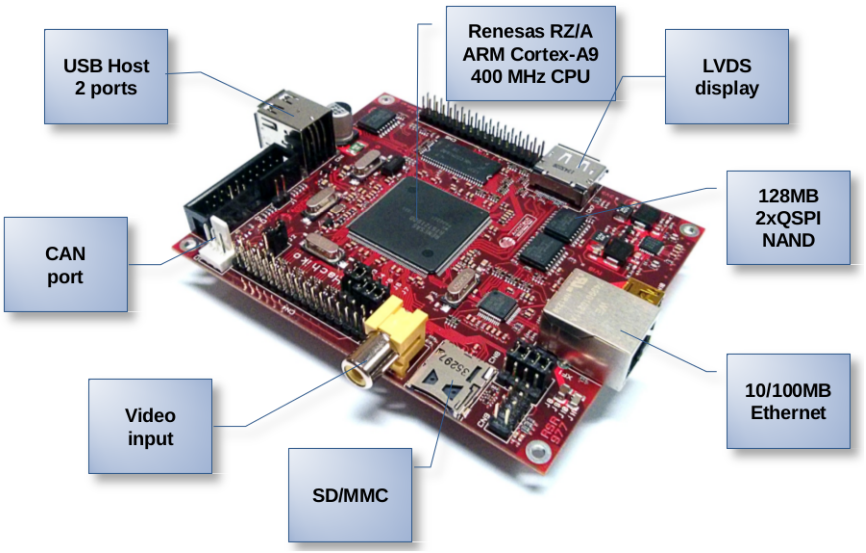

Figura 1.3: La scheda *Hachiko* e le sue specifiche tecniche.

1.3 Analisi dei requisiti

1.3.1 Specifiche generali di progetto

Il progetto riguarda la creazione di un sistema *IT* per la raccolta dei dati di esercizio e la manutenzione dei componenti *Marelli MEC-100 sparsi nel mondo*. Il sistema deve fornire supporto per quanto riguarda:

i) la manutenzione e l'assistenza del regolatore;

ii) il rilievo di statistiche per migliorare la fase di produzione;

iii) il servizio di consultazione dello stato dei regolatori da parte dei clienti.

L'applicazione gestione del suddetto sistema ha lo scopo di comunicare con i componenti e rilevarne i parametri di configurazione e i dati d'esercizio. Deve essere possibile risalire alle manutenzioni eseguite su di un particolare dispositivo.

Tre applicativi tra loro correlati costituiscono il sistema nella sua totalità.

Figura 1.4: L'applicazione installata sulle schede *Hachiko* interagisce direttamente con il regolatore e comunica gli aggiornamenti ad un server attraverso Internet. Ai clienti verrà distribuita un'applicazione *Android* per consultare i dati relativi ai propri regolatori.

- L'applicazione lato server si occupa di interagire con il data base di supporto, gestire la comunicazione con le schede *Hachiko* e alcuni servizi addizionali come le ricerche tramite *query* o le analisi di tipo statistico.
- L'applicazione ospitata dalle singole schede *Hachiko* ha il compito di estrarre i dati dal *MEC-100* a intervalli regolari ed inviarli al server.
- L'applicazione *Android* da fornire ai clienti per la consultazione delle informazioni relative alle proprie macchine. Queste riguardano i dati di esercizio e altre informazioni presenti nel data base.

Allo scopo di migliorare l'assistenza e limitare l'intervento sul posto del personale *Marelli*, il software deve offrire servizi riguardanti la manutenzione dei componenti. A supporto di tali attività l'applicazione deve garantire nel complesso alcune funzionalità.

Storico dei dati di esercizio. I valori letti dalla scheda *Hachiko* devono essere opportunamente archiviati, con la possibilità di esplorare le letture a grana sempre più fina, fino ad arrivare alla singola lettura.

Storico degli interventi di assistenza. Per ogni regolatore si deve poter risalire all'elenco delle operazioni di manutenzione che vi sono state fatte. Per ogni assistenza si devono registrare tutte le attività che il tecnico svolge durante la manutenzione. Manutenzioni future trovano semplificazione 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. Si deve prevedere la possibilità di fornire statistiche di vario tipo riguardanti i regolatori. Tramite i dati memorizzati all'interno della base di dati si possono effettuare analisi circa il funzionamento, i guasti e gli interventi di manutenzione.

La scheda *Hachiko* può essere integrata con uno schermo *touchscreen*. Questo può essere utilizzato per offrire al tecnico una nuova interfaccia verso il dispositivo. Per mantenere un controllo più diretto sulle attività di manutenzione e rilevarne ogni singola operazione di configurazione, il software *Marelli* verrà riscritto per essere contenuto nella scheda *Hachiko*. In questo modo le operazioni di assistenza saranno eseguibili direttamente dalla scheda. Spesso lo spazio fisico sulla quale il tecnico è costretto a lavorare rende difficile lo stesso collegamento tra computer e regolatore. Questa nuova interfaccia vi porrà rimedio.

1.3.2 Raccolta delle informazioni

Chiarito il funzionamento dei vari dispositivi che compongono il sistema, sono state raccolte ulteriori informazioni dall'azienda che impatteranno sullo svi-

luppo del data base e delle applicazioni software.

Gli applicativi inizialmente sono stati sviluppati per uno specifico modello di regolatore, il *MEC-100*. Tuttavia in futuro potranno essere utilizzate altri controllori. Le applicazioni dovranno essere adattate per supportare i nuovi regolatori, ma il data base deve da subito venir sviluppato per essere esteso facilmente ai nuovi regolatori.

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

Il reparto di manutenzione dovrà disporre di un applicativo che comunicando con la scheda riesca a settare alcuni parametri di configurazione. 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*.

- Guasto. Riguarda l'intervento sul posto con l'eventuale sostituzione del regolatore guasto e, 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 che in futuro sarà eseguita da remoto.

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. Questo effettua alcune configurazioni della scheda procedendo per tentativi fino ad ottenere la configurazione ottimale. Tutte le modifiche apportate devono venir 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 del regolatore attraverso la scheda *Hachiko* mediante un controllo di accesso. Questo viene fatto in remoto tramite il server di autenticazione. Nel caso in cui la connessione Internet non sia disponibile, l'autenticazione viene fatta localmente alla scheda, controllando tra gli utenti *Linux* registrati nella scheda *Hachiko*. L'applicazione deve prevedere un controllo abbastanza elaborato 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.

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 sciolto 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 *heartbeat* è necessario per controllare la continua connessione tra scheda *Hachiko* e regolatore. Inoltre la connessione alla rete Internet non è sempre disponibile, le navi nei posti più remoti potrebbero rimanere senza collegamento per molto tempo. La scheda *Hachiko* rileva i dati con continuità e li deve memorizzare temporaneamente nella memoria secondaria, per poi inviarli appena possibile.

Il database dovrà rispondere a query come:

- Individuare quanti generatori utilizzano un determinato tipo di regolatore.
- Quale *MEC-100* è utilizzato per un determinato motore.
- Quanti MEC100 sono utilizzati in motori che hanno una determinata caratteristica.

Al giorno d'oggi ci sono circa 2000 regolatori *MEC-100* in circolazione, di cui l'80% appartiene ad un unico cliente.

Capitolo 2

Database

2.1 Analisi delle informazioni raccolte

La fase successiva alla raccolta delle informazioni consiste nella costruzione del data base. La base di dati deve fornire tutte le operazioni di memorizzazione a supporto delle necessità aziendali riportate. I requisiti definiti dall'azienda sono prima stati filtrati per formare un elenco conciso delle specifiche per poi ricavarne un glossario dei dati. Quest'ultimo ha lo scopo di definire i concetti in modo univoco per non generare incomprensioni dovute alla raccolta delle informazioni.

2.1.1 Frasi filtrate

Il progetto si basa sul modello di regolatore *MEC-100* e sulla scheda *Hachiko*.

Vi sono diverse versioni di scheda *MEC-100* e in futuro si dovranno introdurre nuovi regolatori.

Ogni *MEC-100* ha dei parametri di configurazione, alcuni statici e altri variabili, dei parametri di lettura e dei parametri *booleani* che rappresentano degli *Alarm*.

I dati di esercizio devono essere storicizzati e deve essere possibile risalire alle singole letture puntuali.

Si deve poter verificare la presenza di allarmi per ogni regolatore.

Le operazioni di assistenza possono esser fatte tramite lo schermo della scheda *Hachiko*. Per ogni *MEC-100* si deve poter risalire alle manutenzioni fatte in passato con tutti i dettagli del caso.

I tecnici sono abilitati ad una assistenza dalla scheda *Hachiko* previo un'autenticazione tramite *username* e *password*.

Vi sono alcune regole da rispettare nella configurazione del dispositivo *MEC-100*. In particolare alcuni settaggi ne implicano degli altri.

Un solo tecnico esegue un'operazione di assistenza per volta. Alcuni tecnici hanno il permesso di effettuare operazioni di manutenzione solamente su un insieme limitato di schede.

Per ogni manutenzione si devono poter memorizzare immagini e altri documenti.

C'è la necessità di tenere traccia dei collegamenti tra i regolatori e generatori, e tra regolatori e schede *Hachiko* nel tempo.

La scheda *Hachiko* deve controllare periodicamente di essere connessa al regolatore implementando un protocollo *Hearbeat*.

Se una scheda *Hachiko* rileva una configurazione dei parametri diversa dall'ultima che non è stata registrata, significa che c'è stata una manutenzione non autorizzata.

Si devono poter eseguire configurazioni dei regolatori da remoto.

Attraverso i dati contenuti nel data base devono essere possibili alcune operazioni di analisi statistica.

Il data base deve memorizzare i dati riguardanti i generatori, i quali hanno parametri e informazioni a sè stanti.

Ogni regolatore è di proprietà di un determinato cliente, è installato su un

determinato motore ed è identificato univocamente.

Un regolatore può cambiare l'associazione con il generatore nel tempo.

2.1.2 Glossario dei termini

In tabella 2.1 sono riportati i termini utilizzati fino ad ora nei loro vari sinonimi.

Concetto	Descrizione	Sinonimi
Generatore	Generatore elettrico installato sulle navi per la loro propulsione.	Motore elettrico
Regolatore	Scheda elettronica collegata al generatore per controllarne il funzionamento.	Regolatore, *MEC-100*, Controllore, Scheda
Hachiko	Scheda elettronica collegata al regolatore per rilevarne i dati e comunicare con un server attraverso Internet.	Scheda, Scheda per la comunicazione
Assistenza	Attività di rilevamento e configurazione di dati dal regolatore.	Manutenzione
Tecnico	Adetto alla manutenzione che può essere un dipendente *Marelli* o di altre aziende.	Manutentore
Cliente	Azienda che acquista un regolatore o un generatore.	Acquirente, Utente

Tabella 2.1: Glossario dei dati

2.2 Progettazione concettuale

In questa fase di progettazione si vuole ottenere un modello Entità-Associazione (*ER: Entity-Relationship*) atto a rappresentare la base di dati. Dallo schema, avendo deciso di utilizzare un data base relazionale, si procede tramite la progettazione logica alla implementazione delle tabelle. Il modello *ER* rappresenta uno strumento di sviluppo potente e flessibile che attraverso le semantiche degli elementi che lo compongono stabilisce la logica che i dati assumono per l'applicazione. È inoltre uno strumento facilmente comprensibile per chi commissiona

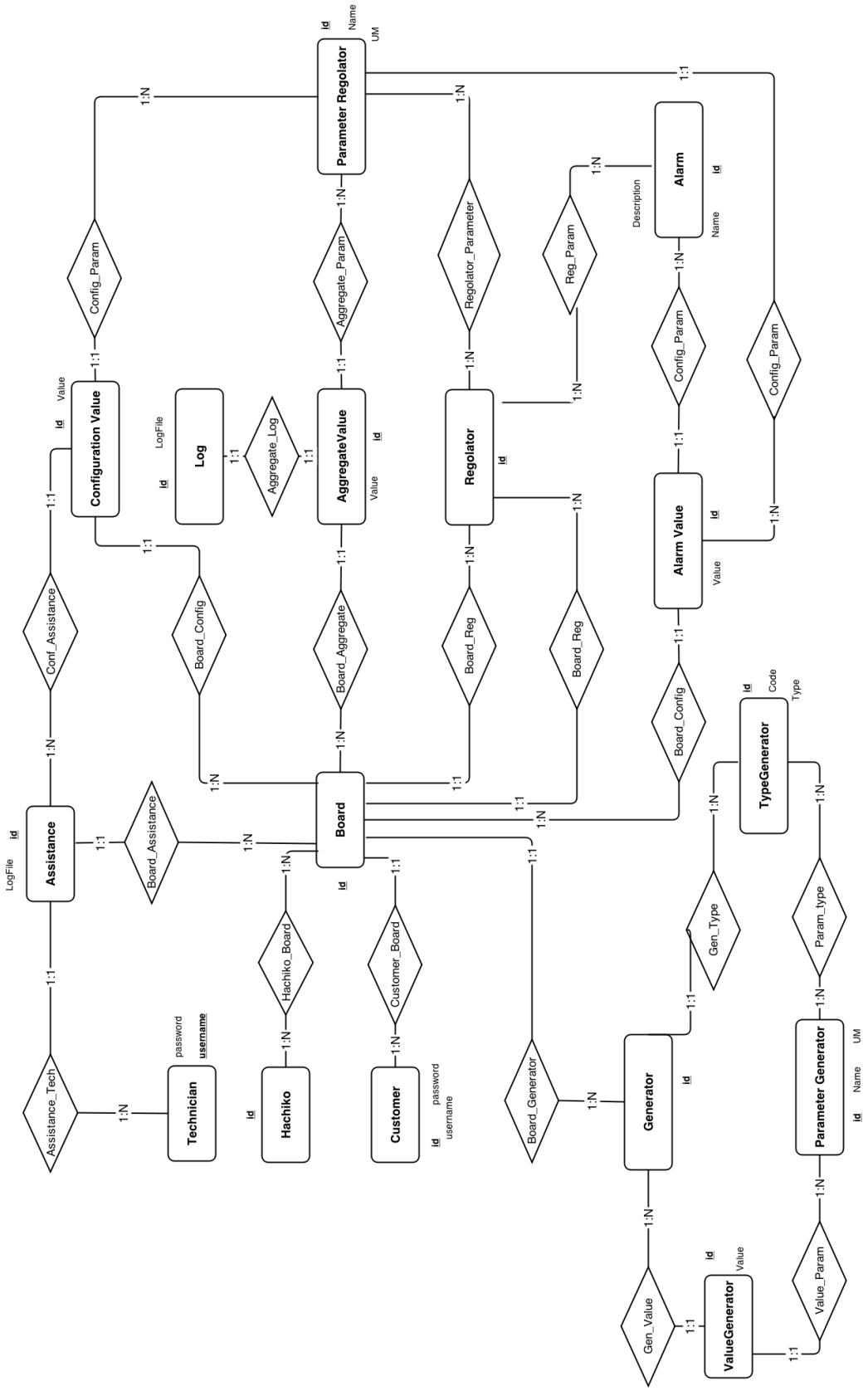

Figura 2.1: Lo schema *ER* finale.

lo sviluppo del data base, che può interagire nella fase di progettazione con gli sviluppatori per individuare gli errori derivati dalla fase di raccolta delle informazioni.

