Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
Corso di Laurea Triennale in Ingegneria Informatica

Elaborato di Laurea Triennale

SMART GRID CON COMUNICAZIONI SU LINEA ELETTRICA: SIMULAZIONE CON HARDWARE IN THE LOOP

Candidato:
Simone Spangaro

Relatore:
Prof. Stefano Tomasin

Anno Accademico 2012-2013

Indice

Capitolo 1

Introduzione

Nei tempi recenti, la gestione delle reti elettriche di media e bassa tensione ha subito un importante cambiamento nella direzione di un utilizzo maggiormente efficiente dell'energia, consentendo la redistribuzione dell'energia stessa in aree mirate della rete e cercando di far fronte al sempre più crescente utilizzo di energie rinnovabili. Questi obiettivi trovano realizzazione nel concetto di Smart Grid, ossia una rete di informazione affiancata ad una rete elettrica che consente di gestire in maniera più intelligente la rete stessa. Un ulteriore ed importante tassello per la creazione di queste reti intelligenti, è la parte di comunicazione ed interazione tra i dispositivi presenti nella rete, per la quale la tecnologia Power Line Communications (PLC) costituisce un fondamentale elemento.

In questo elaborato, si cercherà di studiare le reti Smart Grid tramite la creazione di un ambiente software che consenta di simulare il funzionamento di una rete elettrica vera e propria, con la presenza di dispositivi di rete che interagendo con i modem PLC, che costituiscono la parte intelligente della rete, permetta di controllare l'evoluzione della rete stessa, modificandone opportuni parametri al fine di eseguire le necessarie compensazioni energetiche.

Il capitolo 2 fornisce una breve panoramica sulle reti Smart Grids; in esso saranno presentati alcuni progetti sviluppati al Dipartimento di Ingegneria dell'Informazione dell'Università di Padova, che hanno fornito un importante base di partenza per la realizzazione di questo lavoro. Nei capitoli 3 e 4, verranno descritti invece i modelli software con cui si sono simulati alcuni degli strumenti di rete ed inoltre come è stato possibile realizzare e gestire una rete elettrica virtuale. Nel capitolo 5, vengono presentati i risultati osservati nella fase di test di una rete elettrica virtuale con schema fissato, dove la parte software che emula i dispositivi reali comunica con i modem PLC al fine di produrre uno scambio di informazioni per realizzare le compensazioni ener-

getiche previste. Infine nel capitolo 6 verrà descritta una ulteriore estensione del simulatore di rete, comprendente la realizzazione software dei modem che utilizzano la tecnologia PLC, al fine di ampliare le possibilità offerte dallo stesso simulatore.

Capitolo 2

Smart Grids

Nell'ambito delle reti elettriche negli ultimi anni si è assistito ad un tentativo di rendere maggiormente efficiente la distribuzione di energia elettrica cercando di evitare sprechi, sovraccarichi o cadute di tensione. Questo suggerisce l'idea di una gestione intelligente della rete elettrica stessa, scostandosi dalla precedente visione di una rete dove le grandi centrali di produzione distribuiscono l'energia alle utenze con uno schema rigido, evolvendosi in modo da essere in grado non solo di gestire in tempo reale picchi di energia, redistribuendola in altre aree a seconda delle necessità ma anche che riesca a far fronte a situazioni in cui la produzione di energia risulta in alcuni punti inferiore a quanto necessario o a guasti che possono presentarsi all'interno della rete.

Tutto ciò è vero principalmente per le reti di distribuzione in medio/bassa tensione dove sempre più spesso, oltre alle centrali di energia elettrica sono presenti sistemi di produzione di energia che sfruttano le energie rinnovabili, come fotovoltaico ed eolico, caratterizzati da un livello di produzione non sempre costante e prevedibile. Collegato a quanto detto è il concetto di Smart Grid, ossia una rete elettrica affiancata da una rete di informazione che consente di farla operare in modo intelligente sotto vari aspetti. Si vuole quindi creare un' infrastruttura nella quale, lo scambio di dati tra grandi centrali elettriche e altre strutture con livelli di produzione più ridotti, come ad esempio pannelli fotovoltaici installati sul tetto di edifici, consenta di regolare nel modo più opportuno ed efficiente la distribuzione di energia.

Una tecnologia importante in questo senso è Power Line Communication (PLC) che sfutta la rete di alimentazione elettrica, o per meglio dire i cavi utilizzati per trasportare l'energia stessa, per la trasmissione di dati. Questo è possibile grazie alla sovrapposizione di un segnale a frequenza più elevata rispetto a quello relativo all'informazione della corrente elettrica (che in Europa è di 50 Hz), ricorrendo in seguito al filtraggio per recuperare i dati

situati in bande di frequenza differenti.

Al Dipartimento di Ingegneria dell'Informazione (DEI) dell'Università di Padova, è presente da alcuni anni un progetto dedicato agli studi e alle applicazioni sulle Smart Grid; inoltre, al fine di realizzare sperimentazioni su tali reti, sono presenti strumenti di misura, componenti elettrici e altre apparecchiature che utilizzano la tecnologia PLC. Tra gli argomenti di interesse, particolare attenzione si è posta sulle possibiltà di controllo di una Smart Grid tramite dispositivi che sfruttano la tecnologia PLC: in uno dei lavori [1] sviluppati nei laboratori del DEI, è stato proposto e testato un algoritmo di sincronizzazione per lo scambio di informazioni tra elementi che compongono la Smart Grid funzionante tramite comunicazione power line, basato sull'algoritmo Network Time Protocol (NTP) e il calcolo della stima dell'offset di frequenza. L'obiettivo era quello di verificare le principali problematiche relative alla sincronizzazione (realizzata con timestamp) tra nodi della rete per lo scambio di dati, cercando di analizzare il limiti temporali della comunicazione che avviene su power line. I risultati hanno mostrato buone performance per la tecnologia PLC, relativamente alla strumentazione utilizzata, ottenendo livelli di precisione nell'ordine dei $10\mu s$, più che adeguati per la gestione delle Smart Grids.

In un secondo lavoro [2], è stata creata una piattaforma di test nei laboratori del DEI il cui scopo era quello di ridurre la richiesta di corrente elettrica necessaria alla rete di distribuzione bilanciando la potenza reattiva scambiata all'interno della rete: questo approccio garantisce una maggiore efficienza nella distribuzione dell'energia stessa e consente di ridurre le perdite. Per ottenere ciò è stata realizzata un'infrastruttura di controllo e comunicazione che, servendosi di modem che utilizzano la tecnologia PLC, permette di gestire le informazioni ricavate dalla rete in cui opera e comandare di conseguenza gli altri strumenti in modo che questi intraprendano le necessarie azioni per il raggiungimento del suddetto scopo.

Analizzeremo di seguito alcuni dettagli di quest'ultimo progetto, fornendo innanzitutto una breve descrizione della strumentazione utilizzata in laboratorio, che comprende i seguenti dispositivi:

1. IME Nemo D4-L+: è un monitor di rete multifunzione per reti a bassa/media tensione. Consente di misurare tensione e corrente di fase, potenza attiva e reattiva sia per reti monofase che trifase. I valori misurati sono i valori efficaci delle grandezze elettriche o Root Mean Square (RMS) e sono visualizzati tramite un display LCD. Lo strumento può scambiare informazioni con altri dispositivi tramite una porta seriale RS-485, utilizzando un protocollo di comunicazione proprietario;

2. Chroma 63804 DC/AC Load: è un carico programmabile che può essere impiegato sia per come strumento di test e diagnostica di numerose apparecchiature elettriche sia per effettuare simulazioni di differenti tipologie di carico, sia in corrente continua che alternata. I carichi possono essere simulati impostando i valori del crest factor (rapporto tra corrente di picco e valore (RMS) della corrente) in un range 1.414 - 5.0 e i valori del power factor (rapporto tra potenza attiva e potenza apparente) compresi tra 0 e 1. Lo strumento dispone di due interfacce, General Purpose Interface Bus (GPIB) e RS-232, quest'ultima è associabile alla modalità di funzionamento remota, grazie alla quale un dispositivo esterno può controllare in remoto il carico programmabile inviandogli comandi che adottano il protocollo Standard Command for Programmable Instruments (SCPI.) E' possibile connettere al Chroma 63804 un oscilloscopio per visualizzare le forme d'onda di tensioni e correnti dello strumento;

3. Texas Instrument Power Line Modem Developer's Kit (TMDSPLCKIT-V2): si tratta di modem PLC, basati sull'architettura C2000 di Texas Instrument Technology (TI), impiegati per realizzare la struttura di comando e controllo della Smart Grid. I modem sono dotati di una porta seriale RS-232, che consente di effettuare operazioni di diagnostica dello strumento o la comunicazione con altri dispositivi ed un connettore power grid che, sfruttando la tecnologia PLC, permette loro di scambiare informazioni con altri elementi compatibili, come ad esempio i modem PLC di TI. Il modem è dotato di una Read Only Memory (ROM) interna con firmware aggiornabile (via Universal Serial Bus (USB)), che può essere programmato in un ambiente sviluppato dalla stessa Texas Instrument, chiamato Code Composer Studio basato sul ben noto ambiente di sviluppo software open source Eclipse;

4. un oscilloscopio Agilent MSOX2024A, impiegato per la visualizzazione di forme d'onda della tensione e della corrente e i valori della potenza scambiata dal carico programmabile.

La rete Smart Grid per la realizzazione dei test era composta secondo lo schema riportato in Figura 2.1. Sono presenti:

1. due modem PLC, in configurazione master-slave: il modem master comanda il carico programmabile mentre il modem slave è connesso al Nemo che misura le grandezze elettriche della rete di carico (contrassegnata come LOAD nello schema);

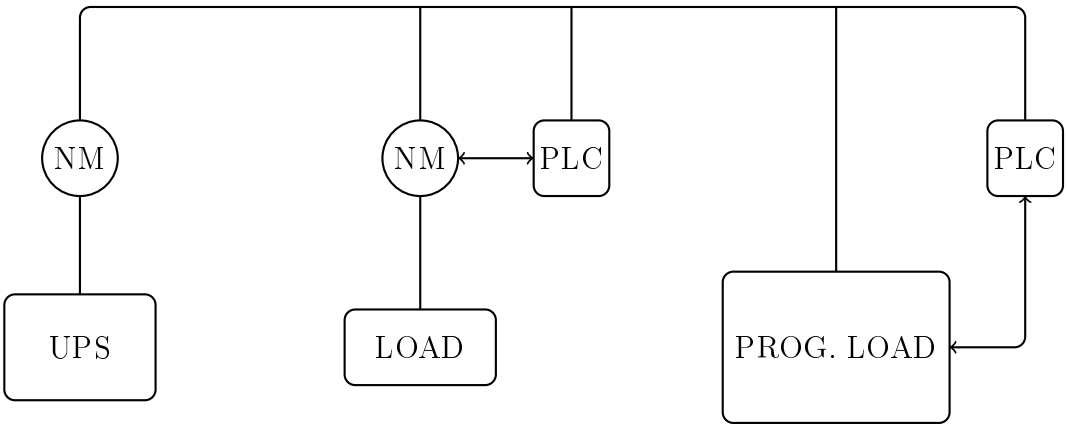

Figura 2.1: Esempio di Smart Grid

2. due monitor di rete NEMO D4-L+, uno connesso ad un Uninterruptible Power Supply (UPS) ed un altro ad una semplice rete di carico, quest'ultimo comunica via porta seriale con il modem PLC slave;

3. un carico programmabile Chroma Load 63804, connesso al modem PLC master tramite porta seriale;

4. una sottorete di carico i cui valori di tensione, corrente sono misurati da uno dei due Nemo D4-L+, il valore dell'impedenza complessiva della rete viene modificato al fine di registrare variazioni nella potenza reattiva dell'intera rete.

Lo scopo del progetto [2] era quello di cercare di ridurre la potenza reattiva scambiata dalla rete (il carico) con l'UPS adattando il comportamento del carico adattativo, che simula il possibile comportamento di una fonte di energia locale controllata opportunamente. Quando la compensazione della potenza reattiva da parte del carico adattativo è perfetta, il Nemo D4-L+ collegato all'UPS misura una potenza reattiva scambiata nulla.

L'intelligenza della rete è stata creata tramite l'utilizzo dei due modem PLC: il modem master è colui che detiene il controllo della rete, comandando al carico programmabile (che opera in modalità remota) le azioni da intraprendere e i valori della potenza attiva, del power factor e del crest factor da impostare tramite la porta seriale; il modem slave funge invece da informatore per il modem master, comunicandogli i valori di tensione, corrente, potenza attiva e potenza reattiva (che costituiscono lo *status vector*) che ha ottenuto interrogando il dispositivo Nemo D4-L+ a lui connesso sempre tramite porta seriale.

TIPO	IMPEDENZA	POTENZA COMPLESSA
R//C	Z = 188 - j318	S = 281 - j166
C	Z = -j1592	S = 0 - j33
L	Z = +j370	S = 0 + j143

Tabella 2.1: Impedenze di carico

Per la sottorete di carico sono stati utilizzati componenti elettronici con valori dell' impedenza equivalente riportati in Tabella 2.1.

Al momento dell'esecuzione dei test con la rete qui sopra descritta, lo strumento di misura Nemo D4-L+ ha mostrato un comportamento non sempre in linea con quanto atteso: in particolare, il valore della potenza reattiva non veniva rilevato correttamente quando la sottorete di carico era costituita da un'impedenza non puramente reale. Per ovviare a questo inconveniente, si è agito sui valori del power factor e del crest factor del carico programmabile modificandoli fino a quando le quantità misurate dal Nemo non fossero coerenti con l'impedenza utilizzata sulla rete. I valori risultanti sono rispettivamente di 0.95 per il power factor e 1.55 per il crest factor, questa coppia è impiegata ogniqualvolta la sottorete di carico comprenda anche elementi reattivi al suo interno.

Analizziamo ora il funzionamento tipico di questa rete Smart Grid:

- inizialmente, il master modem comanda al carico programmabile di simulare un carico completamente resistivo, quindi di impostare un valore per la potenza attiva (P) pari a 100 W, un power factor (PF) pari a 1.00 ed un crest factor (CF) di 1.414. Questi valori rappresenteranno la situazione di default per il carico programmabile;

- il master modem interroga periodicamente il modem slave per ottenere lo status vector, ossia le informazioni relative alla tensione, la corrente, la potenza attiva e reattiva rilevate nella sottorete di carico;

- quando il master modem riceve un valore della potenza reattiva non nullo, comanda al carico programmabile di impostare un PF di 0.95 e un CF di 1.55. Il carico programmabile non riceve però dal modem la potenza reattiva da compensare, ma solamente il valore della potenza attiva che deve impostare al fine di ottenere la compensazione della potenza reattiva, in accordo con le quantità indicate per PF e CF. Il valore di tale potenza attiva P si può ricavare tramite la seguente formula:

$$P = PF \frac{Q}{\sqrt{1 - PF^2}}$$

Dato che la potenza reattiva può essere sia positiva che negativa, il suo segno viene impostato all'interno del carico programmabile tramite quello del power factor. Teoricamente, per componenti passivi il fattore di potenza è per definizione non negativo, ma i costruttori del carico adattativo hanno previsto, per consentire la simulazione di varie tipologie di carichi, di utilizzare valori per PF sia positivi che negativi; pertanto nel caso in esame, per gestire ed ottenere la compensazione della potenza reattiva i valori per il PF sono $\pm$ 0.95 in base all'impedenza della sottorete di carico.

Durante lo svolgimento dei test relativi al progetto [2] sono emerse alcune problematiche e limitazioni imputabili principalmente alla strumentazione utilizzata in laboratorio:

- innanzitutto, il carico programmabile non è in grado di operare come un componente completamente reattivo, come desiderato, ma può impostare un valore minimo del power factor pari a 0.1 che non è ovviamente 0. Inoltre, gli intervalli di valori selezionabili per il power factor e il crest factor sono limitati dalle caratteristiche dello strumento stesso;

- il monitor di rete, il Nemo D4-L+, ha mostrato alcune difficoltà nella rilevazione delle potenze reattive. In particolare, come sopra riportato, quando nella rete venivano inseriti componenti reattivi, visualizzava informazioni relative alle grandezze elettriche della rete che si discostavano anche in maniera significativa dai valori attesi: è stato necessario intervenire sui valori del power factor e del crest factor del carico programmabile per arginare questo inconveniente.

Capitolo 3

Realizzazione dei dispositivi virtuali

La rete Smart Grid descritta nel precedente capitolo, sfrutta la tecnologia Power Line Communications per consentire a due modem in configurazione master/slave di scambiare tra loro informazioni dettagliate sulle tensioni, correnti e potenze rilevate dai monitor di rete e a partire da questi dati, provvedere alla compensazione della potenza reattiva nella rete. Ciò è ottenuto impostando, tramite opportuni comandi, i valori di potenza attiva, power factor e crest factor di un carico programmabile controllato in remoto via porta seriale. Tuttavia l'approccio seguito presentava due limiti principali:

1. non erano disponibili altri carichi adattativi o generatori locali per studiare reti più complesse;
2. i Nemo ed il carico adattativo presentavano dei limiti di prestazione, come descritto nel Capitolo 2.

La soluzione che si è scelto di adottare per risolvere i problemi appena elencati è stata quella di emulare la rete elettrica ed il comportamento dei misuratori e del carico adattativo mediante il PC. L'idea consiste quindi nel realizzare delle versioni virtuali dei dispositivi reali, replicandone in gran parte lo schema di funzionamento, con la necessità di gestire le comunicazioni con i due modem PLC, tramite porta seriale proprio come avviene nelle loro controparti reali. Essendo il software eseguito in un elaboratore, questo dovrà disporre di un adeguato numero di porte seriali per le connessioni necessarie. Si viene così a costituire una simulazione di una Smart Grid con hardware in the loop, ovvero una simulazione in cui alcuni componenti hardware (i modem nel nostro caso) sono integrati nella piattaforma di simulazione.

Più precisamente, i software simuleranno solamente un sottoinsieme delle funzionalità previste dagli strumenti reali; sono state implementate quelle

che consentono una corretta gestione della porta seriale, la rilevazione delle grandezze elettriche della rete e, nel caso del carico programmabile, della simulazione di un carico con valori di power factor e crest factor preimpostati.

Risulta molto importante sottolineare, che anche per la rete elettrica si dovrà provvedere a realizzarne una versione virtuale, accompagnata inoltre da un ulteriore software che permetta di gestirla, modificando in base alle necessità i valori di tensioni, correnti e impedenze della rete stessa. L'insieme dei vari software verrà eseguito su una macchina linux, dotata come detto di un numero sufficiente di porte seriali.

La struttura risultante è quindi costituita da un ibrido tra componenti reali e virtuali in quanto:

- il monitor di rete, il Nemo D4-L+ viene sostituito da una sua emulazione virtuale che è in grado di ricevere ed inviare dati tramite una porta seriale designata della macchina su cui il software è eseguito; i valori di tensione, corrente, potenza attiva e reattiva che il dispositivo virtuale misura sono quelli di una impedenza della rete elettrica virtuale;

- il carico programmabile, il Chroma 63804 viene sostituito da una sua emulazione virtuale che riceve, da una porta seriale del computer, i comandi dal modem PLC master dove sono specificati i valori da impostare per la potenza attiva, il power factor e il crest factor. Queste tre quantità sono utilizzate per simulare un carico all'interno della rete elettrica virtuale, i cui valori sono aggiornati ogniqualvolta è richiesta una compensazione della potenza reattiva;

- la rete elettrica, sostituita da una sua rappresentazione virtuale, costruita secondo uno schema elettrico prescelto;

- i modem PLC, che non verrano invece emulati e rimangono gli unici elementi reali della struttura di test originale presente in laboratorio, fatta eccezione per l'insieme di cavi seriali ed elettrici necessari per alimentare e far comunicare i dispositivi;

- una macchina linux, su cui viene fatto girare il software creato per emulare gli strumenti reali e la rete elettrica.

Quanto appena elencato può essere riassunto dallo schema presente in Figura 3.1. Si noti che questa simulazione di rete con hardware in the loop non permette di valutare l'impatto dei dispositivi elettrici sulla comunicazione PLC, in quanto i modem comunicano tra loro mediante una rete elettrica che non è quella simulata. I dispositivi virtuali che comunicano con i modem PLC saranno associati ciascuno ad una diversa porta seriale dell'elaboratore

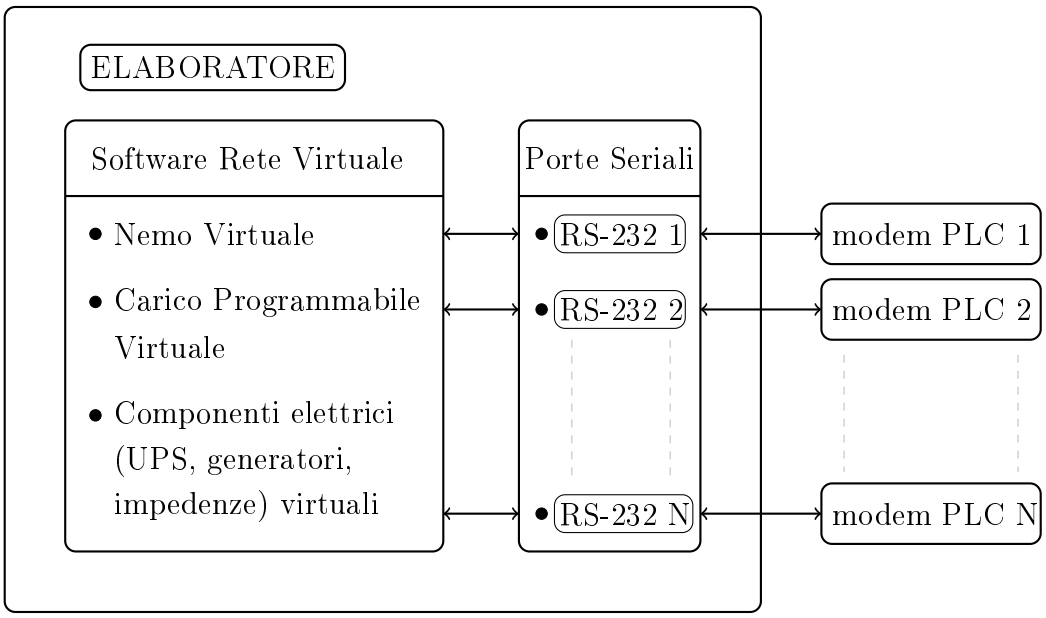

Figura 3.1: Struttura del sistema di test con dispositivi virtuali e modem PLC

su cui è in esecuzione il software di rete. E' importante ricordare che con il termine dispositivo virtuale si intende indicare le emulazioni software degli strumenti di rete Nemo D4-L+ e Chroma 63804. L'ambiente di lavoro su cui tali dispositivi virtuali sono stati creati ed utilizzati è rappresentato da una distribuzione linux, Ubuntu 12.04 LTS, installato su una macchina Dell Optiplex 790. Il linguaggio di programmazione scelto per la scrittura e lo sviluppo delle applicazioni è il C: questa decisione è legata al fatto che, per il sistema operativo Ubuntu sono disponibili librerie ben documentante e standardizzate per la gestione della porta seriale, necessaria per le comunicazioni con i modem PLC, ed un insieme di altri costrutti che verranno presentati in seguito, che hanno consentito la creazione di molteplici dispositivi virtuali nella rete.

Nello schema sovrastante (Figura 3.1), si può notare come all'interno del blocco denominato "Software Rete Virtuale" siano annotate le componenti fondamentali che lo costituiscono. Complessivamente, la struttura software è formata dai seguenti blocchi fondamentali:

- un codice C per simulare il funzionamento del Nemo D4-L+, usato per le rilevazioni di tensioni, correnti e potenze di un componente elettrico (virtuale);

- un codice C per emulare le caratteristiche e le funzioni del carico programmabile Chroma 63804;
- un codice C comprendente un insieme di funzioni per leggere/scrivere dati dalla porta seriale;
- un codice C per la creazione e la gestione di un rete elettrica composta da generatori e impedenze virtuali.

Si è detto in precedenza che uno dei fattori limitanti al fine di creare reti Smart Grid di maggiore complessità è legato al numero di strumenti disponibili in laboratorio. Il primo passo in avanti per risolvere questa situazione è stato compiuto con la scelta di realizzare virtualmente, all'interno di un elaboratore i componenti reali della rete. Il secondo passo, consiste invece nel capire come rendere possibile l'esecuzione di uno stesso codice, che costituisce un singolo dispositivo virtuale, in più istanze al fine di ottenere il numero voluto di dispositivi virtuali, qualsiasi esso sia. Ovviamente, questi dispositivi condividono sia il medesimo codice, cioè si comportano allo stesso modo ed eseguono quindi le medesime funzioni, sia devono necessariamente operare su insiemi di dati e variabili differenti.