La base di dati ha lo scopo di supportare tutte le operazioni che le varie applicazioni dovranno svolgere. Nella stesura del modello *ER* si deve perciò tenere a mente quali sono gli scopi dell'applicazione nel suo complesso e come le singole componenti software devono interagire con il DB.

In Figura 2.1 è riportato lo schema *ER* finale, così come è stato implementato. Per facilitare la comprensione della descrizione delle varie componenti dello schema, questo è stato suddiviso in diverse parti: dov'era possibile le entità e le associazioni sono state raggruppate in base al loro contribuito in termini di significato, ma essendo lo schema molto vasto, non sempre è stato possibile applicare questa strategia. L'*ER* è per natura autodescrittivo, quindi le spiegazioni successive si baseranno su alcune scelte di progettazione particolari, evitando di soffermarsi su dettagli banali.

La modellazione delle schede. L'entità principale su cui ruota l'intero schema è l'entità *Board*. Questa rappresenta un determinato regolatore *MEC-100*. La chiave principale *id* è il codice seriale della scheda. Questa entità dovrebbe contenere gli attributi relativi alla scheda. Essendo che in futuro ulteriori regolatori saranno introdotti nel sistema, i parametri di configurazione e i dati di esercizio non possono essere trattati come attributi di *Board*. Un regolatore diverso dal *MEC-100* potrebbe avere parametri diversi. Fanno parte degli attributi di *Board* anche i valori che le applicazioni devono memorizzare per ogni scheda, come l'indirizzo ip o l'ultima volta che la scheda è stata online.

Regolator è l'entità che rappresenta la tipologia di regolatore. Vi sarà una tupla per ciascun tipo di regolatore supportato dal sistema. Le tre versioni del *MEC-100* avranno ognuna una istanza differente in questa relazione. Altri regolatori avranno le loro rappresentanze. In questo modo ciascuna *Board* è indipendente dalla tipologia di regolatore *Regolator*.

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

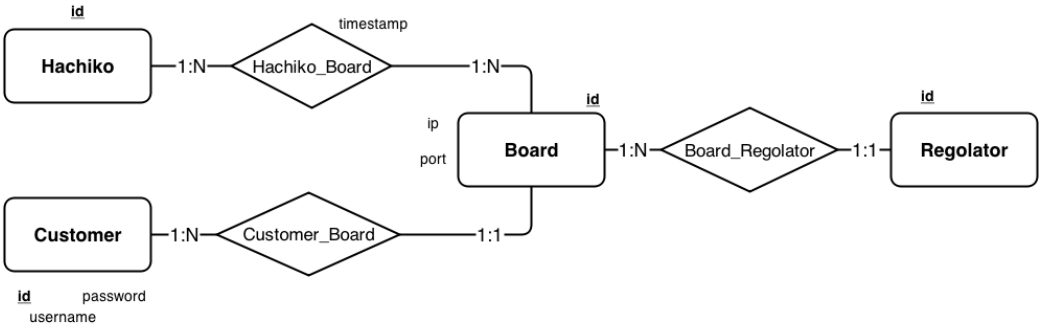

Figura 2.2: Schema *ER* per la rappresentazione delle schede *MEC-100* e *Hachiko* nel data base. Le singole istanze dei regolatori sono disaccoppiate dalla loro tipologia, per permettere l'inserimento di nuovi controllori.

L'entità *Customer* contiene i clienti dell'azienda. Questa entità ha attributi *username* e *password* che potranno essere utilizzati per controllare le *Board* attraverso l'applicazione Android. Altri attributi di questa entità possono essere quelli relativi all'anagrafica o ai contatti aziendali.

I parametri di configurazione e i dati d'esercizio. Per rappresentare i parametri di configurazione e di lettura di un regolatore è stata introdotta l'entità *PrameterRegolator*. Attributi di entità sono il nome completo del parametro, la sua unità di misura ed un codice identificativo. Tramite l'associazione con *Regolator* si costruisce, per ogni regolatore, la lista dei parametri. In questo modo diventa semplice inserire un nuovo regolatore nel sistema.

Per rappresentare i singoli valori numerici assunti dai parametri sono state introdotte le due entità *AggregateValue* e *ConfigurationValue*. I valori di configurazione andranno inseriti in *ConfigurationValue*. Un campo *timestamp* è necessario per ordinare i valori in modo da ricavare la storia delle configurazioni. Tramite l'associazione con *Board* si correlano i valori di configurazione con le schede che li riguardano. L'associazione con *ParameterRegolator* collega il valore al parametro corrispondente. L'entità *AggregateValue* è utilizzata per i parametri di lettura. *AggregateValue* e *ConfigurationValue* sono tenute separate per il diverso significato dei dati che rappresentano. Inoltre l'entità per i valori di configurazione ha, come si vedrà di seguito, un'associazione con l'entità che rappresenta le sessioni di assistenza.

Per limitare gli inserimenti dei valori di lettura da parte dei molteplici regolatori, i dati vengono inviati dalle schede *Hachiko* in forma aggregata. I valori che il server riceverà saranno perciò delle medie. Assieme ad ogni

media viene generato un file di *log*, contenente tutti i singoli valori rilevati dalle schede *Hachiko*. Questo per ridurre gli inserimenti nel data base, diminuendo il carico totale che il sistema deve sopportare. Nel data base vanno inseriti i valori aggregati che vengono collegati al file di log che contiene i singoli valori puntuali. Ciò è rappresentato nello schema *ER* dall'entità *AggregateValue* per i valori aggregati e dall'entità *Log* per i file di log. Le considerazioni che sono qui state fatte dipendono fortemente dalla logica con cui si è sviluppata l'applicazione *Hachiko*.

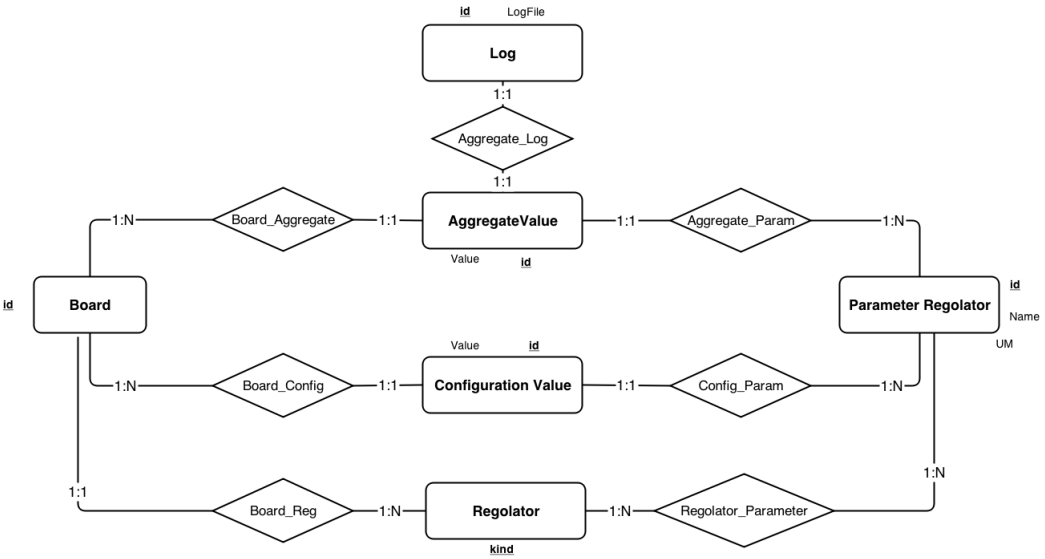

Figura 2.3: Schema per la rappresentazione dei parametri di esercizio e di configurazione.

Gestione degli Alarm. Gli *Alarm* sono stati trattati in modo simile a quanto fatto per i dati di esercizio. Si ha un *Alarm* durante la lettura dei valori, quando viene rilevato un particolare parametro ad `ON`. Anche gli allarmi sono identificati da codici come per gli altri parametri. Questi speciali identificativi vengono memorizzati in forma statica nell'entità *Alarm* ed una associazione li collega a *Regolator* per costruire la lista di allarmi che un determinato regolatore supporta. Quando la scheda *Hachiko* esegue una lettura dei dati d'esercizio e rileva dei flag *Alarm* ad `ON`, i valori letti vengono inseriti in *AlarmValue*: l'entità rappresentante i dati d'esercizio affetti da anomalie. Quest'ultima è associata ad *Alarm* per collegare i dati letti agli *Alarm* generati. Essendo *AlarmValue* un valore di parametro, questo è collegato a *ParameterRegolator*.

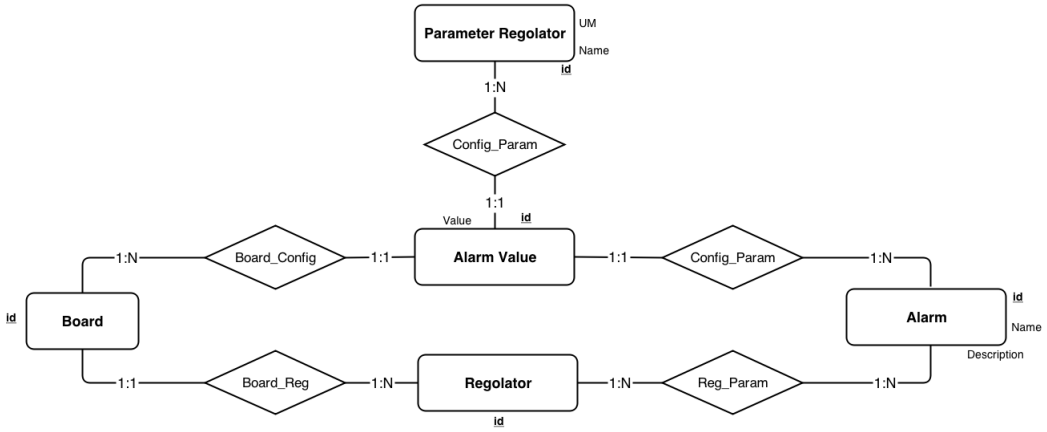

Figura 2.4: Schema *ER* riguardante gli *Alarm*. I dati d'esercizio che generano *Alarm*, sono memorizzati nella entità *AlarmValue*. L'associazione con *Alarm* genera la lista degli *Alarm* sollevati, mentre l'associazione con *ParameterGenerator* individua per ogni valore, il parametro corrispondente.

Assistenza. La seguente parte di schema modella le attività di assistenza. Inanzitutto, un'entità *Technician* riguarda i tecnici abilitati alle manutenzioni. Un *username* e una *password* sono necessari per poter implementare l'autenticazione remota del manutentore da parte delle *Hachiko*. Ogni attività d'assistenza è modellata dall'entità *Assistance*. Questa richiede la partecipazione obbligatoria delle entità *ConfigurationValue*, *Board* e *Techinician*. Ogni valore di configurazione deve perciò essere inserito mediante una sessione di assistenza. Anche i settaggi iniziali devono essere registrati come una normale assistenza. *Assistance* ha attributi binari per permettere l'inserimento delle immagini e dei file di *log*. Uno di questi file serve per contenere tutte le azioni svolte dal tecnico durante la manutenzione.

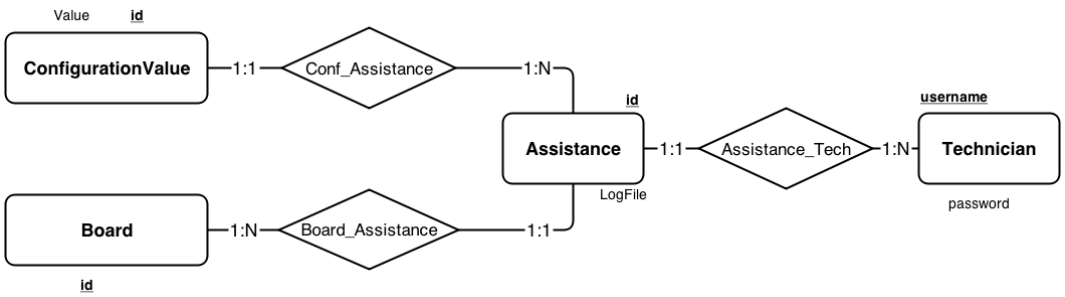

Figura 2.5: Schema *ER* per le operazioni di assistenza. *Assistance* è un'entità associata al tecnico che esegue le operazioni, alla scheda *MEC* che la riguarda e ai valori di configurazione finali che sono stati impostati.

Modellazione dei generatori. I generatori sono stati modellati in modo simile a quanto fatto per i regolatori. In questo modo è possibile inserire nuovi generatori con parametri differenti da quelli già presenti nel sistema. *Generator* rappresenta una singola istanza di generatore elettrico identificato dalla propria matricola. *TypeGenerator* rappresenta il tipo di generatore. *ParameterGenerator* è un entità che modella un singolo parametro. Tramite l'associazione con *TypeGenerator* si costruisce la lista dei parametri del singolo generatore, mentre l'associazione con *ValueGenerator* ne da un valore.

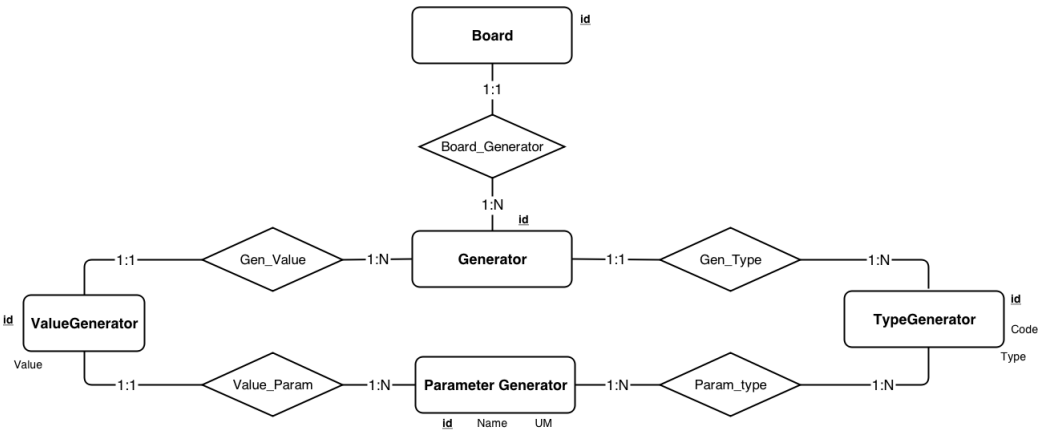

Figura 2.6: Lo schema per la modellazione dei generatori, della loro tipologia e dei loro parametri.

2.3 Progettazione logica

La progettazione logica ha lo scopo di tradurre lo schema *ER* in uno schema relazionale, facilmente implementabile poi nello specifico linguaggio del *DBMS*. Sono state utilizzate le forme di conversione standard per passare da un modello all'altro. Di seguito verranno presentate le tabelle che compongono lo schema.

Le entità *Board*, *Hachiko* e *Customer* vengono convertite in tabelle nel modello relazionale. L'unica associazione tra queste entità a diventare una relazione è *Hachiko_Board* che, con il suo *timestamp*, rappresenta lo storico dei diversi collegamenti tra scheda Hachiko e regolatori. L'associazione tra *Customer* e *Board* viene fatta incorporando in *Board* la chiave primaria di *Customer*.

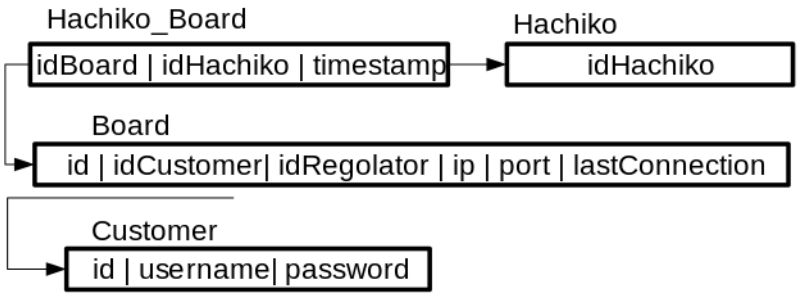

Figura 2.7: Schema relazionale ottenuto dalle entità *Board*, *Hachiko* e *Customer*.

La parte del *ER* riguardante i parametri di configurazione viene convertita nello schema di figura 2.8. Il campo *value* contenuto nello schema *ER* viene

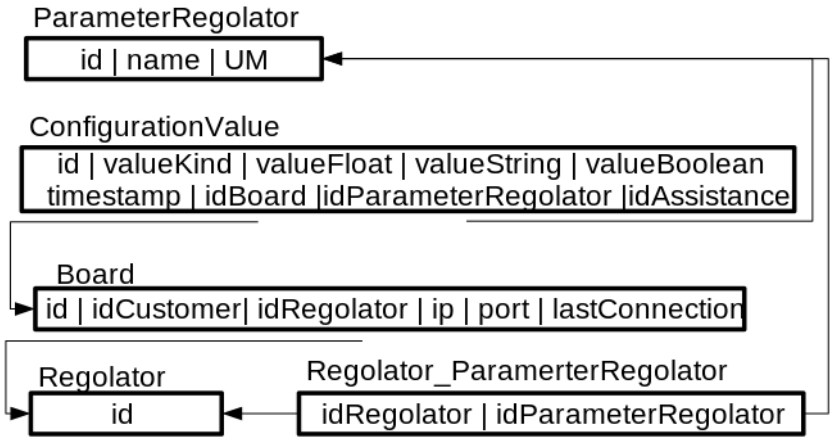

Figura 2.8: Lo schema relazionale per i parametri di configurazione.

trasformato negli attributi *valueKind*, *valueFloat*, *valueString* e *valueBoolean*. Questa conversione è necessaria in quanto i parametri del regolatore non sono tutti dello stesso tipo. Così, *valueKind* contiene un flag per indicare la tipologia del valore. In base a questo il dato sarà inserito in uno dei tre campi menzionati.

Viene introdotta la tabella *Regolator_ ParameterRegolator* per rispondere ai vincoli di cardinalità dettati dallo schema *ER*. Grazie a questa relazione *Regolator* può essere associato a più *ParameterRegolator* e viceversa.

Nello schema relazionale i valori d'esercizio rispecchiano fedelmente lo schema *ER*. Essendo questi dati rappresentabili tramite valori numerici, è necessario solamente il campo *value* di tipo *float*. Nella tabella *Log* l'attributo *logFile* serve per contenere il file aggregato come una sequenza di byte.

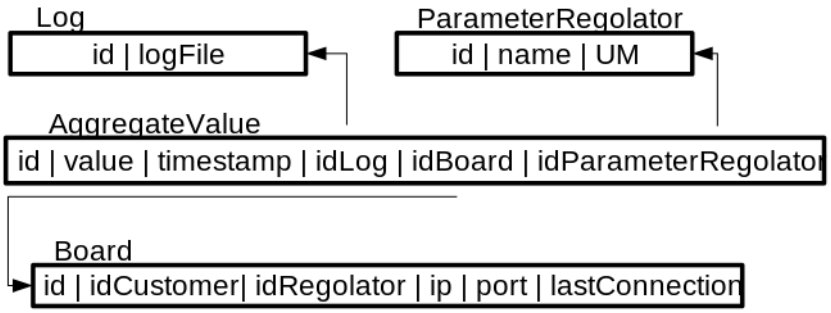

Figura 2.9: Lo schema relazionale per i dati d'esercizio.

La figura 2.10 rappresenta lo schema relazionale riguardante la gestione degli *Alarm*. *Regolator_Alarm* modella l'associazione tra *Regolator* e *Alarm* dello

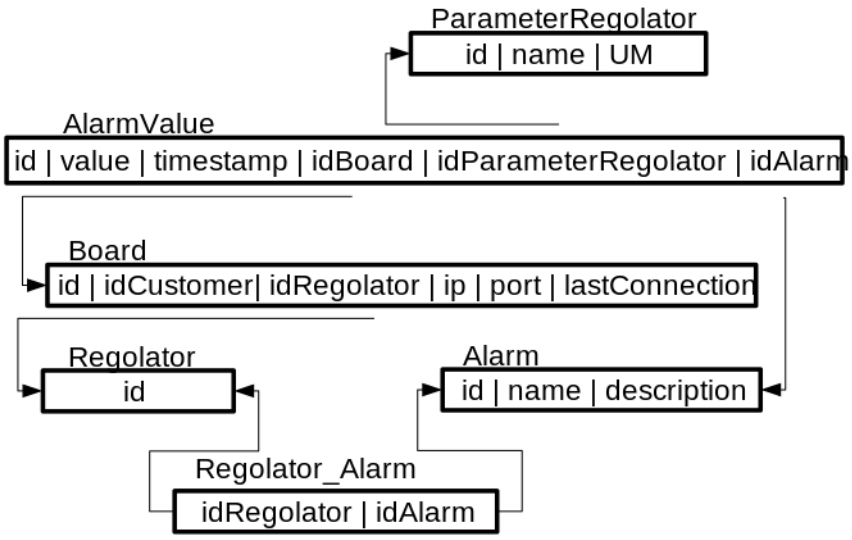

Figura 2.10: Lo schema relazionale per gli *Alarm*.

schema *ER*. In questo modo un regolatore piò avere più *Alarm* e viceversa. Le restanti tabelle derivano direttamente dalle rispettive entità del *ER*.

L'entità *Assistance* convertita nell'omonima relazione è associata a *Techinician*, *Board* e *ConfigurationValue*. Tutte le associazioni sono fatte tramite l'in-

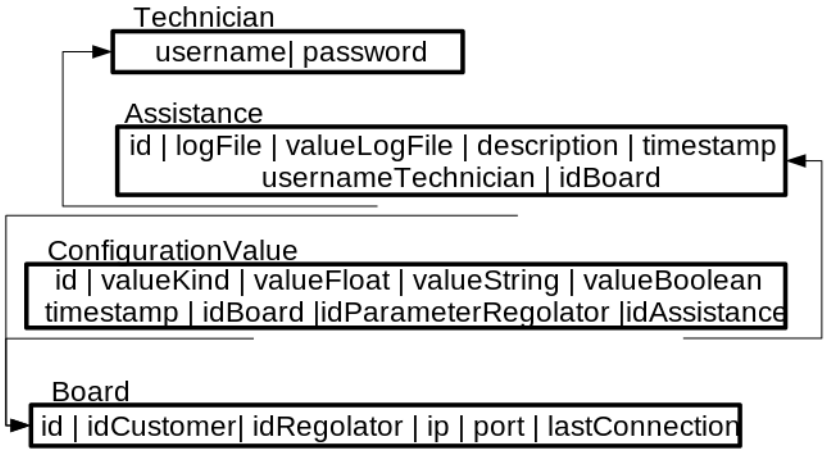

Figura 2.11: Lo schema relazionale relativo alle manutenzioni dei parametri di configurazione.

serimento di chiavi esterne nelle varie relazioni. L'attributo *logFile* serve per contenere il file con l'elenco delle operazioni fatte durante la sessione di manutenzione. *valueLogFile* è utilizzato per contenere i valori dei dati di esercizio letti durante la sessione di assistenza. Questi valori non vanno mescolati con i normali dati d'esercizio, poiché "sporcati" dall'intervento del tecnico.

Le relazioni riportate in figura 2.12 riguardano infine i dati relativi ai generatori, il loro tipo e parametri. Le associazioni che nel modello *ER* avevano cardinalità multipla sono state trasformate in relazioni.

2.3.1 Vincoli di dominio e di tupla

Sono stati utilizzati, oltre alle usuali tipologie di attributi, alcuni domini personalizzati.

Gli indirizzi *ip* vengono memorizzati come attributi *string* e viene controllata la presenza dei puntini tra i numeri. Le porte, rappresentate da *integer*, devono essere limitate superiormente da $2^{16} = 65536$.

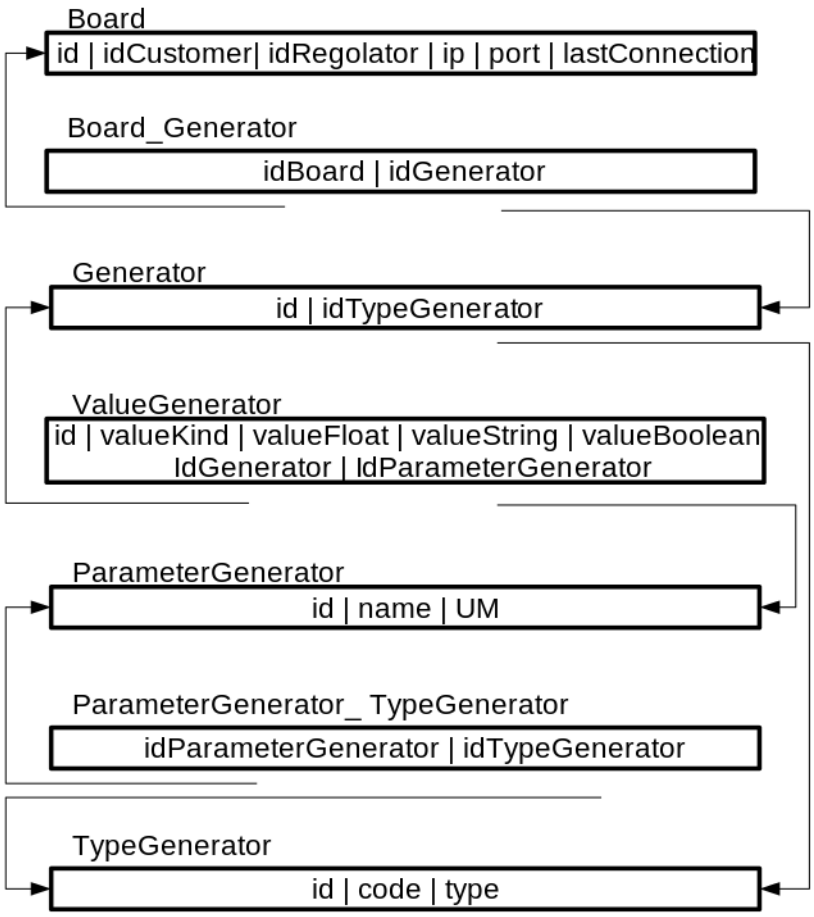

Figura 2.12: Lo schema relazionale per la modellazione dei generatori, della loro tipologia e dei loro parametri.

Gli identificativi di *ParameterGenerator* e *Alarm* sono delle stringhe formate da 6 caratteri. Tra i caratteri non ammessi vi è lo space e il cancelletto. Quest'ultimo viene utilizzato come carattere d'*escape* o per indicare i commenti nei file di testo formattati.

Gli attributi di tipo *timestamp* vengono instanziati automaticamente all'inserimento della tupla.

Nelle tabelle *ConfigurationValue* e *ValueGenerator* esistono i vincoli che seguono. L'attributo *valueKind* di tipo *char* può assumere solamente i valori `s`, `f`, o `b`, a seconda se il valore è di tipo *string*, *float* o *boolean*. È l'applicazione poi a gestire gli inserimenti e le letture in base a questo *flag*. *valueString* è

perciò di tipo stringa, *valueFloat* è un numero decimale e *valueBoolean* è un valore booleano.

Gli attributi *password* sono destinati a memorizzare le *password* all'interno del data base. Per non salvarle in chiaro, il *DB* memorizza il loro *hash MD5*.

I campi binari vengono memorizzati come *array* di byte. Un file viene in questo modo copiato all'interno del data base.