3.1 Gestione del multithreading

Quanto detto finora è legato al concetto di multithreading, cioè la possibilità di far eseguire al processore centrale di un elaboratore più threads concorrenti (che competono cioè tra loro al fine di impadronirsi del controllo del processore e proseguire nella loro evoluzione). Un thread è un flusso esecutivo (di istruzioni) a cui è associato un codice, dati statici e dati dinamici. E' quindi possibile, una volta scritto il codice (in un certo linguaggio di programmazione) che descrive le azioni compiute da un determinato thread, creare più thread che condivideranno, in base a quanto detto, lo stesso codice (ossia si comporteranno allo stesso modo) ma saranno dotati di un proprio stack per i dati dinamici (variabili e valori privati utilizzati solamente da quello specifico thread). All'interno dell'elaboratore e della rete elettrica virtuale, gli strumenti di rete saranno rappresentati quindi come dei threads, ed opereranno simultaneamente e in concorrenza tra di loro. Vediamo allora nel dettaglio come sarà costituito il software:

- per ogni monitor di rete (Nemo D4-L+ virtuale) che si vuole inserire all'interno della rete, viene creato un apposito thread che esegue il codice C scritto per simulare le funzioni svolte dallo strumento reale. Come detto in precedenza, questo codice è lo stesso per tutti i threads

creati. Ogni Nemo virtuale viene associato ad un elemento della rete elettrica virtuale (può trattarsi ad esempio, dell'UPS o di un'impedenza). Si può inoltre decidere di associare ad un Nemo virtuale (e quindi al thread che lo rappresenta) anche una porta seriale dell'elaboratore: questa eventualità è prevista nel caso in cui il Nemo virtuale scambi informazioni con un modem PLC;

- per ogni carico programmabile (Chroma 63804 virtuale) che si intende utilizzare, viene creato un thread apposito che esegue le funzioni inserite nel codice C che emula il carico programmabile reale. Come accade per il Nemo virtuale, ogni carico programmabile virtuale può essere associato ad una porta seriale se è previsto lo scambio di dati con un modem PLC, ricevendo in questo modo i comandi contenenti i valori da impostare circa la potenza attiva, il power factor ed il crest factor;

- viene inoltre creato un ulteriore thread per la gestione dell'intera rete elettrica virtuale, che provvede ad aggiornare i valori delle tensioni, correnti e potenze per ciascun elemento della rete. Questo thread non sarà eseguito in più istanze e non necessiterà di comunicare con alcun modem.

Si rende ora necessaria una breve riflessione sulla scelta di utilizzare il multithreading all'interno della piattaforma software finora descritta: solitamente il multithreading è impiegato per cercare di rendere più efficiente e veloce l'esecuzione di un determinato programma. Spesso inoltre, questo comporta la suddivisione del programma principale in differenti sottoprogrammi, che vengono associati a threads separati in modo da migliorare i tempi con cui il processore porta a termine il lavoro designato.

L'utilizzo massiccio del multithreading può in realtà rivelarsi un punto debole quando un numero elevato di threads viene creato, sia perché come detto questi condividono un numero, seppur limitato, di risorse sia perché può rendersi necessario l'utilizzo di meccanismi di sincronizzazione per gestire l'interferenza tra threads quando questi operano su dati comuni.

Analizzando la situazione della rete virtuale creata per i test sulle Smart Grids, si evince che l'utilizzo del multithreading comporta un buon numero di vantaggi, e nessun particolare svantaggio. Infatti:

- il multithreading rende agevole la creazione e l'utilizzo di più dispositivi virtuali che operano all'interno della rete virtuale;

- le funzioni svolte da ciascun dispositivo virtuale sono piuttosto semplici e di modesta complessità. Esse si limitano alla lettura o alla modifica di

alcune variabili (si pensi alla lettura di una tensione di un Nemo virtuale o al salvataggio della potenza attiva ricevuta per il carico programmabile virtuale). Le uniche operazioni che richiedono invece un controllo più rigido sono quelle relative alla comunicazione tramite porta seriale in quanto in questo caso, è necessario fornire messaggi di risposta ai modem PLC ed impostare le azioni specifiche che essi provvedono a comandare;

- le uniche possibili interferenze che possono manifestarsi all'interno della rete riguardano le operazioni svolte dai dispositivi virtuali e dal thread gestore della rete: è possibile infatti che quest'ultimo possa voler eseguire operazioni di aggiornamento dei valori dei componenti elettrici della rete virtuale (ad esempio la modifica di una tensione o la corrente di un certo lato della rete) nello stesso istante in cui un dispositivo virtuale voglia leggere i valori di tali variabili. Questa situazione potrebbe generare informazioni inconsistenti (ad esempio un Nemo virtuale potrebbe leggere il valore di una tensione o di una corrente riferito ad un valore "non aggiornato" di un'impedenza che è stata modificata di recente) e generare quindi compensazioni della potenza errate all'interno della rete;

- non sono richiesti livelli di prestazioni particolarmente elevati o critici per la rete elettrica virtuale, quindi anche la presenza di un numero relativamente elevato di threads concorrenti non impedirebbe il corretto funzionamento della rete stessa.

Per consentire la realizzazione di un programma multithread scritto in codice C, è necessario l'impiego di una libreria che fornisca alcune funzioni per creare e gestire i threads. POSIX thread indica un API (Application Program Interface) standard POSIX (ANSI/IEEE POSIX 1003.1c) [5] che consente di operare con i threads ed include anche un insieme di costrutti di sincronizzazione per gestire la concorrenza degli stessi. Viene ora presentato un elenco di alcune funzioni proprie delle librerie "pthread" e "semaphore" che sono state utilizzate al momento della scrittura del codice C, gli header delle librerie (rispettivamente "pthread.h" e "semaphore.h") vanno inclusi nel codice C (con il comando "*#include*"):

- **int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(start_routine)(void*), void *arg)** è la funzione che permette la creazione di un nuovo thread:

  - *pthread_t *thread* costituisce l'identificatore del thread, se questo viene creato correttamente;

  - *const pthread_attr_t *attr* sono invece gli attributi del processo da creare (NULL per specificare l'utilizzo di valori di default);
  - *void *(*start_routine)(void *)* funzione che verrà eseguita al momento della creazione del thread;
  - *void *arg* costituisce l'argomento da passare al thread;
  - **pthread_create** restituisce un intero (*int*) come valore di ritorno, che è zero (0) in caso la creazione del thread vada a buon fine altrimenti il valore sarà diverso da zero;

- **int pthread_join(pthread_t *thread, void **value)** è una forma molto semplice di sincronizzazione, consente ad un thread di attendere la terminazione di un altro thread specificato:
  - *pthread_t *thread*: identificatore del thread di cui si voule attendere la terminazione;
  - *void **value*: è il valore restituito dal thread che termina la sua esecuzione;
  - il valore di ritorno, di tipo *integer* (*int*) è zero se la funzione viene correttamente eseguita;

- i semafori sono uno strumento generale per sincronizzare i threads, ed in particolare per gestire l'interferenza quando i threads operano su un insieme di variabili comuni. Un semaforo è associato ad valore intero, impostabile dal programmatore che consente o nega l'accesso ad una certa sezione di codice, e a due operazioni fondamentali, *wait* e *post* (o *signal*) che permettono rispettivamente di verificare la disponibilità di un semaforo ed accedervi se esso ha un valore maggiore di zero (0), e di incrementare il valore del semaforo al termine del suo utilizzo. Un semaforo è descritto dalla variabile di tipo *sem_t* all'interno della libreria pthread;

- **int sem_init(sem_t *sem, int pshared, unsigned int value)**:
  - *sem_t *sem* è il puntatore al semaforo che deve essere inizializzato;
  - *int pshared* va posto a 1 se il semaforo è condiviso tra i processi, 0 se invece è privato ad uno specifico thread (0 è il valore da utilizzare nell'implementazione POSIX);
  - *unsigned int value* costituisce il valore che si vuole assegnare al semaforo;

  - il valore di ritorno è 0 se l'inizializzazione viene correttamente eseguita, -1 in caso contrario;

- **int sem_wait(sem_t *sem)** è una chiamata bloccante che consente di verificare la disponibilità di un determinato semaforo (il cui puntatore è costituito dalla variabile *sem_t *sem*) e nel caso il valore del semaforo sia maggiore di zero, questo viene decrementato di una unità;

- **int sem_post(sem_t *sem)** consente di incrementare il valore del semaforo puntato dalla variabile *sem_t *sem*;

- *sem_wait* e *sem_post* restituiscono un valore *integer* che è pari a 0 se l'operazione è stata corretamente eseguita, -1 in caso di errore;

- **int sem_getvalue(sem_t *sem, int *sval)** permette di leggere il valore (*int *sval*) di un semaforo (puntatore *sem_t *sem*). Viene sempre restituito 0 come valore di ritorno.

3.2 Gestione della porta seriale

Come detto più volte in precedenza, la comunicazione tra modem PLC e i dispositivi Nemo D4-L+ e Chroma 63804 avviene tramite una porta seriale RS-232 (nel caso del Nemo D4-L+ è anche presente un convertitore RS-485 -> RS-232). Questo ha richiesto, al momento della scrittura del codice C che realizza le versioni virtuali dei suddetti dispositivi, la necessità di poter operare con delle funzioni che consentano di gestire la porta seriale sia per leggere/scrivere informazioni dalla/nella porta stessa che per impostare alcuni fondamentali parametri di comunicazione quali velocità, bit di parità ecc... Per fare ciò, si è scelto di utilizzare l'API UNIX termios, che permette di lavorare con le porte RS-232 di un elaboratore, con le più comuni operazioni di I/O .

Quando, durante la scrittura di un programma su piattaforma unix, si vuole inserire una sezione di codice in cui sono previste operazioni di I/O, per fare ciò si deve leggere e/o scrivere un file descriptor. Un file descriptor è un valore di tipo *integer* (*int*) associato ad un file aperto nel sistema. Il file però può rappresentare una connessione di rete, un terminale, un vero e proprio file salvato nella memoria secondaria dell'elaboratore o molte altre possibilità. Per ottenere il file descriptor, è necessario ricorrere ad opportune chiamate a sistema (system call) in base alla tipologia di I/O che si vuole utilizzare. Tali system call, restituiscono il file descriptor come valore *int*

di ritorno nel caso la funzione non abbia riscontrato errori durante la sua esecuzione. Ciascuna porta seriale quindi in un sistema Linux ha uno o più *device files* (ossia files nella directory "\dev") ad esso associata: per esempio la porta seriale S0 ha associato il file "\dev \ttyS0". In pratica, una porta seriale può essere pensata come un vero e proprio file.

Per la realizzazione del software per per la gestione della porta RS-232, si sono utilizzate principalmente le funzioni contenute nel file header "termios.h" dell' API termios; inoltre funzioni supplementarie e talvolta necessarie si trovano negli header files "stdio.h", "fcntl.h", e "unistd.h". Presentiamo brevemente le funzioni della libreria termios:

- *struct termios*, è una struttura dati molto importante, utile sia per la configurazione di una porta seriale sia per la richiesta informazioni circa il valore di determinati parametri con cui la porta stessa è stata inizializzata e/o sta funzionando. Nel dettaglio:

```
struct termios
{
        tcflag_t c_iflag ;   // Input modes
        tcflag_t c_oflag ;   // Output modes
        tcflag_t c_cflag ;   // Control modes
        tcflag_t c_lflag ;   // Local modes
        cc_t c_cc[NCCS] ;    // Control characters
}
```

  - *c_ iflag*: queste opzioni consentono di controllare il processing sull'ingresso, che viene eseguito sui caratteri ricevuti sulla porta seriale;
  - *c_ oflag*: flags per le modalità di output che consentono di gestire il filtraggio dei dati in uscita dalla porta seriale;
  - *c_ cflag*: questi parametri consentono di controllare il baud rate, il numero di bits dei dati, la parità, il numero di stop bits usati e il controllo di flusso;
  - *c_ lflag*: flags di controllo locale utilizzati per scegliere come far gestire i caratteri in ingresso al driver della porta seriale. Solitamente la scelta è tra input di tipo canonico o raw;
  - *c_ cc*: questo array contiene definizioni di caratteri di controllo e parametri per gestire i timeout.

  Solitamente (in base all'implementazione di "termios.h") la struttura *termios* è ben più complessa di quella sopra presentata, che costituisce invece il modello basilare.

- **open()** e **close()** sono utilizzate per aprire/chiudere rispettivamente la porta seriale (o per meglio dire il file associato alla porta seriale stessa, in base a quanto detto precedentemente, che deve essere fornito come parametro di ingresso);

- **cfsetispeed()** e **cfsetospeed()**: vengono impiegate per impostare le velocità di ingresso e uscita per la porta seriale;

- **tcgetattr()**: serve per ottenere le opzioni (valori dei parametri) attualmente in uso per la porta seriale;

- **tcsetattr()**: serve per rendere valide modifiche effettuate ai parametri di una porta seriale;

- **read()**, **write()**: queste due funzioni consentono rispettivamente di leggere e scrivere dati sulla porta seriale. Una nota importante riguarda il valore di ritorno che le due funzioni restituiscono: nei parametri di ingresso, le due funzioni ricevono rispettivamente il numero di caratteri che devono essere letti dalla porta seriale e il numero di caratteri da scrivere sulla porta. Se si è scelto di operare con un I/O di tipo non bloccante (ossia dove le funzioni ritornano immediatamente, evitando di rimanere bloccate fino a quando la richiesta non è stata soddisfatta), le funzioni **read()** e **write()** restituiscono:

  - un intero, pari a -1 in caso si siano verificati degli errori o non sia possibile leggere o scrivere alcun carattere;
  - il numero di caratteri letti o scritti.