2.3.2 Vincoli esterni

Per come è stato concepito il data base, vi è una necessità di inizializzazione dei dati che non può avvenire dopo che le schede inizino il loro funzionamento. In particolare ogni regolatore e scheda *Hachiko* devono essere inseriti all'interno del data base prima di avviare la procedura di raccolta dati. Anche i codici dei parametri e i rispettivi collegamenti con le tipologie di regolatori devono essere inseriti prima di avviare l'applicazione. Le schede, infine, devono essere associate ai rispettivi clienti, in modo da poter essere consultate.

I parametri di configurazione sono sempre associati ad una assistenza. Tuttavia, nel caso in cui a *runtime* venga rilevata una nuova configurazione, questa deve essere memorizzata all'interno della base di dati, anche se non vi è stata alcuna manutenzione da parte di un tecnico. In questo caso speciale verrà utilizzata un'istanza di assistenza associata ad un tecnico con *username* `NOTAUTHORIZED`. Tutte le manutenzioni di questo particolare tecnico sono state fatte senza l'ausilio della scheda *Hachiko*.

2.4 Costruzione del data base

Il *Data Base Managment System* utilizzato è *PostgreSQL*. La traduzione dello schema relazionale in linguaggio *SQL* si è dimostrata semplice. Si è trattato di implementare le tabelle utilizzando la sintassi e i costrutti del *DBMS* scelto.

Per semplificare le modifiche alle varie tabelle nella fase di sviluppo dell'applicazione è tornato utile inserire il database all'interno di uno schema. Inoltre il codice per la costruzione della base di dati è stato inserito in un file con estensione *sql*. In questo modo, per modificare la base di dati si modifica il file in questione per poi eseguirlo. Anche per gli inserimenti dei parametri e delle schede si è usato questo approccio.

Il codice *SQL* per la costruzione del data base è riportato in Appendice A.

Capitolo 3

Architettura del sistema

In questa sezione viene presentata l'organizzazione del software sia della scheda *Hachiko* che della applicazione server. Vengono illustrate l'architettura del sistema, il modello di comunicazione adottato e il funzionamento delle varie componenti software.

3.1 Struttura generale della applicazione

Il sistema è stato realizzato secondo un'architettura di tipo *client-server*. Il server è un elaboratore centrale dotato di una applicazione dedicata - il tema principale di questo elaborato - mentre il ruolo del client è rappresentato dalla scheda *Hachiko*. È stato adottato un modello a tre livelli. A differenza però dell'architettura *three-tier* utilizzata dalle applicazioni *web based*, in questo caso la parte di presentazione normalmente gestita dal *browser*, è sostituita dal preprocessing dei dati. Il software installato sulla scheda *Hachiko* ha infatti lo scopo di rilevare i dati d'esercizio.

La comunicazione tra client e server fa utilizzo di socket *TCP*. Il server si mette in ascolto su una determinata porta in attesa di messaggi da parte delle schede *Hachiko*. Ogni client può inviare diverse richieste in base al tipo di aggiornamento dei dati (dati d'esercizio, *Alarm* o parametri di configurazione) oppure per effettuare alcune richieste precise (inserimento di una manutenzione o autenticazione verso il sistema). Le diverse interazioni sono riportate nella sezione dedicata, con la rispettiva struttura dei pacchetti da trasmettere per ciascuna comunicazione.

Ad ogni richiesta corrisponde una sessione diversa di comunicazione *TCP*. Così viene aperta una porta differente per ciascuna richiesta, che viene chiusa

nel momento in cui tutti i dati vengono consegnati. Non tutte le richieste necessitano di una risposta da parte del server. In questo caso, la comunicazione dell'avvenuta ricezione di tutti i pacchetti corrisponde con la chiusura della comunicazione da parte del server. Il protocollo *TCP* garantisce l'ordinamento, l'effettiva ricezione e l'integrità di tutti i pacchetti. Nello sviluppo delle applicazioni client e server sono perciò date per scontate e indispensabili queste tre funzionalità.

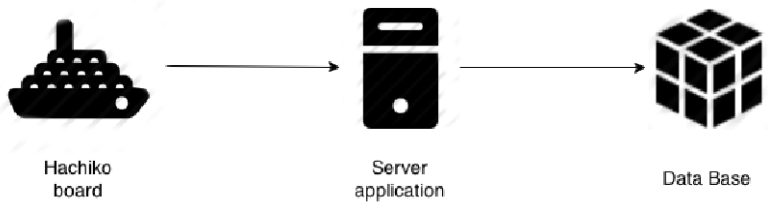

Figura 3.1: Architettura *Three-tier*. La scheda *Hachiko*, installata sul regolatore della nave esegue elaborazioni per l'aggregazione e il rilevamento dei dati. Per comunicare con il server si utilizzano pacchetti costruiti ad hoc in base alle richieste. Il server gestisce la logica dell'applicazione, smista le varie richieste e si occupa di interagire con il data base. La gestione dei dati è affidata al *Data Base Managment System*.

Si è impropriamente detto che l'architettura è di tipo *client-server*. Questo risulta vero se ci si concentra solamente sulle interazioni che la scheda *Hachiko* fa nel suo normale funzionamento. In futuro dovrà essere possibile configurare un regolatore da remoto. Per permettere questo tipo di interazione la comunicazione deve iniziare dal server. Le schede *Hachiko* muovendosi per il mondo variano il loro indirizzo *ip* e la connettività Internet non è sempre garantita. Benché questa funzionalità non è ancora stata implementata, l'applicazione server rileva l'ultimo indirizzo che la scheda *Hachiko* utilizza per la comunicazione e mette a disposizione una primitiva per l'invio di dati verso un particolare regolatore. Il software installato sulla scheda *Hachiko* dovrà implementare un server *TCP* che resti in ascolto delle richieste per le configurazioni. L'architettura muterebbe in questo modo verso una architettura di tipo paritario.

3.2 Organizzazione del software

L'applicativo server è stato sviluppato in *Java*, mentre il programma destinato alle schede *Hachiko* in `C++`. L'intera progettazione è avvenuta in modo modulare. In questo modo, diventa più semplice modificare le componenti software in

futuro. Una progettazione *thread safe* permette la parallelizzazione del server tramite semplici modifiche software.

Quattro moduli compongono l'applicazione server. Tre di questi implementano tutte le funzionalità base del applicativo e sono i moduli *Comunicazione Remota*, *Gestione delle Richieste* e *Accesso al Data Base*. Un ulteriore modulo si occupa degli aspetti di sicurezza, come la gestioni delle chiavi asimmetriche di cifratura. Il modulo per la comunicazione remota si occupa di accogliere le richieste provenienti dalle schede *Hachiko* per poi consegnarle al modulo che le gestisce. Questo le smista a seconda della loro tipologia e comunica con il modulo di accesso al database per interagire con il *DBMS*.

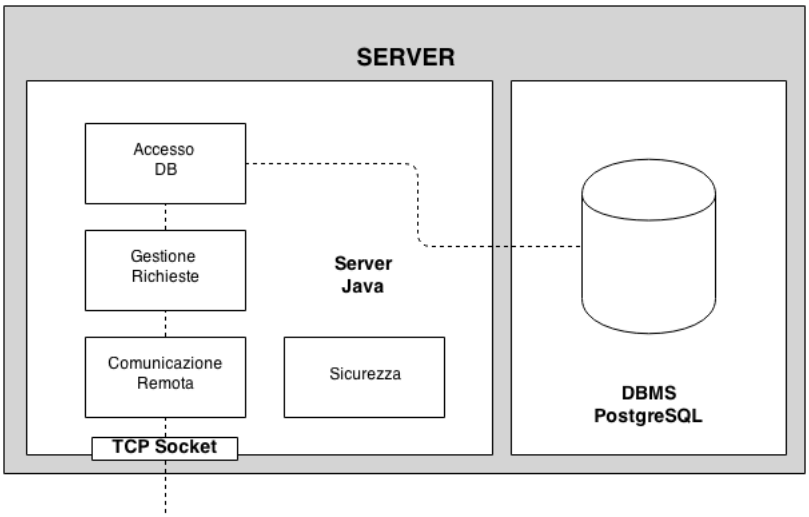

Figura 3.2: Il server, nel suo complesso è formato da una applicazione Java e da un DBMS. Quattro moduli compongono il programma Java. Questi garantiscono la comunicazione con la scheda *Hachiko*, lo smistamento delle varie richieste, l'accesso alla base di dati e alcune operazioni di sicurezza di contorno.

In Figura 3.3 è riportata la struttura della applicazione *Hachiko*. Tra i moduli indispensabili troviamo il modulo di comunicazione remota per gestire le interazioni con il server, il modulo di interfaccia *MEC-100* per il dialogo tra le due schede ed il modulo di elaborazione dei dati di esercizio. Funzionalità aggiuntive vengono dal modulo per gestire la autenticazione del tecnico, il modulo *GUI* per l'interazione con lo schermo *touchscreen* e il modulo per la gestione degli aggiornamenti software. Infine, i moduli *Heartbeat* e collegamento server gestiscono la creazione dei pacchetti delle richieste, all'interpretazione delle risposte dal server e alla gestione del protocollo *Heartbeat*.

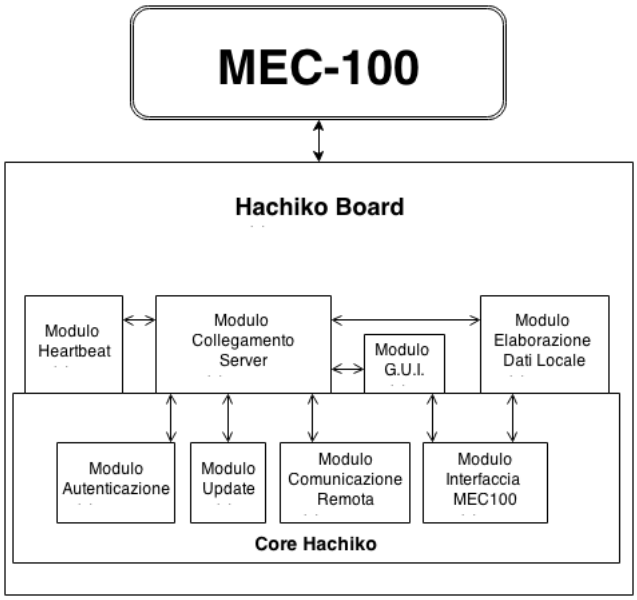

Figura 3.3: Organizzazione software dell'applicativo installato sulle schede *Hachiko*.

3.2.1 Interazioni tra la scheda Hachiko e il Server

Per la comunicazione tra le schede *Hachiko* ed il server è stato definito un protocollo a cui devono attenersi tutte le interazioni. Una scheda *Hachiko* attualmente può comunicare con il server tramite 7 tipologie di messaggi. Ad ogni messaggio corrisponde una specifica richiesta. Questo sono qui di seguito riportate.

Autenticazione nel sistema. La scheda *Hachiko*, prima di iniziare con il suo normale funzionamento, deve autenticarsi verso il sistema. A tale scopo deve comunicare al server alcune informazioni essenziali, come il codice identificativo che la caratterizza ed il codice seriale del regolatore collegato. In questo modo è possibile mantenere lo storico delle associazioni tra le schede. Il server, a seguito di questa richiesta, può fornire alcune informazioni utili al funzionamento del client, come la chiave pubblica di criptazione del server. Ogni volta che una scheda *Hachiko* viene accesa o collegata ad un regolatore, un messaggio di questo tipo deve essere inviato.

Bootstrap. La richiesta di bootstrap viene inviata sempre e solo dopo l'invio del messaggio di autenticazione. La scheda client deve comunicare al server la configurazione dei parametri del regolatore. Il server, confrontandoli

con l'ultima configurazione registrata è in grado di capire se sono state fatte manutenzioni nel periodo di inattività della scheda.

Dati di esercizio. L'aggiornamento dei dati di esercizio avviene comunicando al server i loro valori mediati su diverse letture ed un file contenente i valori puntuali. Il client *Hachiko* rileva i dati di esercizio ogni secondo, ma solo periodicamente li comunica al server. Nel caso in cui le letture contengono degli *Alarm* la comunicazione viene però fatta immediatamente. Se vi sono delle anomalie i dati contenuti nel server sono, Internet permettendo, sempre aggiornati.

Alarm. Uno speciale messaggio è inviato quando viene rilevato un *Alarm*. Il server riesce in questo modo a gestire i dati relativi all'*Alarm* in modo indipendente dalle richieste di aggiornamento dei dati di esercizio.

Heartbeat. Con una frequenza prefissata di un minuto la scheda *Hachiko* invia un messaggio di *Heartbeat* al server. Il server aggiorna l'indirizzo *ip* e il *timestamp* relativo all'ultimo avvistamento. Se una scheda ha *timestamp* recente, molto probabilmente è raggiungibile tramite Internet e il server dispone dell'indirizzo *ip* per farlo.

Login. Per autenticare un tecnico prima di una manutenzione, il client invia *username* e *password* al server. La risposta di quest'ultimo determina se il tecnico è stato autenticato correttamente. L'operazione di login è sempre necessaria prima di effettuare un'assistenza dalla scheda *Hachiko*.

Maintanence. Al termine di ciascuna sessione di assistenza, il client invia al server la nuova configurazione dei parametri, un file contenente tutte le operazioni eseguite dalla scheda e i dati di esercizio letti durante le operazioni di manutenzione.

Risulta molto importante per il funzionamento del sistema l'ordine in cui queste richieste vengono effettuate. L'autenticazione e il *bootstrap* vanno eseguite ogni qualvolta la scheda viene alimentata o collegata ad un regolatore. Le restanti richieste possono avvenire in ordine casuale, con l'unico vincolo che il *login* del tecnico deve essere utilizzato per la manutenzione e quindi precedere temporalmente il messaggio di *maintanence*.

3.2.2 Packet design

Ogni richiesta da parte del client avviene tramite protocollo *TCP* e lo scambio di pacchetti orientati al byte. I dati da trasferire devono rispettare una particolare struttura che varia da richiesta a richiesta, ma utilizzano però uno stesso *header* riportato in Figura 3.4. Questo contiene informazioni riguardanti il codice della richiesta e gli identificativi seriali delle due schede.