  Tra i parametri di ingresso della funzione, vanno specificati, oltre al numero di caratteri da leggere/scrivere, il file descriptor della porta seriale su cui operare e il carattere o l'array di caratteri da cui prelevare i dati da scrivere nel caso della system call **write()**, mentre per **read()**, il carattere o l'array di caratteri dove salvare i dati che vengono letti.

Possiamo schematizzare le azioni di un programma che svolge operazioni di I/O con la porta seriale nel seguente modo:

- la porta seriale viene aperta chiamando la system call **open()** e viene associata ad un certo file descriptor;

- vengono configurati i parametri di comunicazione ed altre opzioni dell'interfaccia tramite le funzioni specifiche contenute in "termios.h", salvando queste informazioni in una struttura dati di tipo *termios*;

- al completamento del setup della porta seriale, si può iniziare a lavorare con la porta stessa. Le due system calls **read()** e **write()** permettono di leggere e scrivere rispettivamente caratteri o messaggi composti da caratteri dalla porta;

- quando le operazioni di I/O sono concluse, la porta seriale viene chiusa con la system call di Unix **close()**.

E' importante sottolineare che al momento dell'apertura di una porta seriale, devono essere prese alcune decisioni in merito all'utilizzo che si intende fare della porta stessa. Ad esempio, la porta potrebbe essere impiegata solamente per leggere dati, solo per scrivere o ambedue le opzioni, inoltre è da specificare se utilizzare un I/O bloccante o non bloccante, in modo da rendere tali anche le chiamate alle funzioni **read()** e **write()**. Inoltre, l'API termios consente di trattare due differenti tipologie di I/O: il modo Canonical dove l'input è gestito linea per linea ed è consentito il pre-processing dei dati, e la modalità Non-Canonical o Raw che non prevede alcun pre-processing ed i dati sono gestiti carattere per carattere.

3.2.1 Funzioni per la gestione della porta seriale

Oltre al della libreria termios presentata brevemente qui sopra, è stato creato un codice C denominato "serial_port_functions.h" che contiene le funzioni che i dispositivi virtuali utilizzano per operare con la porta seriale. Tali funzioni si basano sulla libreria termios descritta in precedenza:

- **int openPortNumber(const char* portToOpen, char *deviceID)**: questa funzione apre la porta seriale specificata da parametro. Nel dettaglio:
  - *const char* portToOpen* specifica la porta seriale da aprire;
  - *char *deviceId* identifica invece il nome del dispositivo che compie l'operazione (apertura della porta);
  - la funzione restituisce il file descriptor associato alla porta seriale nel caso l'operazione di apertura della porta sia stata completata con successo o -1 in caso siano sopraggiunti degli errori.

- **void setControlOptions(int fileDescriptor, struct termios *options)**: questa funzione consente di impostare le opzioni di controllo in base a dei valori prestabiliti che sono stati decisi in accordo con il funzionamento dei dispositivi della rete e della tipologia delle comunicazioni degli stessi con i modem PLC. Tali valori sono salvati in una struttura *termios* (di cui si è parlato in precedenza). Inoltre:

  - *int fileDescriptor* specifica il file descriptor associato alla porta seriale su cui si intende operare;
  - *struct termios *options* è un puntatore ad una struttura dati di tipo *termios* che contiene i parametri della porta seriale da gestire.

- **int setSerialSpeed(speed_t speedSelected, struct termios *options, char *deviceID)**: consente di impostare la velocità di ingresso e di uscita per la porta seriale al valore indicato. Tale valore viene salvato nella struttura *termios* puntata dalla variabile *options*. Nel dettaglio:
  - *speed_t speedSelected* è il valore della velocità (ingresso e uscita) selezionato da una lista di possibili valori, ad esempio B9600;
  - la funzione restituisce 0 nel caso la sua esecuzione sia andata a buon fine, -1 altrimenti.

- **void setParity(int typeOfParity, struct termios *options, char *deviceID)**: questa funzione imposta la tipologia di parità che si intende utilizzare nella comunicazione seriale. Nello specifico:
  - *int typeOfParity* indica il tipo di parità utilizzato nella comunicazione. Tre sono i valori selezionabili: 0 nessuna parità, 1 per parità dispari, 2 per parità pari;

- **void setHardwareFlowControl(int activator, struct termios *options, char *deviceID)**, **setSoftwareFlowControl(int activator, struct termios *options, char *deviceID)**: queste due funzioni permettono di attivare/disattivare rispettivamente il controllo di flusso a livello hardware e software sulla base del valore della variabile *activator*. Nel dettaglio:
  - *int activator* è un valore *integer* che consente di attivare/disattivare il controllo di flusso: 0 per disattivarlo, 1 per attivarlo;

- **void setInputOutputOptions(int modeIn, struct termios *options, char *deviceID)**: con questa funzione si può selezionare la tipologia di I/O per la porta seriale. In particolare:
  - *int modeIn* permette di scegliere differenti modalità operative: 0 per modo Raw, 1 e 2 per modo Canonical con relative opzioni;

- **void setVMinAndVTime(int minChar, int timeForTimeout, struct termios *options, char *deviceID)**: questa funzione consente di impostare timeout e tempo inter-carattere. Più precisamente:

  - *int minChar* imposta il minimo numero di caratteri che la funzione dovrà leggere prima di ritornare;

  - *int timeForTimeout* indica invece il tempo inter-carattere;

Vengono ora presentati i dettagli dei dispositivi virtuali che sono stati realizzati in laboratorio.

3.3 Nemo Virtuale

Il Nemo D4-L+ è uno strumento che consente di effettuare misurazioni di tensione, corrente, potenza attiva, potenza reattiva, ecc. sia monofase che trifase all'interno di una rete elettrica. Esso non viene utilizzato solamente per visualizzare a display le necessarie informazioni relative a grandezze elettriche della rete, ma anche al fine di comunicare tali dati ad un dispositivo esterno a lui connesso (nel nostro caso un modem PLC) quando quest'ultimo gli invia opportuni messaggi. Queste richieste giungono al Nemo tramite una porta seriale che funziona ad una velocità di 9600 baud. Gli altri parametri della comunicazione via RS-232 prevedono un utilizzo di 8 bits, uno stop bit e nessun bit di parità. Il Nemo si serve di una memoria di ridotte dimensioni per salvare i risultati delle rilevazioni eseguite sulla rete. Sono utilizzati tre tipi di formati per rappresentare un dato nella memoria: il byte, la word costituita da due bytes che costituisce l'unità dati di base ed infine il long che è composto da due parole (word) ossia da quattro bytes. I valori nella memoria del nemo sono tutti positivi, i segni delle quantità sono salvati in locazioni di memoria dedicate.

3.3.1 Protocollo di comunicazione

Il protocollo di comunicazione adottato dal nemo consta di messaggi composti nel seguente modo:

Richiesta

Instrument Address	Functional Code	Data	CRC Word

Risposta con dati

Instrument Address	Functional Code	Data	CRC Word

Risposta con errore

Instrument Address	Functional Code + 0x80	Error Code	CRC Word

Spieghiamo brevemente il significato di tali campi:

- *Instrument Address*: è il numero di identificazione dello strumento all'interno della rete; deve essere lo stesso sia per i messaggi di richiesta che per quelli di risposta. Ha una dimensione di un byte, con possibili valori da 0 a 0xff;

- *Functional Code*: è un codice di dimensione pari ad 1 byte che può assumere due valori, 0x03 e 0x10. Il primo, 0x03 serve per richiedere la lettura di parole (formate da più bytes) consecutive mentre il secondo, 0x10 per scrivere più parole consecutivamente;

- *Data*: se il messaggio è di richiesta, è formato dall'indirizzo e dal numero delle parole richieste mentre se il messaggio è di risposta è formato semplicemente dai dati richiesti;

- *CRC Word*: è il risultato del calcolo del cyclic redundacy check (crc) a 16 bit eseguito su tutti i bytes che costituiscono il messaggio. Il valore di questo campo è piuttosto importante perché consente allo strumento di verificare la presenza di errori dovuti alla ricezione del messaggio o a malfunzionamenti del modem o della linea seriale.

I messaggi che il modem PLC invia al Nemo sono solamente richieste di lettura dati, quindi con *Functional Code* pari a 0x03; pertanto ci limiteremo

ad esporre le funzionalità del Nemo relative a tale modalità di funzionamento. Vediamo ora nel dettaglio lo schema di un comando:

Richiesta lettura dati - Codice 0x03

BYTE	BYTE	MSB LSB	MSB LSB	MSB LSB
Instrument Address	Functional Code	First Word Address	WORDS Number	CRC16

Ciascuna richiesta è quindi formata da 8 byte complessivamente. Le risposte possibili sono invece di due tipi:

Risposta di tipo data - Codice 0x03

BYTE	BYTE	MSB LSB	MSB LSB	MSB LSB	MSB LSB
Instrument Address	Functional Code	BYTES Number	WORDS 1..	WORD N.	CRC16

Risposta in caso di errore - Codice 0x03

BYTE	BYTE	MSB LSB	MSB LSB
Instrument Address	Functional Code +0x080	Error Code	CRC16

Un messaggio di errore viene inviato come risposta al modem quando la richiesta giunta è errata: il campo *Error Code* specifica la tipologia di errore riscontrato: può trattarsi di un *Functional Code* non corretto, in tal caso il codice di errore è 0x01 o di un *First Word Address* errato, in tal caso invece il codice di errore è 0x02.

La memoria del Nemo D4-L+ è costituita da due tabelle che contengono i dati relativi a variabili o gruppi di variabili misurate dallo strumento. Quando il Nemo riceve, tramite porta seriale, una opportuna richiesta preleva le informazioni necessarie dalle due tabelle di memoria. Il modem PLC interroga il Nemo in modo da ottenere informazioni relative a:

1. Tensione: in questo caso, il valore del *First Word Address* è 0x301, sono richieste 6 words (2 per ciascuna fase) e nessuna informazione relativa ai segni di tali quantità. I valori salvati nella memoria dello strumento hanno una precisione massima di 3 cifre decimali. Il messaggio di risposta, di tipo data, è formato da 17 bytes;

2. Corrente: in questo caso, il valore del *First Word Address* è 0x30d, sono richieste 6 words (2 per ciascuna fase) e nessuna informazione relativa ai segni di tali quantità. I valori salvati nella memoria del Nemo hanno

una precisione (massima) di 3 cifre decimali. Il messaggio di risposta, di tipo data, è formato da 17 bytes;

3. Potenza Attiva: in questo caso, il valore del *First Word Address* è 0x102c, sono richieste 6 words (2 per ciascuna fase) e 3 words per i segni di tali valori (una word per il segno di ogni potenza, con 0 che indica una quatità positiva e 1 una quantità negativa). I valori salvati nella memoria del dispositivo hanno una precisione di 2 cifre decimali. Il messaggio di risposta, di tipo data, è formato da 23 bytes;

4. Potenza Reattiva: in questo caso, il valore del *First Word Address* è 0x1035, sono richieste 6 words (2 per ciascuna fase) e 3 words per i segni di tali valori (una word per il segno di ogni potenza, con 0 che indica una quatità positiva e 1 una quantità negativa). I valori salvati nella memoria dello strumento hanno una precisione di 2 cifre decimali. Il messaggio di risposta, di tipo data, è formato da 23 bytes.

Quanto presentato finora, costituisce l'insieme di funzionalità di cui il Nemo virtuale, cioè la versione software del Nemo D4-L+ dovrà disporre. Riassumendole brevemente esse sono:

- lettura e scrittura di dati utilizzando la porta seriale;
- una struttura dati che rappresenta la memoria del dispositivo virtuale, contenente le due tabelle che costituiscono la memoria del Nemo D4-L+ di cui si è parlato in precedenza;
- per ciascun messaggio letto dalla porta seriale, è necessario effettuare una analisi preliminare per verificare che quanto ricevuto non contenga errori, cioè il calcolo del crc restituisca zero. Se questo è vero, l'analisi del messaggio continua, al fine di ottenere i valori per i campi *Functional Code*, *First Word Address* e *Words Number*;
- una volta ottenuti i valori di *Functional Code*, *First Word Address* e *Words Number*, questi vengono esaminati in modo da controllarne la correttezza: se sono presenti errori, viene inviato un messaggio di risposta al modem PLC contenente il codice di errore rilevato. Se invece i campi del messaggio sono corretti, il Nemo virtuale individua la tipologia di richiesta che gli è giunta e compone il messaggio di risposta prelevando i dati dalla memoria.

Per realizzare il Nemo virtuale, si é preso come riferimento il codice salvato nella memoria del modem PLC che si occupa della comunicazione con il

dispositivo Nemo D4-L+. Questo codice, invia in successione e ad intervalli regolari tramite porta seriale, messaggi di richiesta per conoscere la tensione, la corrente, la potenza attiva e la potenza reattiva misurati dallo strumento di rete. Se una risposta non arriva entro un certo intervallo di tempo al modem o se essa è errata in qualche suo campo, il modem invia nuovamente la richiesta, e prosegue con questo modalità operativo fino a quando il messaggio proveniente dal Nemo non è corretto. Un ciclo di richieste si conclude solamente se tutti i messaggi di risposta, per tensione, corrente, potenza attiva e reattiva giungono nell'ordine al modem.