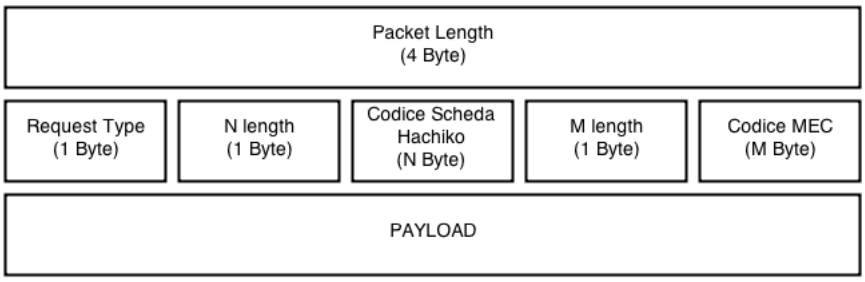

Figura 3.4: Struttura dei dati delle richieste client. L'*header* è comune a tutte le richieste. Queste sono identificabili da un campo contenente il codice che le identifica. Contiene inoltre i codici *Hachiko* e *MEC-100* a cui il pacchetto fa riferimento. Il payload varia invece in base alla richiesta.

I primi 4 byte del messaggio contengono un intero *unsigned* indicante la lunghezza, in numero di byte, dei dati trasmessi. Il campo *Reguest Type* contiene un codice che identifica la natura della richiesta. Esiste un codice per ciascuna delle interazioni possibili.

Codice	Richiesta
0	autenticazione
1	bootstrap
2	dati esercizio
3	alarm
4	heartbeat
5	login
6	manutenzione

Tabella 3.1: Codici delle possibili richieste che il client invia al server.

Il byte successivo contiene la lunghezza dell'identificativo della scheda *Hachiko*. Subito dopo si ha proprio questo identificativo. Stessa cosa viene fatta per il codice seriale del *MEC-100*. La codifica dei codici seriali è sempre accompagnata dalla loro lunghezza, in questo modo dispositivi futuri potranno

avere codici identificativi di lunghezza differente.

Il pacchetto con codice 0, relativo alla autenticazione che la scheda effettua verso il sistema, non invia ulteriori dati nel campo *payload*. La coppia degli identificativi è tutto ciò che serve in questa fase.

Il pacchetto di *bootstrap* invia invece delle informazioni aggiuntive. Come *payload* viene inviato un file testuale contenente i parametri di configurazione del regolatore. Ciascuna riga del file contiene un parametro di configurazione nel formato `PARAMETRO=VALORE`. Il valore può essere numerico, una stringa oppure un booleano.

I dati relativi ai valori di esercizio vengono trasmessi nel *payload* della richiesta con codice 2. Il primo byte contiene il loro numero totale. Per ogni parametro viene trasmesso il codice e il rispettivo valore, tramite coppie di 6 byte per il codice e di 4 byte per il valore *float* corrispondente. Sempre nel *payload* viene inserito un file testuale contenente tutte le letture puntuali. Questo file rappresenta il *log* da inserire nell'omonima tabella del data base.

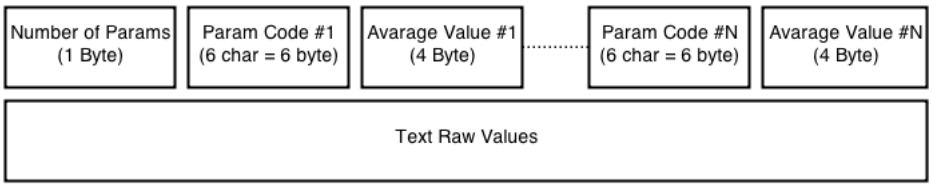

Figura 3.5: Payload relativo alla richiesta di aggiornamento dei dati di esercizio. Per ciascun parametro viene trasmesso il codice ed il rispettivo valore aggregato. I valori puntuali sono contenuti in un file testuale e racchiudono tutte le letture fatte dalla scheda *Hachiko*.

Nel *payload* della richiesta relativa agli *Alarm* viene inserito un file contenente i valori d'esercizio letti in presenza di anomalie ed i codici degli *Alarm* rilevati.

Il pacchetto di *Heartbeat* non prevede alcun *payload*. Ha come unico scopo quello di comunicare al server che la scheda è attiva e connessa a Internet.

Nel pacchetto di *login* il client deve trasmettere *username* e *password* del tecnico. Essendo la lunghezza di questi due valori variabile, due campi da 1 byte ciascuno contengono la loro lunghezza. Il server deve fornire una risposta alla richiesta. A tal scopo un byte di soli 1 indica la corretta autenticazione, mentre

un byte contenente soli 0 sta a indicare l'errata combinazione delle credenziali d'accesso.

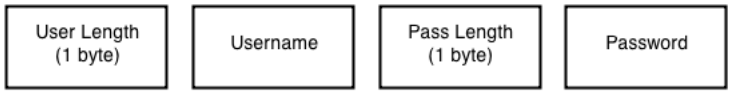

Figura 3.6: Payload relativo alla richiesta di login. Le credenziali del tecnico vengono comunicate al server che ne verifica la correttezza.

L'ultima richiesta possibile riguarda l'inserimento da parte del client di una attività di manutenzione. In questa richiesta vengono comunicati l'*username* del tecnico coinvolto e tre file testuali. Per determinare le varie lunghezze dei campi ci sono altrettanti campi destinati a contenerle. Tra i file trasmessi il primo contiene i valori finali dei parametri di configurazione, il secondo l'elenco delle modifiche che il tecnico ha apportato e l'ultimo contiene i valori rilevati dalle letture durante la sessione di manutenzione.

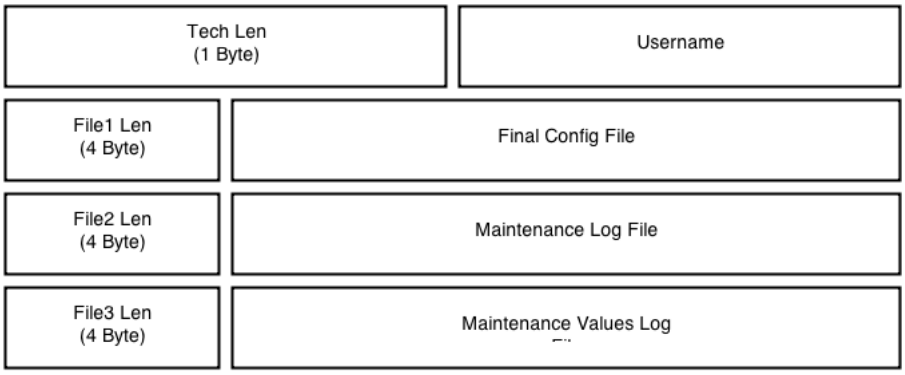

Figura 3.7: Payload di una richiesta per l'inserimento di una manutenzione. L'inserimento dell'assistenza richiede l'*username* del tecnico e i tre file contenenti la configurazione finale, le modifiche fatte dal tecnico e i valori letti durante la sessione d'assistenza.

Capitolo 4

Il Server

4.1 Diagramma di flusso dell'applicazione

Per gestire le diverse richieste concorrenti provenienti dalle applicazioni client, un server deve avere una architettura multi *thread*. La gestione delle richieste deve essere fatta in parallelo, senza dover aspettare che la richiesta precedente sia stata servita prima di procedere con la successiva. Risulta però necessario, per mantenere validi i dati del data base, che le richieste provenienti da uno stesso client vengano serializzate. Alcune di queste, le più complesse, devono effettuare diverse operazioni in base alla configurazione dei dati nel data base, che non devono variare nell'arco dell'esecuzione. Per far fronte a questa problematica si è fatto utilizzo di un meccanismo di *lock*. Il server associa a ciascuna scheda un *lock* differente che viene controllato prima di procedere con l'esecuzione di una sua richiesta. Non è invece garantito alcun ordine con cui queste vengono eseguite. Per imporre un ordinamento dev'essere il client ad inviare la richiesta successiva quando quella precedente è stata servita. La maggior parte delle azioni che il client può richiedere sono tuttavia intercambiabili in termini temporali.

Il programma principale avvia un *thread* auto sospensivo che si mette in attesa di richieste da parte dei client. Per ogni nuova connessione viene creato un nuovo *thread* che si occupa della gestione del singolo *socket*. Java tratta ciascun input come uno *stream* di byte. Non c'è dunque alcuna informazione su dove termini ciascun messaggio. Per far fronte a questa necessità è stata introdotta nei primi byte la lunghezza totale del messaggio. I dati trasmessi possono perciò venir rilevati nella loro interezza.

Dall'*header* vengono estratti i codici seriali delle schede coinvolte e l'identi-

ficativo della richiesta. È in base a quest'ultimo che viene selezionato il metodo da eseguire. Questi metodi implementano tutta la logica della applicazione. Alcuni aggiornano il data base in base alle richieste, mentre altri inviano risposte al client. I metodi di accesso al *DB* sono tutti contenuti in una classe per formare un'unica libreria.

L'immagine 4.1 riporta il diagramma di flusso seguito dall'applicazione. Le entità *ReguestManager* e *DBManager* sono delle raccolte di funzioni, quindi il loro flusso d'esecuzione fa parte del *thread* che gestisce la richiesta.

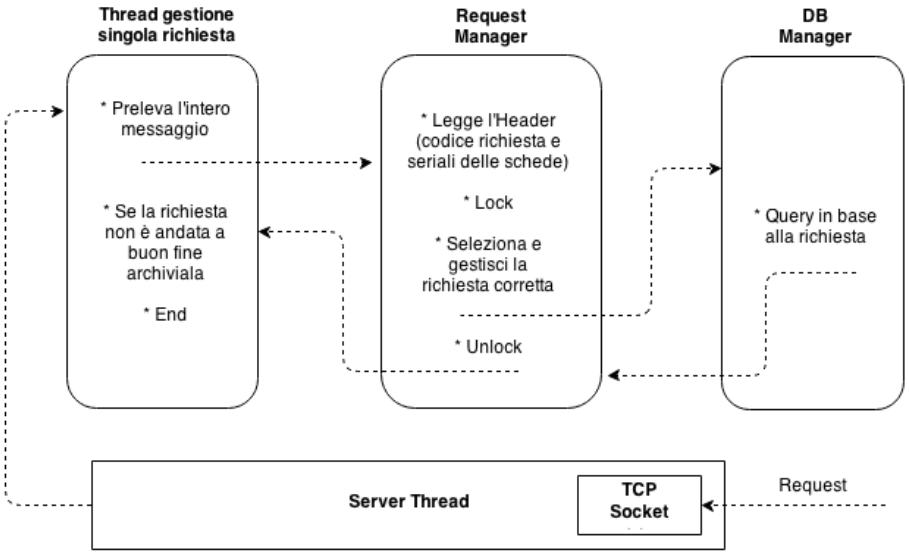

Figura 4.1: Flusso d'esecuzione per ciascuna richiesta. La struttura a *thread* garantisce la concorrenza delle richieste.

4.2 Gestione delle singole richieste

In questa sezione viene presentata la logica che risiede dietro alle richieste. Ciascuna interazione tra client e server si traduce in alcune interrogazioni al data base, verranno quindi riportate le *query* nel linguaggio *SQL*. L'applicazione mantiene inoltre una tabella con le corrispondenze tra regolatore e il suo ultimo indirizzo *ip*. Tutte le richieste contribuiscono all'aggiornamento di queste associazioni.

Autenticazione. La fase di autenticazione fa utilizzo unicamente dei codici seriali della scheda *Hachiko* e del regolatore. L'applicazione lancia una *query* al *DBSM* per verificare l'ultimo collegamento tra le due schede. Se la tupla di risposta ha i due identificativi uguali a quelli contenuti nella richiesta allora il collegamento registrato nel data base non necessita di alcuna modifica.

```
SELECT * FROM CLOUD.HACHIKO_BOARD
WHERE idHachiko = ? OR idBoard= ?
ORDER BY tmstp DESC LIMIT 1;

INSERT INTO CLOUD.HACHIKO_BOARD
(idHachiko,idBoard) VALUES(?,?);
```

Il server provvede all'invio di alcune informazioni di configurazione al client. Al momento viene inviata la chiave pubblica per la criptografia asimmetrica dei messaggi. Dopo di che il *socket* viene chiuso.

Bootstrap. Oltre alle informazioni sui seriali delle schede il messaggio contiene un file di testo con i parametri di configurazione del regolatore. Il metodo di gestione della richiesta per prima cosa preleva tutti i valori di configurazione ed i rispettivi codici dei parametri dal file testuale. Dopo di che, per ciascun parametro, viene chiesto al *DB* l'ultimo valore memorizzato.

```
SELECT * FROM CLOUD.CONFIGURATIONVALUE
WHERE idBoard= ? AND idParameterRegolator= ?
ORDER BY tmstp DESC LIMIT 1;
```

Si verifica a questo punto se il valore nel data base corrisponde a quello rilevato dalla scheda *Hachiko*. Se i valori sono uguali si prosegue con il parametro successivo. In caso contrario c'è stata una manutenzione non autorizzata dalla scheda *Hachiko* che ha portato ad un cambiamento del parametro. La configurazione viene ugualmente memorizzata, ma viene associata una assistenza effettuata da un tecnico speciale con *username* `NOTAUTHORIZED`.

```
INSERT INTO CLOUD.ASSISTANCE(description,
usernameTechnician, idBoard) VALUES
("Unauthorized maintenance session",
"NOTAUTHORIZED", ?);
```

L'inserimento del valore di configurazione utilizza come *idAssistance* il codice generato dall'inserimento della assistenza. Ciascuna configurazione nuova

utilizza questo identificativo negli inserimenti. Essendo i valori di configurazione differenti per tipologia del valore del parametro, sono state previste tre *query* diverse da utilizzare in base al tipo del parametro.

```
INSERT INTO CLOUD.CONFIGURATIONVALUE(valueBoolean,
idBoard, idAssistance, idParameterRegolator, valueKind)
VALUES (?,?,?,?,?);

INSERT INTO CLOUD.CONFIGURATIONVALUE(valueFloat,
idBoard, idAssistance, idParameterRegolator, valueKind)
VALUES (?,?,?,?,?);

INSERT INTO CLOUD.CONFIGURATIONVALUE(valueString,
idBoard, idAssistance, idParameterRegolator, valueKind)
VALUES (?,?,?,?,?);
```

Aggregate Values. Dalla richiesta vengono prelevati il file contenente i parametri puntuali e i valori aggregati con i loro rispettivi codici parametro. Per prima cosa si inserisce il file così com'è nella tabella *Log*.

```
INSERT INTO CLOUD.LOG(data) VALUES (?);
```

Per ciascun valore aggregato si effettua un inserimento nella tabella *AggregateValue* collegandolo al rispettivo parametro. La chiave identificativa generata per la tupla di *Log* viene utilizzata in questo aggiornamento.

```
INSERT INTO CLOUD.AGGREGATEVALUE(idParameterRegolator,
value, idBoard, idLog) VALUES (?,?,?,?)
```

Alarm. Nella gestione della richiesta viene dapprima analizzato il file testuale ricevuto come parametro. Vengono quindi estratti i valori d'esercizio e i codici degli *Alarm* rilevati dal *client*. Il metodo deve gestire due tipologie di inserimenti. I dati d'esercizio vanno inseriti nella tabella *AlarmValue*. Questa tabella sostituisce la relazione *AggregateValue* per i valori affetti da anomalie. Il secondo inserimento riguarda il collegamento dei valori con i codici degli *Alarm* rilevati.