Il manuale del Nemo D4-L+ [4] contiene tutti i dettagli relativi alle modalità di comunicazione e formato dei dati inviati tramite porta seriale (che sono stati illustrati in precedenza), inoltre lo si è utilizzato anche per la creazione di una memoria di immagazzinamento dati che rispecchiasse quella presente nello strumento fisico. Per la gestione della comunicazione con il modem, che avviene tramite l'utilizzo di una porta seriale dell'elaboratore su cui il software è eseguito, ci si è avvalsi delle funzioni contenute nel file "serial_port_functions.h" che sono state presentate nella sezione precedente. Dato che il Nemo reale opera solamente con quantità positive, per la sua controparte virtuale si sono impiegati valori interi senza segno, ossia *uint8_t*, *uint16_t* e *uint32_t* (utilizzabili includendo l'header "stdint.h").

3.3.2 Funzioni del Nemo virtuale

Il software che consente di emulare il Nemo D4-L+ è composto da due file distinti: un primo, "virtual_nemo_functions.h" è un file C appositamente realizzato che contiene tutte le funzioni di cui il Nemo virtuale dispone, oltre alla struttura dati che ne costituisce la memoria. In un secondo file, si trova invece il thread che simula il comportamento vero e proprio dello strumento reale:

- **struct virtualNemoMemory**: questa struttura dati emula la memoria del Nemo D4-L+ dove sono contenuti i valori di tensione, corrente, potenza attiva e potenza reattiva letti dallo strumento stesso. Vengono inoltre salvati temporaneamente in questa struttura i messaggi di risposta che il dispositivo provvede ad inviare al modem a lui connesso, aggiornati ad ogni nuova richiesta ricevuta;

- **uint16_t calculate_crc(uint8_t *buffer, uint16_t buffer_dimension)**: questa funzione permette di calcolare il cyclic redundacy check (crc) a 16 bit per un buffer, ossia un'area di dati di dimensione specificata. In particolare:

  - *uint8_t *buffer* è il puntatore al buffer su cui viene eseguito il calcolo del crc;
  - *uint16_t buffer_dimension* specifica la lunghezza del buffer;
  - la funzione **calculate_crc** restituisce un *unsigned integer* a 16 bit che costituisce il risultato del calcolo del cyclic redundacy check.

- **void setVirtualNemoMemory(float valueTension, float valueCurrent, float valueActivePower, float valueReactivePower, struct virtualNemoMemory *memory, int numberOfPhasesUsed)**: questa funzione permette di salvare nella memoria del Nemo virtuale i valori contenuti nelle variabili *valueTension*, *valueCurrent*, *valueActivePower* e *valueReactivePower* che indicano rispettivamente la tensione, la corrente, la potenza attiva e la potenza reattiva di un certo componente della rete elettrica virtuale. I valori vengono inseriti nella struttura puntata da *memory*, la variabile *numberOfPhasesUsed* invece specifica se la rete sta operando in regime monofase o trifase. Importante nota, la precisione, intesa come numero massimo di cifre decimali che vengono considerate per ciascun numero *float* originario: per tensione e corrente, la precisione massima è di 3 cifre decimali, mentre per la potenza attiva e reattiva è solamente di due cifre decimali;

- **uint8_t *extractValuesFromMemory(uint16_t initialAddressMem, uint16_t addressForData, int numberOfValuesRequested, struct virtualNemoMemory *memory)**: preleva dalla memoria del Nemo virtuale i dati a partire dall'indirizzo specificato da *addressForData* e in quantità indicata da *numberOfValuesRequested*. Nel dettaglio:

  - *uint16_t initialAddressMem* è l'indirizzo di memoria di base che consente di conoscere se i dati devono essere prelevati dalla prima tabella di variabili o dalla seconda;
  - *uint16_t addressForData* rappresenta l'indirizzo specifico della memoria da cui iniziare a prelevare i dati;
  - *int numberOfValuesRequested* indica il numero esatto di valori da leggere;
  - la funzione ritorna un array di *unsigned bytes* dove sono salvati i dati richiesti.

- **uint8_t* getData(uint16_t dataAddress, uint16_t wordsRequested, struct virtualNemoMemory *memory)**: in base all'in-

dirizzo di memoria specificato dal parametro *dataAddress*, questa funzione riconosce la tipologia di informazione contenuta nella richiesta ricevuta dal modem (può trattarsi come detto della tensione, della corrente, della potenza attiva o potenza reattiva) e preleva i valori dalla memoria del Nemo virtuale (puntati dalla variabile *memory*). Questo è ottenuto chiamando la funzione **extractValuesFromMemory()**, i dati sono salvati in un array di *uint8_t*;

- **uint8_t checkFirstWordAddress(uint16_t firstWordAddress)**: con questa funzione è possibile verificare se l'indirizzo di memoria contenuto nel parametro d'ingresso *firstWordAddress* è valido, ossia se rientra tra quelli previsti dal manuale del Nemo D4-L+;

- **int createDataAnswer(uint8_t functionalCode, uint16_t firstWordAddress, uint16_t wordsNumber, struct virtualNemoMemory *memory)**: consente di creare un messaggio di risposta di tipo data, la cui lunghezza dipende dal valore specificato dal parametro di ingresso *wordsNumber*. Infatti, in caso venga richiesta una tensione od una corrente, il messaggio sarà costituito da 17 bytes mentre se si deve fornire la potenza attiva o reattiva tale valore aumenterà a 23 bytes in modo da contenere anche i segni di tali potenze. I valori dei parametri d'ingresso *functionalCode*, *firstWordAddress* e *wordsNumber* sono utilizzati per completare i campi iniziali della risposta. Quando tutti i dati sono stati inseriti nel messaggio, su di esso viene invocata la funzione **calculate_crc()** al fine di calcolare il crc a 16 bit, utilizzato per la verifica di errori di ricezione; questo valore viene poi inserito in coda al messaggio stesso. Quando la risposta è completa, viene salvata temporaneamente nella memoria del Nemo virtuale (puntata da *memory*) prima di essere inviata al modem tramite porta seriale. La funzione restituisce un valore intero pari a 0 se il messaggio di risposta è stato correttamente creato, -1 altrimenti;

- **void createErrorAnswer(uint8_t functionalCode, uint8_t errorCode, struct virtualNemoMemory *memory)**: crea un messaggio di risposta identificato dalla tipologia di errore riscontrato da *errorCode*. Il messaggio ha una lunghezza fissa di 5 bytes e viene salvato nella struttura dati puntata da *memory*;

- **int messageAnalyzer(unsigned char *messageReceived, struct virtualNemoMemory *memory)**: questa funzione analizza la richiesta ricevuta dal modem. Il messaggio viene dapprima controllato con la funzione **calculate_crc()** per verificare che non siano presenti

errori relativi all'operazione di ricezione dei dati tramite porta seriale. Successivamente vengono esaminati i campi *Functional Code* e *First Word Address* del messaggio. In base agli esiti di tali verifiche, il dispositivo virtuale stabilirà la tipologia di risposta da inviare al modem. La funzione restituisce un valore intero, pari a zero nel caso nessun errore venga riscontrato, mentre è -1 se qualche campo del messaggio ricevuto è incorretto;

- **int writeToModem(int analysisResult, struct virtualNemoMemory *memory, char *deviceName, int fileDescriptor)**: in base al valore della variabile *analysisResult* (che rappresenta il valore di ritorno della funzione **messageAnalyzer()**), la funzione **writeToModem()** invia un messaggio di errore o dati in risposta alla richiesta ricevuta dal modem. Per fare ciò, viene invocata la system call **write()**, fornendo come parametri il file descriptor della porta seriale su cui inviare tali dati, contenuto nella variabile *fileDescriptor*, e il messaggio di risposta e la relativa dimensione (creati dalle funzioni **createDataAnswer()** o **createErrorAnswer()**) salvati nella memoria del nemo virtuale puntata da *memory*. La funzione **writeToModem()** segnala un errore nel caso la funzione **write()** invii una quantità errata di bytes (solitamente minore di quanto richiesto), restituendo un intero pari a -1. Se invece l'invio di dati è avvenuto correttamente, 0 è il valore di ritorno.

3.3.3 Il thread VirtualNemo

Le funzioni appena presentate costituiscono i blocchi fondamentali per il funzionamento del Nemo virtuale, a cui si aggiunge la struttura di tipo *virtualNemoMemory* che costituisce la memoria del Nemo virtuale. Come detto, il nemo virtuale vero e proprio è costituito da un thread, a cui è connessa una diversa e ulteriore struttura, chiamata *virtualNemoData* che contiene invece tutte le informazioni necessarie al corretto funzionamento del dispositivo. Sono presenti campi per:

- porta seriale dell'elaboratore da utilizzare per le comunicazioni con strumenti esterni, relativa velocità specificata in baud rate al secondo e strutture di tipo *termios* in cui salvare i parametri con cui viene configurata la porta;

- abilitazione/disabilitazione della comunicazione via seriale (non tutti i Nemo infatti necessitano di scambiare dati con modem e simili);

- identificatore del dispositivo virtuale e il suo stato (attivo/disattivo);
- semafori di mutua esclusione per l'accesso ai campi della struttura senza causare interferenza con gli altri thread, nello specifico il thread che gestisce la rete elettrica virtuale;
- la sottorete di componenti elettrici virtuali dove il Nemo esegue le misurazioni delle grandezze elettriche.

Esaminiamo ora le azioni svolte dal thread del Nemo virtuale:

1. per prima cosa, la porta seriale viene aperta associandola ad un file descriptor, vengono poi impostati i parametri della comunicazione (baud-rate, bit di parità, input canonico o raw, minimo numero di caratteri da leggere e tempo inter-carattere) che sono gli stessi utilizzati dallo strumento Nemo D4-L+ reale. Sono eseguite delle verifiche per controllare che le opzioni specificate siano state applicate nel modo corretto; se nessun errore viene riscontrato la porta seriale può essere utilizzata senza problemi;
2. lo stato della porta seriale viene periodicamente controllato dal Nemo virtuale: nel caso nessun dato venga letto dalla porta in un certo lasso di tempo, il Nemo virtuale interrompe la comunicazione e chiude la porta seriale con la system call **close()**. Questo di solito avviene quando il modem PLC è scollegato oppure presenta un malfunzionamento;
3. il Nemo virtuale suddivide i messaggi ricevuti dal modem in cicli di richieste: un ciclo di richieste inizia quando il modem PLC invia il comando per conoscere la tensione, e termina con il comando per conoscere la potenza reattiva. E' importante però evidenziare che al fine di ritenere valido un ciclo di richieste, queste devono essere giunte in modo continuativo e senza interruzioni, altrimenti il ciclo viene azzerato e fatto ripartire da capo fino alla sua conclusione. Per ogni nuovo ciclo di richieste, il Nemo virtuale legge i valori di tensione, corrente, potenza attiva e reattiva dell'impedenza della rete elettrica virtuale a lui associata; questi valori vengono salvati all'interno della memoria del dispositivo virtuale chiamando la funzione **setVirtualNemoMemory()**;
4. il Nemo virtuale attende la ricezione di una richiesta da parte del modem a lui connesso tramite porta seriale. Una richiesta è rappresentata da una sequenza di 8 caratteri esadecimali (8 bytes quindi) in cui viene specificata la tipologia della richiesta, nel nostro caso una lettura di dati, l'indirizzo di memoria in cui prelevare queste informazioni e

il numero di bytes che devono essere letti: il valore dell'indirizzo di memoria (campo *First Word Address* del messaggio) consente di capire quale grandezza si sta richiedendo: 0x301 per la tensione, 0x303 per la corrente, 0x102c per la potenza attiva e 0x1035 per la potenza reattiva. Nella parte terminale della richiesta, il modem inserisce il risultato del crc a 16 bit calcolato sul messaggio inviato;

5. quando una nuova richiesta è ricevuta, questa viene analizzata dettagliatamente, chiamando la funzione **messageAnalyzer()**. Innanzitutto viene verificata la correttezza del messaggio ricevuto, o per meglio dire l'assenza di errori dovuti alla ricezione dei dati tramite porta seriale. Per fare ciò viene invocata la funzione che calcola il crc del messaggio, **calculate_crc**: se il valore restituito come output dalla funzione non è zero il messaggio è stato corrotto nella fase di trasmissione (o per un malfunzionamento della porta seriale), il Nemo virtuale in questa situazione non invia alcuna risposta al modem PLC. Quest'ultimo, allo scadere di una temporizzazione prefissata, provvederà ad inoltrare nuovamente la richiesta. Nel caso invece il calcolo del crc abbia valore zero, cioé il messaggio è corretto, sono eseguiti dei controlli, in particolare per verificare che la tipologia della richiesta e i campi *Instrument Address*, *Functional Code* e *First Word Address* del messaggio rientrino tra quelli previsti dal dispositivo Nemo D4-L+;

6. sulla base dei risultati forniti dall'analisi della richiesta ricevuta dal modem, viene creato il messaggio di risposta: se non sono riscontrati errori, il messaggio creato contiene i valori della grandezza elettrica che il modem PLC ha richiesto. In caso contrario, viene creato un messaggio di errore, dove tramite un apposito codice viene segnalata al modem la causa del malfunzionamento. Qualunque sia la tipologia di risposta, questa viene completata inserendo i campi *Instrument Address*, *Functional Code*, *Bytes Number* o *Error Code* (rispettivamente per risposte di tipo data e error) e in coda al messaggio, il crc a 16 bit;

7. il messaggio viene infine inviato al modem PLC.