A tale scopo il metodo inserisce iterativamente un parametro per poi inserire i collegamenti con i vari *Alarm*. La struttura delle iterazioni è a doppio *loop*. Nel ciclo esterno si inseriscono i valori, mentre in quello interno si generano i collegamenti con gli *Alarm* per ciascun valore. Le due *query*, una per *loop*, sono di seguito riportate.

```
 INSERT INTO CLOUD.ALARMVALUE(val, idBoard,
idParameterRegolator) VALUES (?,?,?);

INSERT INTO CLOUD.ALARMVALUE_ALARM(idAlarmValue,
idAlarm) VALUES (?,?);
```

Heartbeat. Una richiesta *Heartbeat* aggiorna il *timestamp* relativo all'ultima notifica da parte della scheda *Hachiko*. Un client invia questo tipo di richiesta con una frequenza molto elevata con lo scopo principale di aggiornare l'indirizzo *ip* della scheda. Questo dovrebbe permettere di raggiungere il regolatore nelle future configurazioni da remoto.

```
UPDATE CLOUD.HACHIKO_BOARD
SET tmstp = CURRENT_TIMESTAMP
WHERE idBoard = ? AND idHachiko = ?;
```

Login. Dalla richiesta di *login* viene prelevato *username* e *password* del tecnico. Tramite una *query* si verifica l'effettiva autenticazione, confrontanto le credenziali d'accesso ricevute con quelle memorizzate nel *DB*. Le *password* all'interno del database sono memorizzate con codifica *MD5*.

```
SELECT * FROM CLOUD.TECHNICIAN WHERE
username = ? AND password = MD5(?);
```

Al client viene inviato un byte di soli 1 se l'autenticazione ha esito positivo, di soli 0 altrimenti.

Manutezione. Tramite la richiesta di assistenza si vuole inserire una nuova *entry* nella tabella *Assistance* relativa alle manutenzioni. Ogni tupla della relazione deve contenere l'*username* del tecnico che l'ha eseguita, il codice seriale della scheda coinvolta e i file contenenti rispettivamente i passi della assistenza e i dati di esercizio che sono stati letti nel frattempo. Un ulteriore file contenente la configurazione finale è trasmesso nel messaggio. L'inserimento della assistenza avviene tramite la seguente *query* di inserimento.

```
INSERT INTO CLOUD.ASSISTANCE(idBoard, usernameTechnician,
logFile, valueLogFile) VALUES (?,?,?,?);
```

La chiave primaria generata per questa nuova entry viene memorizzata ed utilizzata per gli inserimenti successivi riguardanti i singoli valori di configura-

zione. A seconda della tipologia del parametro si utilizza una delle *query* che seguono.

```
INSERT INTO CLOUD.CONFIGURATIONVALUE(valueBoolean, idBoard,
idAssistance, idParameterRegolator, valueKind)
VALUES (?,?,?,?,?);

INSERT INTO CLOUD.CONFIGURATIONVALUE(valueFloat, idBoard,
idAssistance, idParameterRegolator, valueKind)
VALUES (?,?,?,?,?);

INSERT INTO CLOUD.CONFIGURATIONVALUE(valueString, idBoard,
idAssistance, idParameterRegolator, valueKind)
VALUES (?,?,?,?,?);
```

Capitolo 5

L'Applicazione Android

5.1 Server Android

Per la consultazione dei dati rilevati dalle schede *Hachiko*, è stata sviluppata un'applicazione Android. In futuro sarà resa disponibile ai clienti dei regolatori che utilizzano il servizio. L'applicazione è nata per la presentazione dei dati, ma più avanti sarà dotata di ulteriori funzionalità, come la configurazione remota dei regolatori. Il programma necessita di un server dedicato. Dovendo utilizzare alcune funzionalità già descritte fin'ora, per semplificare lo sviluppo si è utilizzata la stessa architettura software esposta nel Capitolo 4, riutilizzando parti del codice.

Il server per Android è perciò scritto in linguaggio Java e le comunicazioni avvengono tramite *socket*. L'utilizzo di Java ha, in questo caso, semplificato lo sviluppo della piattaforma permettendo, per esempio, l'invio diretto di oggetti dal server al client.

Il diagramma di flusso rimane quello riportato in figura 4.1. Tuttavia il *socket TCP* utilizza una porta differente, per permettere al server Android di essere collocato sullo stesso calcolatore dell'altro server. Nulla vieta però di installarlo su un computer diverso per non disturbare le richieste delle schede *Hachiko*. Le trasmissioni sono criptate dal protocollo *SSL*.

Anche in questo caso le comunicazioni utilizzano dei dati predisposti in un determinato modo che variano in funzione della richiesta. Al momento sono state implementate 5 differenti tipologie di messaggi.

Codice	Richiesta
0	Login
1	Valori aggregati dei dati d'esercizio
2	Dati di targa del regolatore
3	Valori dei parametri di configurazione
4	Verifica degli Alarm
5	Valori puntuali dei dati d'esercizo

Tabella 5.1: Codici delle possibili richieste che l'applicazione Android invia al server.

5.1.1 Organizzazione dei messaggi

Le comunicazioni utilizzano un *header* comune per tutti i messaggi. Un campo di 4 Byte contiene la lunghezza totale dei dati, dando la possibilità al server di ricostruire ciò che viene inviato. Il byte successivo contiene invece il codice della richiesta, scelto tra quelli riportati in Tabella 5.1. È in base a quest'ultimo che viene decisa l'azione da intraprendere.

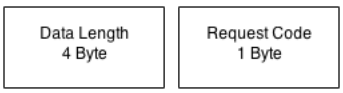

Figura 5.1: I primi 5 byte che compongono ciascun messaggio inviato dal client Android.

Per la richiesta di *Login* si devono fornire le credenziali di accesso dell'utente. Essendo *username* e *password* di lunghezza variabile, è necessario comunicare al server dove finisce un campo e dove inizia l'altro. Per questo si trasmettono le lunghezze dei rispettivi parametri. Se l'autenticazione ha buon fine, il server

Figura 5.2: Payload della richiesta di *Login*.

risponde inviando un byte composto di soli 1. In questo caso viene comunicata anche la lista dei regolatori dell'utente. Se l'autenticazione ha esito negativo, viene invece inviato un byte di soli 0.

Le richieste con codice da 1 a 4 trasmettono solamente il codice identificativo del regolatore. Variano però le azioni intraprese dal server, che risponde

inviando dati testuali secondo una formattazione comune. Ogni riga fa riferimento ad un *item* diverso: un valore aggregato, un dato di targa, un parametro di configurazione o un *Alarm*. Le informazioni di ciascun *item* sono separate da uno spazio e trasmesse nel formato `INFO=VALORE`. Il carattere *space* è *escaped* dal carattere cancelletto. È la logica della applicazione Android ad interpretare poi questi dati.

Figura 5.3: Payload delle richieste relative ai valori aggregati, ai dati di targa, ai parametri di configurazione e alla verifica degli *Alarm*.

Ad utilizzare un payload differente è invece la richiesta per ottenere i valori puntuali di un singolo parametro di lettura. Oltre all'identificativo del regolatore, negli ultimi 6 byte viene comunicato il codice del parametro d'interesse.

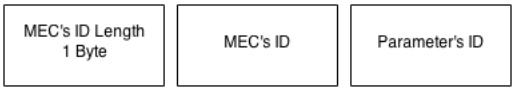

Figura 5.4: Payload per la richiesta dei valori puntuali di un dato d'esercizio.

5.1.2 Gestione delle signole richieste

Il server gestisce le richieste come in Figura 4.1. Le operazioni che deve eseguire dipendono esclusivamente dal codice della richiesta. Si utilizzano così *thread* separati per le varie esecuzioni. Dev'essere l'applicazione Android sul telefono a regolare l'ordine con cui vengono effettuate le varie richieste. Qui di seguito sono riportate, per ciascuna operazione, le interrogazioni che il server effettua verso il *DBMS* e la risposta che viene inviata al client.

Login. Il server gestisce la richiesta di Login interrogando direttamente la tabella *Customer* del data base. Se esiste una *entry* contenente la coppia *username* e *password* l'utente ha accesso al sistema.

```
SELECT * FROM CLOUD.CUSTOMER WHERE
username = ? AND password = MD5(?);
```

In caso di corretta autenticazione viene inviata la lista dei regolatori posseduti dal cliente. Per ciascuna scheda viene trasmesso anche l'identificativo del generatore cui è collegata. A tal scopo si è utilizzata la seguente *query*.

```
SELECT B.id, G.idGenerator, B.idRegolator
FROM CLOUD.CUSTOMER AS C INNER JOIN
CLOUD.BOARD AS B ON (C.id = B.idCustomer)
INNER JOIN CLOUD.BOARD_GENERATOR
AS G ON (B.id = G.idBoard)
WHERE C.username = ?;
```

L'operazione di *join* è necessaria per risalire da ciascun regolatore al codice del relativo generatore. Il formato con cui vengono inviate le informazioni è qui riportato.

```
BOARD=id GENERATOR=idGenerator REGOLATOR=type
```

Ogni riga riferisce ad un regolatore diverso e contiene il suo codice, il generatore pilotato e la tipologia di regolatore.

Valori aggregati dei dati d'esercizio. Partendo dal codice seriale del regolatore si possono recuperare i dati aggregati relativi all'ultimo inserimento. Ciascun parametro deve comparire una sola volta e deve riferire all'aggiornamento più recente. Tramite una *query* innestata, si ricavano prima i valori aggregati, per poi risalire al nome completo e all'unità di misura del parametro mediante un'operazione di *join*.

```
SELECT * FROM (
    SELECT DISTINCT ON(idParameterRegolator) *
    FROM CLOUD.AGGREGATEVALUE WHERE idBoard=?
    ORDER BY idParameterRegolator, tmstp DESC) as T
INNER JOIN cloud.parameterregolator as P
ON (T.idParameterRegolator = P.id);
```

Anche in questo caso il server risponde inviando delle informazioni sotto forma di file testuale. Ogni riga contiene il codice del parametro, il suo nome completo, il relativo valore aggregato e l'unità di misura.

```
ID=idParameterRegolator PARAMETER=name VALUE=value UM=um
```

Dati di targa del regolatore. I dati di targa fanno parte dei parametri di configurazione. Questi riferiscono ad alcuni parametri nominali di funzionamento e alle versioni hardware e software del regolatore. Il server interroga

il *DB* tramite una *query* innestata. Per prima cosa, partendo da un determinato parametro di targa, si ottiene l'ultimo valore memorizzato. Da questo, tramite un'operazione di *join*, si ricavano i dati relativi al codice, al nome e all'unità di misura del parametro. Dato che questa *query* riguarda solamente un parametro, sta al server, tramite un *loop*, il compito di eseguirla per ciascun parametro di targa.

```
SELECT P.name, P.UM, V.valueKind, V.valueString,
V.valueFloat, V.valueBoolean FROM(
    SELECT DISTINCT ON (idParameterRegolator) *
    FROM CLOUD.CONFIGURATIONVALUE
    WHERE idBoard = ? AND
    idParameterRegolator= ?
    ORDER BY idParameterRegolator, tmstp desc) as V
INNER JOIN CLOUD.PARAMETERREGOLATOR AS P
ON(V.idParameterRegolator = P.id);
```

Il file di risposta contiene in ogni riga il nome, il valore e l'unità di misura di un parametro.

```
PARAMETER=name VALUE=value UM=um
```

Valori dei parametri di configurazione. La richiesta viene gestita in modo analogo alla precedente. Vengono però prelevati i restanti parametri di configurazione, tralasciando i dati di targa.

Verifica degli Alarm. Per un singolo regolatore, tramite una *query* si recuperano i valori d'esercizio rilevati in presenza di anomalie. Le *join* che vi compaiono servono per ricostruire i dati relativi ai parametri (nome e unità di misura) e ai codici *Alarm* attivi.

```
SELECT PR.name, PR.UM, AV.val, AA.idAlarm, AV.tmstp
FROM CLOUD.ALARMVALUE WHERE idBoard=?) AS AV
INNER JOIN CLOUD.ALARMVALUE_ALARM AS AA
ON (AV.id = AA.idAlarmValue)
INNER JOIN CLOUD.PARAMETERREGOLATOR AS PR
ON(AV.idParameterRegolator = PR.id
ORDER BY tmstp DESC LIMIT 80;
```

Il risultato è stato limitato ad 80 entry. Il server risponde in modo analogo ai precedenti, riportando per ogni riga le informazioni delle letture affette da *Alarm*.

```
PARAMETER=name VALUE=value UM=um ALARM=idAlarm TIMESTAMP=tmsp
```

Se un valore riguarda più *Alarm*, questo comparirà più volte nella risposta, una per ciascun *Alarm*.

Valori puntuali dei dati d'esercizo. L'applicazione Android deve trasmettere il seriale del regolatore e il codice di un parametro di lettura. La richiesta ha lo scopo di ottenere gli ultimi valori d'esercizio nella loro forma puntuale. Vengono quindi prelevati gli ultimi 5 file di *log* dalla rispettiva tabella *Log*.

```
SELECT log.data from CLOUD.AGGREGATEVALUE as AV
INNER JOIN CLOUD.LOG AS log on(AV.idLog = log.id)
WHERE AV.idBoard =? AND AV.idParameterRegolator=?
ORDER BY tmstp DESC LIMIT 5;
```

Da questi, tramite delle operazioni di *parsing*, si ricava l'elenco dei valori. Il server li invia, uno per riga, assieme ad un intero indicante l'ordinamento temporale, dal valore più dato al più recente. Attualmente si trasmettono un massimo di 60 valori. La risposta è così strutturata:

```
0 VALUE#0
1 VALUE#1
...
N-1 VALUE#N-1
N VALUE#N
N+1 VALUE#N+1
...
```

5.2 L'applicazione Android

L'applicazione destinata agli smartphone, per comunicare con il server, fa utilizzo delle richieste descritte precedentemente. In questa sezione viene descritta l'applicazione dal punto di vista del utilizzatore schermata dopo schermata.

È stata data importanza all'interfaccia grafica che compone l'applicazione. Ogni schermata dispone di un menù che permette di risalire alla vista precedente e, tramite alcuni collegamenti, è possibile visitare il sito aziendale o comunicare con il team di supporto. L'interfaccia è stata ottimizzata sia per smartphone che per tablet e l'applicazione è disponibile sia in italiano che in

inglese.

Nella schermata principale dell'applicazione viene proposto un *form* per il *login*. L'utente deve inserire *username* e *password*. Il client invia al server la richiesta di *login* e, se l'autenticazione ha successo, gli viene comunicata la lista dei regolatori. Viene quindi caricata una nuova schermata, che presenta le informazioni ricevute e dalla quale è possibile selezionare il regolatore da analizzare. Tramite un pulsante di ricerca, si può inserire il codice del regolatore per una selezione mirata.

Si accede in questo modo al pannello di controllo del regolatore. Questa schermata presenta 4 pulsanti e le descrizioni delle azioni che si possono intraprendere. È attualmente possibile visionare i dati di targa, di configurazione,

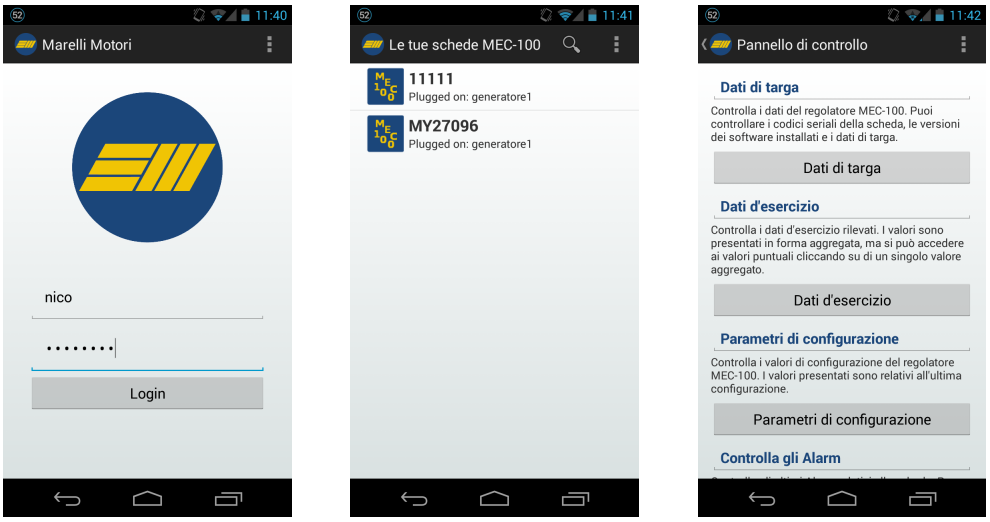

Figura 5.5: Dopo l'autenticazione tramite la schermata principale dell'applicazione, i regolatori vengono mostrati attraverso una lista selezionabile. Alla selezione della scheda viene presentato il pannello di controllo dal quale è possibile consultare i dati del regolatore.

d'esercizio e i vari *Alarm* del regolatore. Ogni pulsante genera una richiesta (con codice 1, 2, 3 o 4) per reperire i dati richiesti. Questi vengono riportati tramite delle liste in cui compare, per ciascun parametro, il nome, il valore e l'unità di misura. L'unica lista cliccabile è quella relativa ai valori aggregati. Selezionando un qualsiasi parametro verrà generata una richiesta con codice 5 per reperirne gli ultimi valori assunti. Viene così caricata una schermata contenente un grafico con le varie variazioni dei valori.

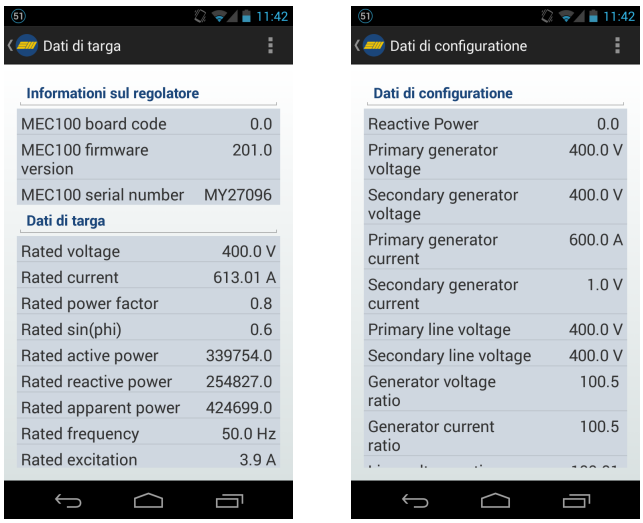

Figura 5.6: La schermata per verificare i dati di targa e i valori di configurazione di un determinato regolatore.

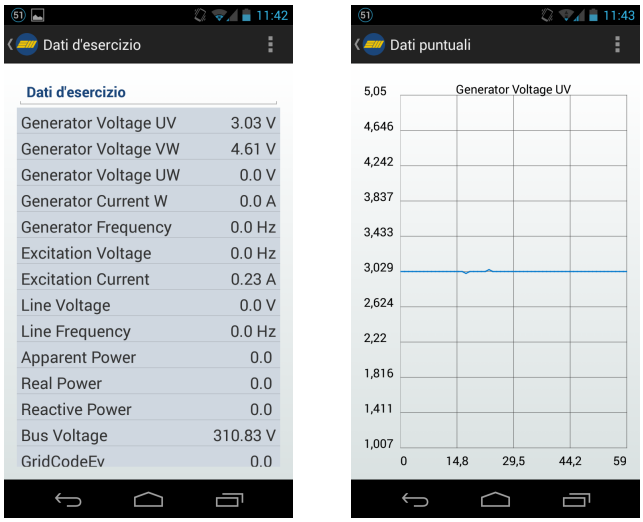

Figura 5.7: I dati d'esercizio sono consultabili nella forma aggregata e, tramite un grafico, nella loro forma puntuale.

Capitolo 6

Conclusioni e Sviluppi futuri

Il sistema presentato è stato implementato e reso funzionante, coprendo molte delle specifiche di progetto. Tuttavia ci sono alcuni aspetti importanti che, sebbene siano stati considerati, non fanno ancora parte del sistema software. In questa sezione sono presentate alcune possibili estensioni riguardanti le problematiche di sicurezza e scalabilità. Per concludere, verranno riportate alcune considerazioni sullo sviluppo di una architettura per le configurazioni remote dei regolatori.

6.1 Sicurezza

La scheda *Hachiko*, durante le comunicazioni con il server centrale, trasmette varie informazioni. Alcune di queste sono estremamente sensibili. Ne sono un esempio l'*username* e la *password* che vengono inviate per l'autenticazione di un tecnico. Questi dati, che al momento vengono trasmessi in chiaro, dovrebbero esser criptati per garantirne la riservatezza.

Il server dispone di una coppia di chiavi *RSA* asimmetriche per la criptazione, ma al momento non sono utilizzate. Per una prima implementazione basilare, si è pensato di trasmettere la chiave pubblica del server in risposta alla richiesta di autenticazione da parte della scheda *Hachiko*. Quest'ultima, nella trasmissione dei dati, li deve criptare tramite la chiave pubblica, mentre il server li può leggere tramite chiave privata.

Questa soluzione presenta però alcune problematiche che ne limitano l'utilizzo. Per prima cosa a venir criptati sono solamente i messaggi che vanno dal client al server, viceversa, i dati di risposta che il server invia al client non vengono criptati. Per permettere una cifratura completa entrambe le entità

dovrebbero possedere una chiave simmetrica condivisa, oppure anche il client dovrebbe fornire al server una propria chiave asimmetrica pubblica. Un'altra problematica è legata alle performance di questa situazione. L'onere computazionale dovuto all'utilizzo di un algoritmo a chiave asimmetrica è maggiore rispetto ad un algoritmo a chiave simmetrica. Le comunicazioni dei file di *log*, le più corpose, sarebbero quelle che più ne risentirebbero di questa implementazione. Infine, l'*header* dei dati non può esser criptato come il resto del messaggio perché non permetterebbe di distinguere la richiesta di autenticazione dalle altre.

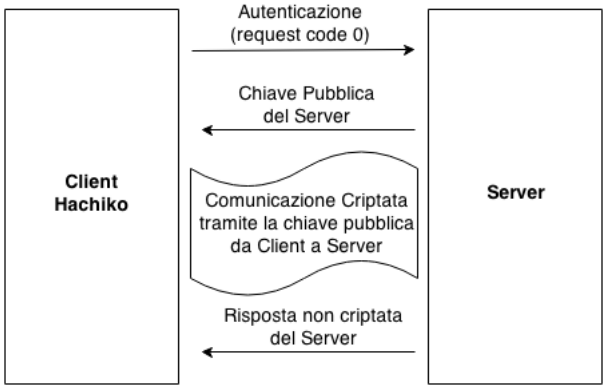

Figura 6.1: Schema per la criptazione tramite chiave pubblica del server. A seguito della richiesta di autenticazione nel sistema il server invia la propria chiave asimetrica pubblica. I messaggi che il client invia al server possono venir criptati, tranne nella parte relativa all'*header*. Le risposte del server sono invece trasmesse in chiaro.

Una possibile soluzione consiste nel adottare il protocollo *SSL*. Questo garantirebbe lo scambio di una chiave condivisa differente per ciascuna comunicazione, tramite l'utilizzo di un algoritmo a chiave asimmetrica. Dopo l'*handshacking* iniziale la comunicazione avviene tramite chiave simmetrica, che garantisce uno scambio di messaggi veloce e criptato. Il client deve possedere il certificato del server prima di iniziare una comunicazione. Questo può venir memorizzato direttamente nella memoria della scheda *Hachiko*.

Il meccanismo deve essere replicabile anche nel caso in cui sia il server a dover iniziare la comunicazione. Benché questa interazione non è al momento utilizzata, sarà indispensabile per permettere le configurazioni da remoto. Le schede *Hachiko* dovrebbero disporre a loro volta di un certificato e di una coppia di chiavi di criptazione.

6.2 Scalabilità e Analisi del carico

È fondamentale garantire la scalabilità del sistema per permettere l'inserimento nel tempo di nuove schede regolatrici. Per come è stato costruito il data base, è possibile aggiungere un nuovo regolatore molto semplicemente. Basta infatti introdurre una nuova *entry* nella relazione *Regolator*, inserire i parametri del regolatore che non sono già contenuti nella tabella *ParameterRegolator* ed associarli tramite la relazione *Regolator_Parameter*. Questo procedimento dev'essere fatto *offline*, prima di collegare la nuova tipologia di regolatore. La scheda *Hachiko* deve invece essere aggiornata, per permettere la comunicazione, magari tramite altre interfacce, con il nuovo regolatore.

Se confrontiamo questa scelta di progetto, rispetto al disporre di un'unica tabella che ha come attributi i parametri di lettura del regolatore, introduciamo della complessità temporale dovuta agli aggiornamenti. Indichiamo con N il

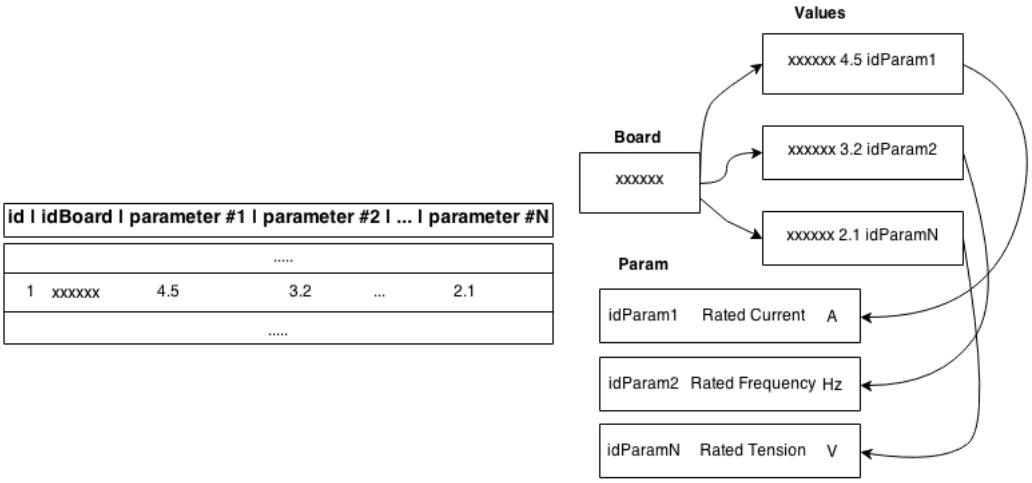

Figura 6.2: In figura sono riportate le due possibili implementazioni per la memorizzazione dei valori di esercizio. A sinistra c'è la rappresentazione che utilizza solamente una tabella i cui campi sono i valori dei parametri. La seconda introduce flessibilità al sistema: permette di aggiungere nuovi regolatori e memorizzare informazioni legate alla natura dei parametri come il nome completo e l'unità di misura. Un inserimento nella prima implementazione corrisponde a molteplici inserimenti nella seconda implementazione.

numero di parametri d'esercizio di un regolatore. Ogni aggiornamento, invece di inserire una tupla con $N + 2$ campi, provoca l'inserimento di N tuple da 4 campi ciascuna[1]. Sono $N + 2$ campi contro $4N$, un singolo inserimento contro

[1]Nel primo caso si devono inserire N valori d'esercizio, la chiave primaria generata in modo incrementale e l'identificativo della scheda. Nel secondo caso il valore d'esercizio, la chiave primaria incrementale, l'id della scheda e l'id del parametro inserito

N del secondo caso.