Il funzionamento descritto in precedenza viene ripetuto in un ciclo *while*. Il ciclo si interrompe solamente se il Nemo virtuale non riceve più alcun comando dal modem in un certo intervallo di tempo. Il Nemo virtuale, utilizza una variabile di controllo per valutare lo stato degli errori in lettura dalla porta seriale, che funziona come un contatore. Viene fissata una soglia massima per gli errori, superato tale limite il Nemo interrompe la comunicazione sulla

porta seriale e l'esecuzione del thread che lo rappresenta all'interno della rete termina. Il contatore è incrementato ad ogni errore riscontrato. Quando invece, vengono lette nuove richieste inviate dal modem PLC, il contatore degli errori viene azzerato. Un dettaglio importante riguarda l'utilizzo della funzione **read()** della libreria termios per la lettura di messaggi dalla porta seriale. Dato che tutte le richieste provenienti dal modem hanno la stessa dimensione, pari a 8 caratteri, la funzione **read()** avrà tra i propri argomenti di ingresso proprio l'intero 8, il che significa che il Nemo virtuale si aspetta di leggere esattamente 8 caratteri dalla porta seriale.

3.3.4 Schemi riassuntivi

In Figura 3.2, è riportato lo schema di funzionamento del Nemo virtuale, dove sono riassunte le azioni principali da esso svolte mentre è in esecuzione. In Figura 3.3 è invece presente un ulteriore schema dove vengono descritti con maggior dettaglio, i blocchi di "analisi del messaggio" e "creazione del messaggio di risposta" presenti nello schema di Figura 3.2.

3.4 Carico Programmabile Virtuale

Il dispositivo Chroma 63804 AC/DC Load è uno strumento che consente, grazie a numerose modalità di funzionamento, di simulare carichi con valori di crest factor e power factor variabili. Inoltre, dispone di una modalità remota con la quale è possibile, per dipositivi esterni, controllarlo tramite appositi comandi. In particolare nella postazione di test del laboratorio è collegato ad un modem PLC (identificato come master) tramite la porta seriale RS-232.

3.4.1 Protocollo di comunicazione

Il protocollo di comunicazione adottato dal Chroma 63804 è lo Standard Commands for Programmable Instruments (SCPI) di cui viene data una breve descrizione qui di seguito. I comandi del carico si basano su una struttura gerarchica, definita *tree system*. Per ottenere un determinato comando, l'intero percorso per tale comando deve essere specificato, inserendo verso destra i nodi più esterni di tale percorso (albero) e nella posizione più a sinistra il nodo che si trova più in alto in tale gerarchia. Per identificare uno specifico comando, vengono utilizzate delle parole chiave definite *program headers*. Un comando può essere costituito da più program headers, questi vanno separati

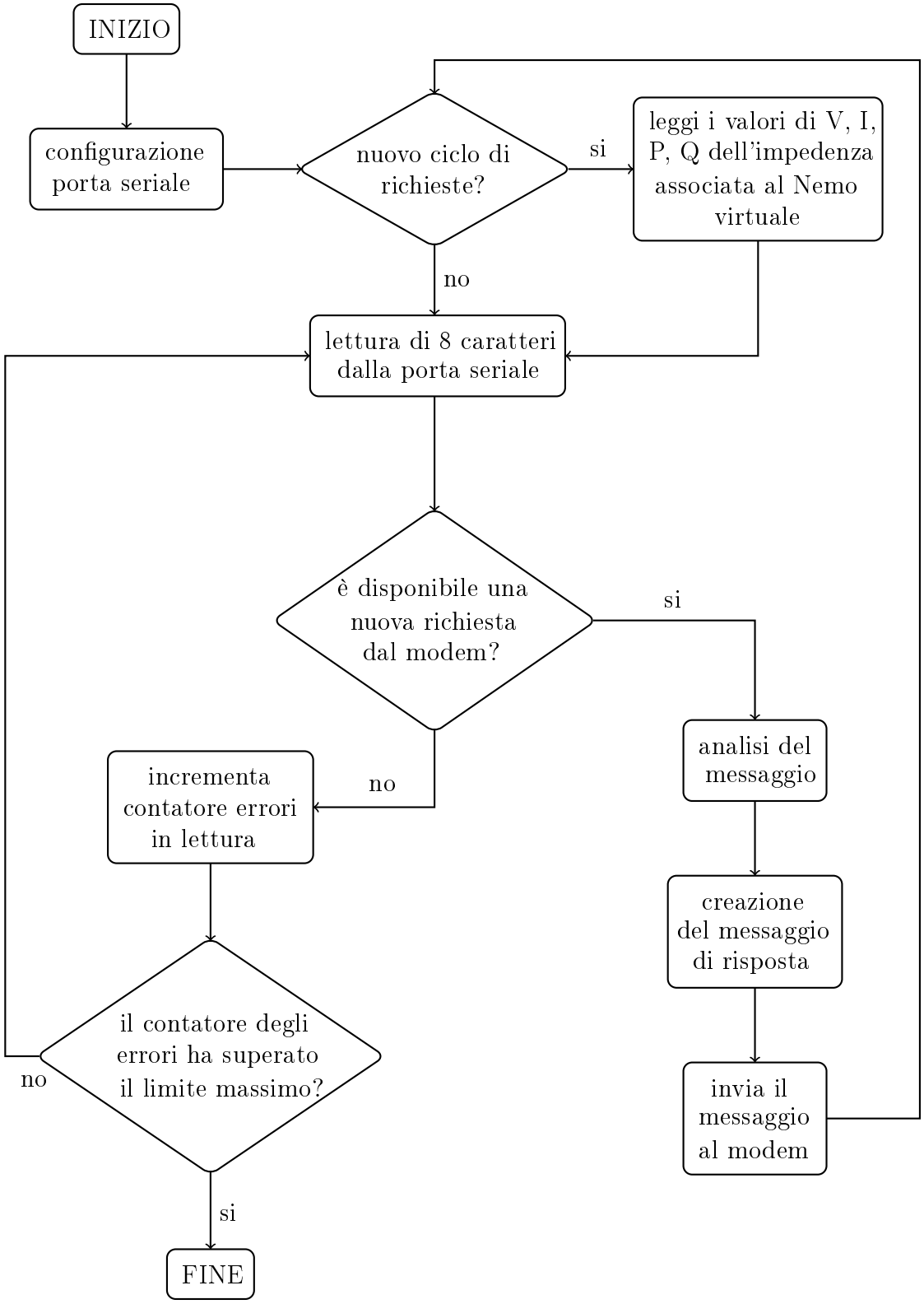

Figura 3.2: Schema di funzionamento del Nemo virtuale

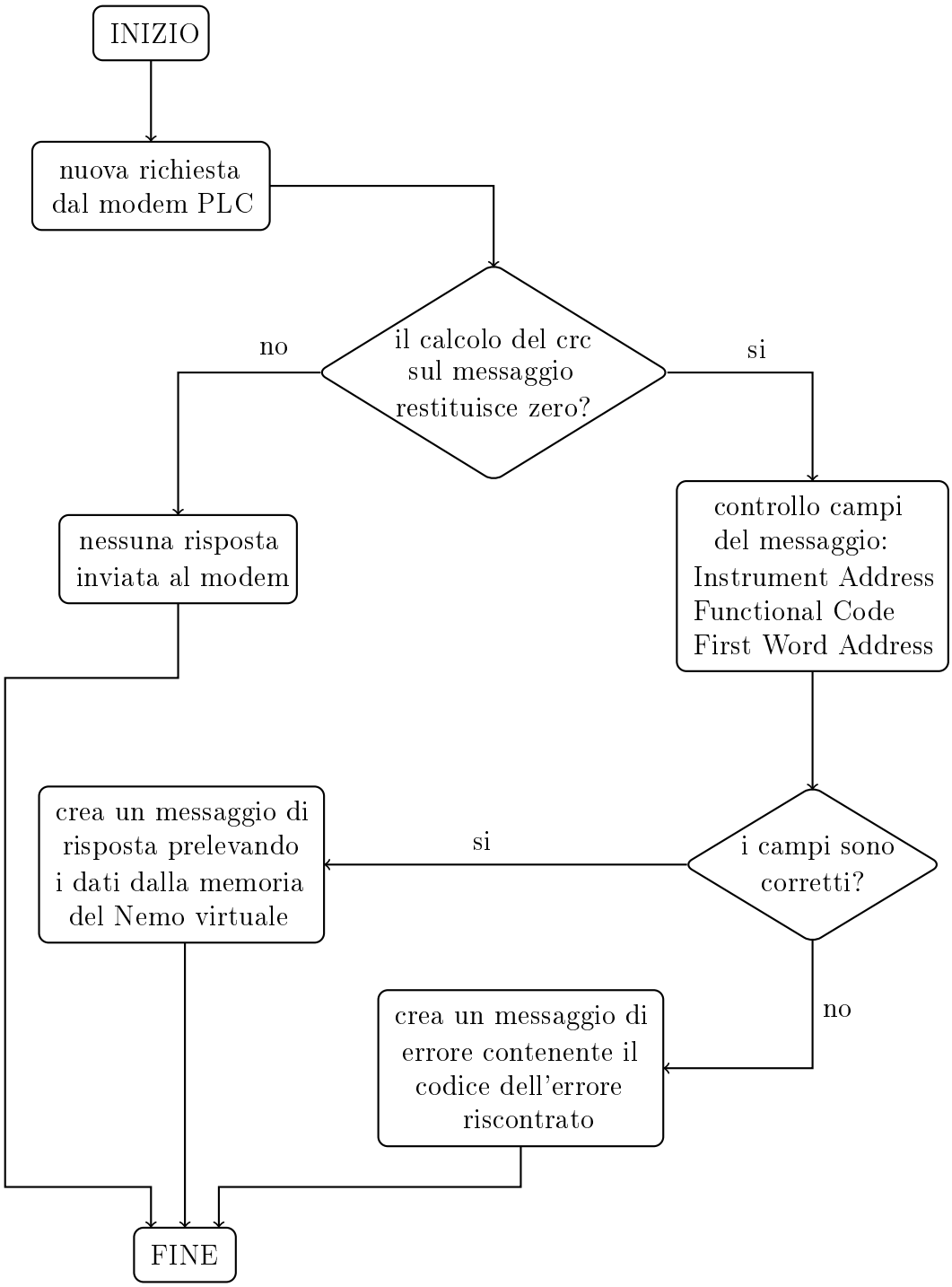

Figura 3.3: Dettaglio dell'analisi di un messaggio di richiesta del Nemo virtuale

tramite il carattere ':'. Un comando può essere accompagnato anche con dei dati, questi vanno separati dal program header con uno spazio.

Esempio: VOLT:DC 10

Il comando è formato da due program headers, VOLT e DC, separati come detto da ':'. Per il program header DC è previsto anche l'uso di un valore, in questo caso 10, separato da uno spazio rispetto al comando.

I messaggi che il carico programmabile riceve quando opera in modalità remota sono detti *program messages*, ciascuno costituito da una o più *program message unit*. Ciascuna program message unit costituisce un singolo comando, un valore da impostare o una query, le program message unit sono tra loro separate dal carattere ';'.

Ad esempio: CURR? e LOAD ON rappresentano due singole program message unit. Un program message può ad esempio essere il seguente: CURR 20;CFAC 2<PMT>, formato da due comandi separati dove <PMT> è definito Program Message Terminator, ossia è il carattere che identifica il termine del messaggio. Generalmente, il <PMT> è il carattere '\n'.

Riassumendo, il protocollo di comunicazione del carico programmabile si può descrivere in questo modo:

- i messaggi che il carico programmabile riceve quando è in modalità remota sono definiti program message;
- un program message è formato da uno o più comandi, chiamati program message unit;
- ogni program message unit, rappresenta un comando, che può essere composto da una o più program headers, delle parole chiave che identificano i comandi stessi del carico. i program headers hanno una struttura gerarchica e modulare, e consentono di specificare una ampia varietà di comandi.