Per ridurre notevolmente questa problematica, come è già stato illustrato, si è introdotta l'entità *Log* in modo da aggregare i dati e limitare le richieste delle varie schede. Durante l'ordinario funzionamento, una scheda effettua un solo aggiornamento giornaliero. Questo genera un inserimento della seconda forma e inserisce il file binario di *log* nella relativa relazione.

In questo modo si sono ridotte notevolmente le richieste che ciascuna scheda fa al sistema. Il rilevamento dei dati viene fatto ogni secondo, se questi venissero inviati costantemente si avrebbero 86400 aggiornamenti per scheda a giornata. Con il meccanismo introdotto questi vengono inglobati all'interno del file di *log* in un unico aggiornamento.

Il *MEC-100* è dotato di circa 30 valori d'esercizio da monitorare. Rilevando i valori ogni secondo ed inviandoli in blocco giornalmente, i dati da trasmettere diventano molto corposi. Delle semplici accortezze permettono però di tenerli limitati. I dati, prima di essere trasmessi vengono compressi in un file *gzip*, riducendo la quantità in termini di byte dei messaggi. È poi superfluo rilevare i dati ogni secondo, in quanto l'azienda non è interessata ad avere delle letture così accurate. Un rilevamento ogni 10 secondi ridurrebbe la crescita dei dati già di un fattore 10. Infine, i parametri che non variano così frequentemente possono venir inseriti nel file di *log* all'occorrenza.

Se ciascuna scheda rilevasse i parametri ogni 10 secondi, a fine giornata invierebbe un file compresso di circa $4MB$ di informazione. Se l'aggiornamento venisse fatto ogni 6 ore, le trasmissioni avrebbero un peso di $1MB$. Se rilevassimo invece i parametri ogni minuto, i dati da inviare sarebbero nel totale $760KB$.

Considerando l'elevato numero di regolatori che il sistema deve supportare e la quantità di dati che viene prodotta in una giornata, può avere significato distribuire il data base su più macchine. Conviene in questo caso frammentare la base di dati orizzontalmente, destinando un regolatore sempre al solito *DBMS*.

6.3 Configurazione remota dei parametri

Benché attualmente non sia possibile configurare i parametri dei regolatori da remoto, il server fornisce alcune funzionalità di supporto per una futura implementazione. Viene infatti mantenuta una tabella di associazione tra ciascun

regolatore e il suo ultimo ip rilevato, mentre una primitiva permette di inviare dei dati a quest'ultimo.

Ogni configurazione del *MEC-100* deve essere fatta inviando tutti i 120 valori di configurazione al dispositivo. Non tutte le configurazioni potranno essere fatte a distanza: i settaggi più delicati, che possono compromettere il funzionamento del generatore, per ragioni di sicurezza, dovranno essere fatte sul posto. La scheda *Hachiko* dovrà essere dotata di un server *TCP* in modo da essere reperibile dalla rete. Le configurazioni dovranno invece iniziare dall'applicazione Android. Ci sono due possibili strade da intraprendere. Il server potrebbe fare da intermediario tra Android e *Hachiko*. L'applicazione Android comunica al server la richiesta di configurazione delegando quest'ultimo ad interagire con la scheda *Hachiko*. Questa architettura permetterebbe al server di tracciare tutti i settaggi che vengono fatti in modo molto semplice. L'aggiunta di un intermediario introduce però dell'ulteriore ritardo tra l'invio di una richiesta di configurazione ed il settaggio vero e proprio. Anche per le analisi delle risposte o per monitorare i dati d'esercizio a seguito di una configurazione, sarebbe opportuno mantenere il procedimento il più veloce possibile.

In alternativa, l'applicazione Android potrebbe interrogare il server per ottenere l'indirizzo *ip* della scheda *Hachiko* e poi comunicare direttamente con essa. In questo caso, al termine della configurazione, la scheda *Hachiko* dovrà inviare al server la manutenzione fatta. Tramite questa seconda soluzione si avrebbe un controllo più diretto del regolatore. L'applicazione Android dovrebbe però chiedere, prima di iniziare una comunicazione, l'indirizzo ip al server. Questo per ovviare alla problematica per cui la scheda *Hachiko*, in continuo movimento, vari il proprio indirizzo Internet.

Queste architetture non possono sostituirsi ad un sistema *real time*, non offrono cioè alcuna garanzia nei tempi di risposta. Nemmeno la comunicazione tra smartphone e regolatore è in alcun modo garantita. Configurazioni che hanno esigenze *real time* non potranno essere svolte da questo sistema.

6.4 Conclusioni

L'architettura server soddisfa le richieste aziendali permettendo la continua tracciabilità dei dati d'esercizio. Le anomalie rilevate durante il normale funzionamento del motore sono consultabili da remoto evitando al tecnico di dover spostarsi sul luogo per una semplice rilevazione. Non solo, mentre prima gli *Alarm* venivano notificati solamente durante le manutenzioni, ora è possibi-

le rilevarli quando la nave si trova ancora al largo. Se l'applicazione Android notificasse il verificarsi di un *Alarm* tramite un messaggio nella barra delle notifiche, l'anomalia verrebbe comunicata all'utente quasi in tempo reale. Il settaggio dei parametri da remoto ridurrà ulteriormente gli spostamenti dei tecnici sul luogo. Un tecnico ha a disposizione nuovi strumenti che lo aiuteranno nella manutenzione, primo fra tutti lo storico degli andamenti dei parametri d'esercizio. Anche l'elenco delle diverse configurazioni con i dettagli del caso può guidare un manutentore nelle scelte dei parametri di configurazione.

Appendice A

Costruzione del data base

```
-- 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.REGOLATOR(
    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),
    idRegolator     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 (idRegolator) REFERENCES
            CLOUD.REGOLATOR(id)
);

CREATE TABLE CLOUD.HACHIKO(
    id VARCHAR(32),
    PRIMARY KEY (id)
);

CREATE TABLE CLOUD.HACHIKO_BOARD(
    id      SERIAL,
    idHachiko   VARCHAR(32),
    idBoard     VARCHAR(32),
    tmstp       TIMESTAMP NOT NULL
            DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (id),
    FOREIGN KEY (idHachiko) REFERENCES CLOUD.HACHIKO(id),
    FOREIGN KEY (idBoard) REFERENCES CLOUD.BOARD(id)
);
```

```
CREATE TABLE CLOUD.PARAMETERREGOLATOR(
    id  CLOUD.CODE,
    name    VARCHAR(72),
    UM  VARCHAR(10)         DEFAULT '',
    min     CLOUD.FLOATVALUE    DEFAULT 0,
    max     CLOUD.FLOATVALUE    DEFAULT 0,
    PRIMARY KEY(id)
);

CREATE TABLE CLOUD.REGOLATOR_PARAMETER(
    idRegolator         VARCHAR(32),
    idParameterRegolator    CLOUD.CODE,
    PRIMARY KEY(idRegolator, idParameterRegolator),
    FOREIGN KEY(idRegolator) REFERENCES CLOUD.REGOLATOR(id),
    FOREIGN KEY(idParameterRegolator) REFERENCES
        CLOUD.PARAMETERREGOLATOR(id) ON UPDATE CASCADE
);

CREATE TABLE CLOUD.LOG(
    id  SERIAL,
    data    BYTEA,
    PRIMARY KEY(id)
);

CREATE TABLE CLOUD.AGGREGATEVALUE(
    id          SERIAL,
    val             CLOUD.FLOATVALUE NOT NULL,
    idBoard         VARCHAR(32),
    idParameterRegolator    CLOUD.CODE,
    idLog           INTEGER,
    tmstp           TIMESTAMP NOT NULL
                DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (id),
    FOREIGN KEY(idBoard) REFERENCES CLOUD.BOARD(id),
    FOREIGN KEY(idParameterRegolator) REFERENCES
        CLOUD.PARAMETERREGOLATOR(id) ON UPDATE CASCADE,
    FOREIGN KEY(idLog) REFERENCES CLOUD.LOG(id)
);
```

```
CREATE TABLE CLOUD.TYPEGENERATOR(
    id  VARCHAR(32),
    code    VARCHAR(32),
    type    VARCHAR(32),
    PRIMARY KEY(id)
);

CREATE TABLE CLOUD.GENERATOR(
    id          VARCHAR(32),
    idTypeGenerator     VARCHAR(32),
    PRIMARY KEY(id),
    FOREIGN KEY(idTypeGenerator) REFERENCES
        CLOUD.TYPEGENERATOR(id)
);


CREATE TABLE CLOUD.PARAMETERGENERATOR(
    id      CLOUD.CODE,
    name        VARCHAR(50),
    UM      VARCHAR(10)     DEFAULT '',
    min         CLOUD.FLOATVALUE    DEFAULT 0,
    max         CLOUD.FLOATVALUE    DEFAULT 0,
    PRIMARY KEY(id)
);

CREATE TABLE CLOUD.PARAMETERGENERATOR_TYPEGENERATOR(
    idParameterGenerator    CLOUD.CODE,
    idTypeGenerator     VARCHAR(32),
    PRIMARY KEY(idParameterGenerator,idTypeGenerator),
    FOREIGN KEY(idParameterGenerator) REFERENCES
        CLOUD.PARAMETERGENERATOR(id) ON UPDATE CASCADE,
    FOREIGN KEY(idTypeGenerator) REFERENCES
        CLOUD.TYPEGENERATOR(id) ON UPDATE CASCADE
);


CREATE TABLE CLOUD.BOARD_GENERATOR(
    idBoard     VARCHAR(32),
```

```
    idGenerator VARCHAR(32),
    PRIMARY KEY(idBoard, idGenerator),
    FOREIGN KEY(idBoard) REFERENCES CLOUD.BOARD(id),
    FOREIGN KEY(idGenerator) REFERENCES CLOUD.GENERATOR(id)
);

CREATE TABLE CLOUD.VALUEGENERATOR(
    id          SERIAL,
    valueFloat      CLOUD.FLOATVALUE,
    valueString         VARCHAR(32),
    idParameterGenerator    CLOUD.CODE,
    idGenerator         VARCHAR(32),
    FOREIGN KEY(idParameterGenerator) REFERENCES
        CLOUD.PARAMETERGENERATOR(id) ON UPDATE CASCADE,
    FOREIGN KEY(idGenerator) REFERENCES CLOUD.GENERATOR(id),
    PRIMARY KEY(id)
);


CREATE TABLE CLOUD.TECHNICIAN(
    username VARCHAR(32),
    password VARCHAR(32),
    PRIMARY KEY(username)
);

CREATE TABLE CLOUD.PERMISSION(
    usernameTechnician  VARCHAR(32),
    idGenerator         VARCHAR(32),
    FOREIGN KEY(usernameTechnician) REFERENCES
        CLOUD.TECHNICIAN(username) ON UPDATE CASCADE,
    FOREIGN KEY(idGenerator) REFERENCES CLOUD.GENERATOR(id),
    PRIMARY KEY(usernameTechnician, idGenerator)
);


CREATE TABLE CLOUD.ASSISTANCE(
    id          SERIAL,
    description         TEXT,
    document        BYTEA,
```

```
    logFile          BYTEA,
    valueLogFile         BYTEA,
    tmstp            TIMESTAMP NOT NULL
                DEFAULT CURRENT_TIMESTAMP,
    usernameTechnician  VARCHAR(32),
    idBoard          VARCHAR(32),
    PRIMARY KEY(id),
    FOREIGN KEY(usernameTechnician) REFERENCES
        CLOUD.TECHNICIAN(username) ON UPDATE CASCADE,
    FOREIGN KEY(idBoard) REFERENCES CLOUD.BOARD(id)
);



CREATE TABLE CLOUD.CONFIGURATIONVALUE(
    id           SERIAL,
    valueKind        CLOUD.KIND,
    valueFloat       CLOUD.FLOATVALUE,
    valueString          VARCHAR(32),
    valueBoolean         BOOLEAN,
    idBoard          VARCHAR(32),
    idAssistance         INTEGER,
    idParameterRegolator    CLOUD.CODE,
    tmstp            TIMESTAMP NOT NULL
                DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (id),
    FOREIGN KEY(idBoard) REFERENCES CLOUD.BOARD(id),
    FOREIGN KEY(idAssistance) REFERENCES
        CLOUD.ASSISTANCE(id),
    FOREIGN KEY(idParameterRegolator) REFERENCES
        CLOUD.PARAMETERREGOLATOR(id) ON UPDATE CASCADE
);

CREATE TABLE CLOUD.ALARM(
    id      CLOUD.CODE,
    name         VARCHAR(50),
    description      VARCHAR(50),
    PRIMARY KEY (id)
);
```

```
CREATE TABLE CLOUD.ALARMVALUE(
    id          SERIAL,
    val             CLOUD.FLOATVALUE,
    tmstp           TIMESTAMP NOT NULL
                DEFAULT CURRENT_TIMESTAMP,
    idBoard         VARCHAR(32),
    idParameterRegolator    CLOUD.CODE,
    PRIMARY KEY(id),
    FOREIGN KEY(idBoard) REFERENCES CLOUD.BOARD(id),
    FOREIGN KEY(idParameterRegolator) REFERENCES
        CLOUD.PARAMETERREGOLATOR(id)
);

CREATE TABLE CLOUD.ALARMVALUE_ALARM(
    idAlarmValue    INTEGER,
    idAlarm     CLOUD.CODE,
    PRIMARY KEY (idAlarmValue, idAlarm),
    FOREIGN KEY(idAlarmValue) REFERENCES
        CLOUD.ALARMVALUE(id),
    FOREIGN KEY(idAlarm) REFERENCES CLOUD.ALARM(id)
);

CREATE TABLE CLOUD.REGOLATORALARM(
    idRegolator     VARCHAR(32),
    idAlarm     CLOUD.CODE,
    FOREIGN KEY(idRegolator) REFERENCES CLOUD.REGOLATOR(id),
    FOREIGN KEY(idAlarm) REFERENCES CLOUD.ALARM(id)
);
```