I comandi che il protocollo SCPI utilizza sono di due tipologie distinte:

- i comandi di base (common commands) riconoscibili perché iniziano sempre con un '*', sono formati da tre lettere e possono comprendere un '?' alla fine. Questi comandi sono utilizzati per impostare valori dei registri del carico programmabile o ottenere informazioni sullo stato dei registri stessi, quando è presente il carattere '?'. In quest'ultimo caso, il carico programmabile provvede all'invio di un messaggio di risposta tramite porta seriale. E' inoltre possibile specificare un valore associato ad un determinato comando, da cui deve essere separato come detto da uno spazio.

  Esempio: *CLS, *IDN?, SRE 32

- comandi composti da program headers distinti che consentono di richiedere misure di tensioni, correnti e potenze, impostare modalità di funzionamento del dispositivo o valori per simulazioni di carico (come power e crest factor o valore della potenza attiva) e anche attivare/disattivare lo strumento.

  Esempio: SYST:SET:MODE AC\n

3.4.2 Funzioni del carico programmabile virtuale

Per realizzare il carico programmabile virtuale, si è preso come riferimento il codice salvato nella memoria del modem che si occupa della comunicazione con il dispositivo Chroma 63804 presente in laboratorio. Inoltre, si è utilizzato il manuale di riferimento del carico programmabile Chroma 63804 [3]. Per la gestione della comunicazione con il modem, che avviene come detto tramite l'utilizzo di una porta seriale, ci si è avvalsi delle funzioni contenute nel file "serial_port_functions.h". Di seguito vengono elencate e brevemente descritte le funzioni che consentono di operare e gestire il dispositivo carico programmabile virtuale raccolte nel file C "virtual_programmable_load_functions.h", realizzato in laboratorio:

- **struct programmableLoadVariables**: questa struttura dati raccoglie alcune variabili che rappresentano tutti i parametri, le modalità di funzionamento e le opzioni che sono presenti all'interno del carico programmabile reale, il Chroma 63804;

- **void initializeProgrammableLoadDevice(struct programmableLoadVariables *loadVars)**: questa funzione inizializza i valori delle variabili della struttura *programmableLoadVariables* puntata da *loadVars* ai valori di default;

- **void set_command_actions(int fileDescriptor, char *command, struct programmableLoadVariables *loadVars, char *loadID)**: tramite questa funzione il carico virtuale identifica il comando che gli è stato comunicato dal modem ed esegue le azioni prestabilite. Non è previsto alcun parametro come argomento della funzione. I comandi senza parametro che il modem invia al carico programmabile virtuale sono solamente delle interrogazioni che pertanto richiedono l'invio di dati in risposta tramite la porta seriale;

- **void set_command_actions_parameter(int fileDescriptor, char *command, char *parameter, struct programmableLoadVariables *loadVars, char *loadID)**: la funzione riconosce il co-

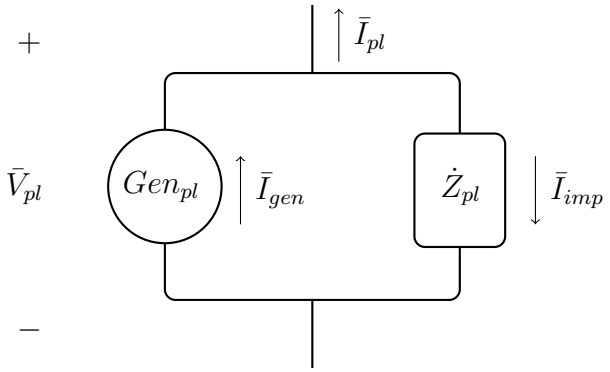

Figura 3.4: Schema a generatore di potenza

mando contenuto nella variabile *command* ed esegue le necessarie azioni utilizzando il parametro specificato come argomento dalla variabile *parameter*. In questo caso sono previste modifiche ai campi della struttura dati di tipo *programmableLoadVariables* puntata da *loadVars* ma i messaggi inviati dal modem non costituiscono delle interrogazioni al dispositivo virtuale pertanto non è necessario l'invio di alcuna risposta con la porta RS-232;

- **void analyze_command(int fileDescriptor, char *readStringCommand, int commandLength, struct programmableLoadVariables *loadVars, char *loadID)**: questa funzione analizza un messaggio (salvato nella variabile *readStringCommand* di lunghezza *commandLength*) ricevuto dal modem connesso al carico programmabile virtuale. Le richieste che il modem invia hanno tutte lunghezza variabile ma sono separate tra loro dal carattere '\n'. Ciascun messaggio è composto da un comando e da un eventuale parametro, che la funzione provvede a riconoscere analizzando l'array passato come argomento della funzione carattere per carattere. L'elemento che consente di distinguere comando e parametro è uno spazio bianco, ossia il carattere ' '. In base alla presenza del parametro o meno, viene rispettivamente chiamata la funzione **set_command_actions()** o la funzione **set_command_actions_parameter()**.

3.4.3 Modello elettrico per il carico programmabile virtuale

Finora si è detto che il carico programmabile ha la funzione di simulare appunto un carico nella rete elettrica che viene configurato in base ai valori

specificati per la potenza attiva, il power factor ed il crest factor. Per simulare il carico programmabile all'interno della rete elettrica virtuale, è stato utilizzato uno schema a generatore di potenza, costituito da un parallelo tra un generatore di corrente e una impedenza. Lo schema è riportato in Figura 3.4. La tensione e la corrente del carico programmabile, o per meglio dire del generatore di potenza a cui è associato il thread che rappresenta il carico programmabile, sono rispettivamente indicate nella figura come $\bar{V}_{pl}$ e $\bar{I}_{pl}$. Essendo i due lati in parallelo, essi avranno la medesima tensione. Con i riferimenti adottati nello schema, la corrente complessiva del carico programmabile si può calcolare semplicemente sommando le correnti (rispettando i riferimenti posti) $\bar{I}_{gen}$ del generatore di corrente e $\bar{I}_{imp}$ che scorre nell'impedenza $\dot{Z}_{pl}$. Per simulare quindi differenti tipologie di carico, sono stati utilizzati valori variabili della corrente impressa del generatore di corrente Gen_{pl} e dell'impedenza $\dot{Z}_{pl}$.

3.4.4 Il thread VirtualProgrammableLoad

Le semplici funzioni implementate per emulare il comportamento dello strumento Chroma 63804 sono sufficienti per gestire la comunicazione con il modem PLC. Al contrario di quanto avviene nel Nemo virtuale, dove è presente una forma di interazione continua con il modem, il carico programmabile riceve semplicemente continui aggiornamenti dalla porta seriale, sempre riguardanti tre parametri specifici. In pratica quindi, il carico programmabile si limita semplicemente a riconoscere i comandi inviati dal modem e impostare le azioni previste per tali comandi.

Oltre alla struttura di tipo *programmableLoadVariables* dove sono contenuti informazioni su registri e modalità di funzionamento del carico programmabile, il thread che costituisce il dispositivo virtuale utilizza un'altra struttura di supporto, *virtualProgrammableLoadData* dove sono invece salvati i dati necessari al corretto funzionamento del dispositivo. Questi riguardano:

- la porta seriale dell'elaboratore da utilizzare per le comunicazioni con strumenti esterni, relativa velocità specificata in baud rate al secondo e strutture di tipo *termios* in cui salvare i parametri con cui viene configurata la porta. Anche per il carico programmabile virtuale, tale configurazione va eseguita solamente se lo scambio di dati con il modem PLC è previsto;

- l' identificatore del dispositivo virtuale e il suo stato (attivo/disattivo);

- semafori di mutua esclusione per l'accesso ai campi della struttura senza causare interferenza con gli altri thread, nello specifico il thread che gestisce la rete elettrica virtuale;
- il generatore di potenza della rete elettrica virtuale cui è associato il carico programmabile.

Dopo aver descritto le funzioni raccolte nel file C "virtual_programmable_load_functions.h", analizziamo in dettaglio il funzionamento del thread che realizza il carico programmabile virtuale vero e proprio:

1. per prima cosa, la porta seriale viene aperta associandola ad un file descriptor, vengono in seguito impostati i parametri della comunicazione (baud-rate, bit di parità, input canonico o raw, minimo numero di caratteri da leggere e tempo inter-carattere) che sono stati ricavati dal manuale del carico. Se non si sono verificati errori nell'impostazione delle opzioni, la porta seriale può essere utilizzata nel modo corretto;

2. la comunicazione del carico programmabile virtuale con il modem PLC può essere suddivisa in due fasi distinte: una prima fase in cui il modem esegue operazioni di verifica sul carico, richiedendo l'identificatore del dispositivo, lo stato di alcuni registri, e impostando le modalità di funzionamento. Infine, il modem predispone l'attivazione del carico stesso. In questa prima fase, il modem si aspetta di ricevere risposte ad alcuni comandi di interrogazione inviati al carico virtuale, che pertanto dovrà rispondere con adeguati messaggi.

   Terminata la fase di setup iniziale, il modem invia periodicamente al carico programmabile una terna di comandi in successione dove specifica i valori della potenza attiva, del power factor e del crest factor che il carico programmabile deve impostare.

   E' importante sottolineare che, a differenza di quanto avveniva con il Nemo virtuale dove ogni nuovo messaggio ricevuto dalla porta seriale costituiva una nuova richiesta cui seguiva una risposta opportuna, nel caso del carico virtuale il modem PLC non si aspetta alcun tipo di feedback (fatto salvo per due soli comandi nella fase di inizializzazione), quindi continua ad inviare i suoi messaggi senza interruzione alcuna;

3. Ogni messaggio ricevuto dal modem può essere schematizzato come un comando (o un insieme di comandi) con un parametro annesso (a volte il parametro può non essere presente). Ciascun messaggio termina con un carattere new-line '\n': questa informazione si è rivelata molto utile

al fine di scrivere il codice che emula il carico programmabile, dato che consente di individuare e separare tra loro le singole richieste inviate dal modem.

Considerando che i comandi possono avere una lunghezza arbitraria e non conoscibile a priori, il carico programmabile virtuale non può leggere dalla porta seriale una quantità prefissata di caratteri sicuro di ottenere così un nuovo messaggio. Per risolvere questo problema, la funzione **read()** legge un solo carattere alla volta, salvando i caratteri letti in un array temporaneo. Ogni volta che viene letto il carattere '\n', un nuovo messaggio è stato ricevuto. Quest'ultimo viene analizzato, identificando il comando e il parametro (se presente) che il modem ha inviato al carico;

4. Ogni messaggio ricevuto corrisponde ad un'azione che il modem comanda al carico di eseguire. Ad esempio possono essere specificate delle particolari modalità di funzionamento, dei valori per la potenza, o ancora l'attivazione/disattivazione del carico stesso. Una volta analizzato un messaggio quindi, il carico virtuale imposta i valori delle variabili che rappresentano caratteristiche del carico programmabile reale in base a quanto comunicato dallo stesso modem nei parametri dei comandi.

Il funzionamento elencato nei punti 3 e 4 viene ripetuto in un ciclo *while*. Come avviene per il nemo virtuale, anche il carico programmabile utilizza un contatore degli errori in lettura dalla porta seriale per controllarne lo stato. Se infatti, la funzione **read()** restituisce -1 come valore di output, il contatore viene incrementato segnalando che nessun nuovo carattere è stato ricevuto. Quando gli errori in lettura superano la soglia massima prefissata, il thread del carico programmabile interrompe la comunicazione con il modem PLC e termina la sua esecuzione. Se invece è possibile leggere una nuovo carattere, il contatore viene riazzerato; questo indica che la comunicazione con il modem PLC è operativa.

3.4.5 Schema riassuntivo

In Figura 3.5 è presente uno schema che descrive le azioni principali svolte dal carico programmabile virtuale durante la sua esecuzione.

3.4.6 Lo schema del generatore di potenza

Si è detto che durante la fase di inizializzazione, il modem PLC comunica al carico virtuale, oltre alle modalità operative da impostare, i valori fissati

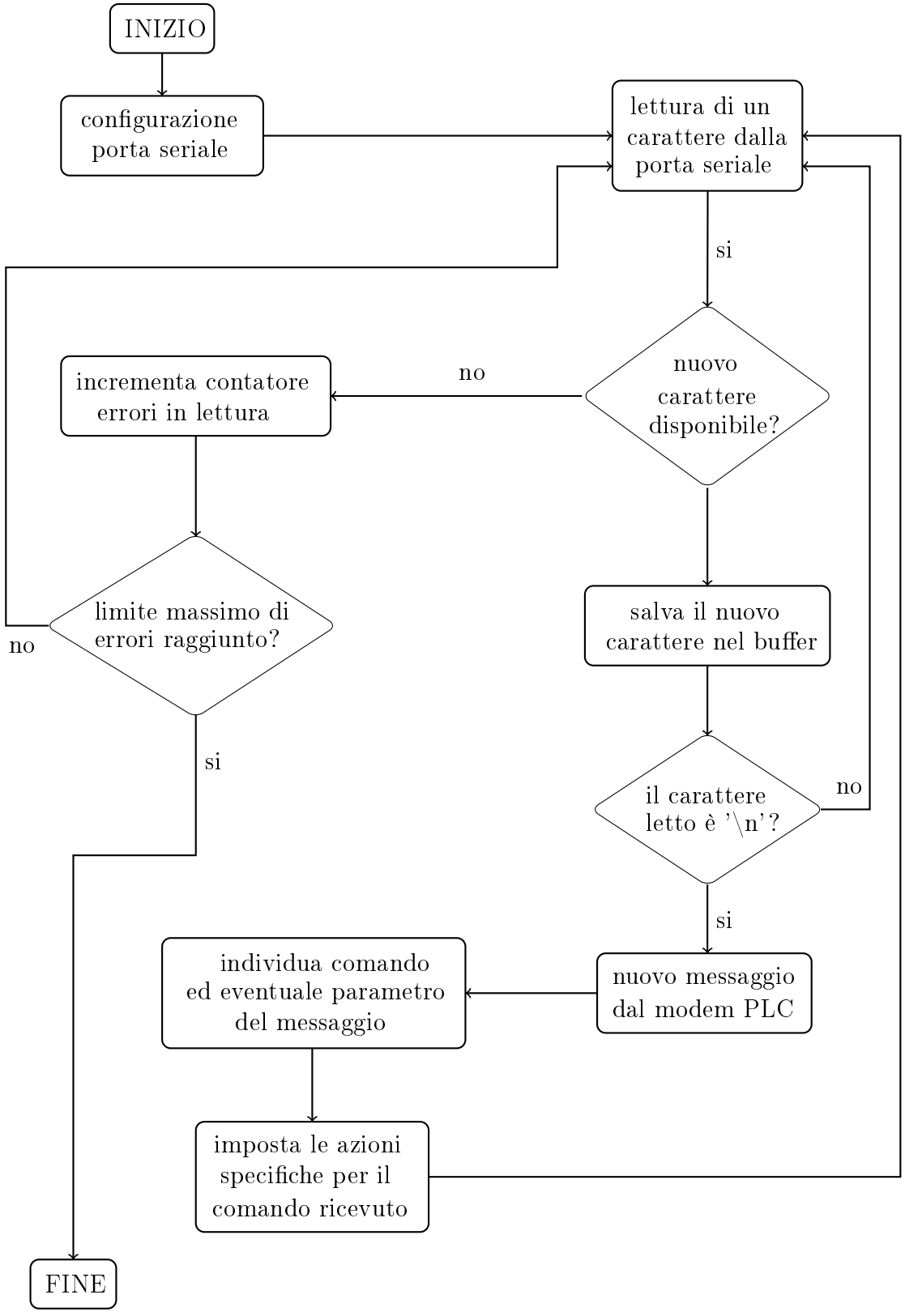

Figura 3.5: Schema di funzionamento del carico programmabile virtuale

di default per potenza attiva, power factor e crest factor. Questi valori sono rispettivamente 100 W, 1.00 e 1.414. Questa situazione corrisponde alla simulazione di un resistore, che quindi non ha alcuna influenza sulla potenza reattiva della rete elettrica. In seguito, il modem master associato al carico interroga il modem connesso invece al Nemo virtuale, da cui riceve lo *status vector* contenente i valori di tensione, corrente, potenza attiva e potenza reattiva relativi ad una sottorete di carico presente nella rete elettrica virtuale. Usufruendo di tali valori per la potenza attiva e la potenza reattiva, il modem master calcola i nuovi valori per potenza attiva, power factor e crest factor che dovrà successivamente inviare al carico virtuale. Quando questi nuovi valori vengono trasmessi al carico, quest'ultimo provvede ad aggiornare i valori del generatore di potenza a lui associato.

Proprio in merito alla comunicazione dei nuovi dati dal modem PLC al carico programmabile, è necessario spiegare un dettaglio del codice contenuto nella Read Only Memory (ROM) del modem PLC master. Si è detto infatti, nel precedente capitolo, che quando nella rete sono presenti elementi reattivi, al carico programmabile è richiesto di impostare un fattore di potenza pari a 0.95. Inoltre, il valore del power factor è accompagnato dal segno + o - a seconda del valore della potenza reattiva che deve essere compensata.

In realtà però, nel codice del master modem il power factor utilizzato per i calcoli della potenza attiva non è 0.95, ma 0.97. Questo scostamento è dovuto all'incapacità dello strumento Chroma 63804 di mantenere un valore del power factor vicino a quanto richiesto: si è infatti notato in test effettuati in laboratorio, che quando al carico programmabile reale veniva specificato un fattore di potenza di 0.95, esso forniva un power factor effettivo di 0.97. A questo punto, in presenza di impedenze a parte immaginaria non nulla nella rete, il modem comanda sempre di impostare per PF un valore, al netto del segno, di 0.95, così come il crest factor rimane fisso a 1.55. Per trovare il corrispondente valore della potenza attiva, considerato il comportamento non ideale dello strumento Chroma 63804, il PF usato è 0.97, valore simile a quanto effettivamente impiegato dal dispositivo.

Durante la stesura del codice per realizzare il carico programmabile virtuale, si è deciso di utilizzare, nei casi in cui la rete elettrica comprenda anche componenti reattivi, un power factor pari a 0.95, come originariamente stabilito. Il carico virtuale ovviamente non simula alcuna fluttuazione del valore del fattore di potenza, mantenedolo fisso a quanto specificato. In considerazione del fatto che il modem PLC continua a comunicare il valore per la potenza attiva calcolato utilizzando un power factor di 0.97, è necessario operare una correzione del dato della potenza attiva ricevuto, in modo da rapportarlo al valore di 0.95 per PF. Il modem, per trovare il valore di P da

trasmettere al carico programmabile, esegue la seguente operazione:

$$P = Q \frac{PF}{\sqrt{1 - PF^2}}$$

Dato che il valore del power factor è stabilito, a meno del segno, la frazione presente nella precedente formula (che moltiplica Q) può essere calcolata in modo da semplificarla. Si ha rispettivamente, che per $PF = 0.97$ la frazione ha un valore pari a 3.99, mentre per $PF = 0.95$ questo diventa di 3.042. Il modem, comunica quindi al carico programmabile di impostare una potenza attiva pari a $|Q|$ * 3.99, dove Q indica la potenza reattiva che il modem legge dallo *status vector*. Il segno della potenza reattiva come detto viene trasmesso attraverso quello del power factor, ovviamente girandolo: così, se Q è positiva, PF sarà negativo e di conseguenza se Q è negativa, PF sarà positivo.

Quando il carico programmabile virtuale riceve dal modem PLC master il valore della potenza attiva, provvede subito ad effetture la correzione del dato, per riporarlo ad un PF di 0.95. La formula utilizzata per eseguire questa correzione è la seguente:

$$P_c = P_m \frac{3.042}{3.99}$$

Nella formula, P_m indica il valore della potenza attiva ricevuto dal modem, mentre P_c il dato ottenuto in seguito all'applicazione della correzione. A questo punto, la potenza attiva P viene impostata al valore P_c appena trovato. Quando tutti i dati sono stati ricevuti e sistemati, nota la potenza attiva, si può procedere al calcolo della potenza reattiva corrispondente. Questo consentirà successivamente di ottenere la potenza complessa e di calcolare quindi i parametri del generatore di potenza con cui si è deciso di rappresentare il carico programmabile virtuale all'interno della rete elettrica virtuale. Per trovare Q, si è fatto uso della seguente formula:

$$Q = P \frac{\sqrt{1 - PF^2}}{PF}$$

Il segno di Q viene determinato tramite quello del power factor. Una volta note la potenza attiva e quella reattiva, la potenza complessa si può ottenere come: $S = P + jQ$. La potenza complessa si può anche calcolare come $S = \bar{V}\,\bar{I}^*$, dove $\bar{I}^*$ indica il coniugato della corrente. In questo caso, $\bar{V}$ e $\bar{I}$ indicano i fasori della tensione e della corrente rispettivamente. Dato che il generatore di potenza con cui il carico programmabile viene schematizzato nella rete è un parallelo tra un generatore di corrente ed una impedenza, la tensione è la stessa per i due bipoli, mentre le correnti sono differenti.

All'interno del simulatore, si è stabilito di inserire lo schema del generatore di potenza con una connessione in parallelo, pertanto in questo caso, essendo nota la tensione $\bar{V}_{pl}$, la corrente del generatore di potenza si può calcolare sommando la corrente del generatore di corrente e la corrente che scorre nel ramo dell'impedenza, rispettando i riferimenti posti. Si ha quindi:

$$\bar{I}_{pl} = \bar{I}_{gen} - \bar{I}_{imp}.$$

Nella formula $\bar{I}_{gen}$ è il fasore della corrente del generatore di corrente, mentre $\bar{I}_{imp}$ è il fasore della corrente dell'impedenza. L'impedenza, identificata come $\dot{Z}_{pl}$, rimane solitamente fissa ad un valore stabilito al momento della creazione del thread che rappresenta il carico programmabile, quindi la corrente che scorre nel ramo dell'impedenza si può sempre calcolare in questo modo:

$$\bar{I}_{imp} = \frac{\bar{V}_{pl}}{\dot{Z}_{pl}}$$

Invece, per conoscere la corrente del generatore, a partire dalla formula $S_{pl} = \bar{V}_{pl}\,\bar{I}^*_{pl}$ e sostituendo $\bar{I}_{pl} = \bar{I}_{gen} - \bar{I}_{imp}$ si ha:

$$\bar{I}_{gen} = \left(\frac{S_{pl}}{\bar{V}_{pl}}\right)^* - \frac{\bar{V}_{pl}}{\dot{Z}_{pl}}$$

Le operazioni descritte in questa sezione sono contenute all'interno del codice C di una funzione denominata **adjustPowerFactorCompensation()**: per prima cosa, essa corregge il valore della potenza attiva P nel caso il power factor abbia un valore (in modulo) di ± 0.95; viene poi calcolata la potenza reattiva basandosi sui dati ricevuti per P e PF. La modifica del fasore della corrente impressa del generatore di corrente viene invece eseguita dal thread gestore della rete elettrica virtuale.

3.5 Test dei dispositivi virtuali

Al termine della stesura del codice C che realizza le due tipologie di dispositivi virtuali descritte qui sopra, sono stati eseguiti alcuni test per verificare la correttezza delle comunicazioni con i modem PLC connessi tramite porta seriale all'elaboratore e la corretta gestione dei messaggi di risposta che devono essere inviati o delle azioni da intraprendere da parte dei dispositivi virtuali. In particolare, in laboratorio sono state testate delle configurazioni miste tra varie tipologie di dispositivi. Esse comprendevano:

- una prima rete, formata da UPS e un Nemo D4-L+ (reali), i due modem PLC in configurazione master-slave, alcuni componenti elettrici sia resistivi che reattivi per creare una sottorete di carico monitorata dal Nemo ed infine il carico programmabile virtuale eseguito come thread nell'elaboratore. Quest'ultimo, al variare della potenza attiva e reattiva della rete elettrica, doveva essere in grado di visualizzare e rispondere correttamente alle richieste provenienti dal master modem prestando particolare attenzione ai valori comunicati per la potenza attiva, power factor e crest factor;

- una seconda rete, comprendente un Nemo virtuale, i due modem PLC in configurazione master-slave e il carico programmabile Chroma Load 63804 (reale): in questo caso il Nemo virtuale doveva essere in grado di gestire correttamente la comunicazione con il modem slave a lui connesso, inviando messaggi di risposta alle richieste per la tensione, la corrente, la potenza attiva e la potenza reattiva rilevate nella rete dove il Nemo virtuale opera. Per semplicità, sono stati utilizzati dei valori fissi per tensioni correnti e potenze, verificando successivamente tramite il carico programmabile che potenza attiva, power factor e crest factor rappresentassero una rete adinamica o una rete con elementi reattivi.

Capitolo 4

Gestione della rete virtuale

Nelle pagine precedenti sono stati introdotti ed analizzati i dispositivi virtuali, Nemo e carico programmabile, realizzati simulando le funzioni ed il comportamento degli strumenti reali. Nella struttura complessiva del software C che è stato scritto, anche per la rete elettrica è stata creata una versione virtuale, costituita da componenti come generatori, impedenze, ecc.. e gestita da un thread apposito, chiamato gestore della rete o network manager. Questo thread non viene creato in più istanze a differenza di quelli che rappresentano i dispositivi virtuali; il suo scopo è quello di interagire con la rete, risolvendola e aggiornando se necessario, i valori di tensioni, correnti e potenze dei componenti elettrici associati ai dispositivi virtuali. Si è già accennato al fatto che il Nemo monitora gli andamenti delle grandezze elettriche in una sottorete di carico mentre invece, il carico programmabile all'interno della rete viene sostituito da un generatore di potenza.

Considerando che il network manager assolve la principale funzione di aggiornamento della rete elettrica, l'insieme dei componenti elettrici associati ai thread dei dispositivi virtuali deve essere, diciamo condiviso con il thread gestore della rete. Per fare questo, è stata creata una struttura dati, definita *electricData*, dove sono proprio raccolte informazioni quali tensione, corrente, potenza attiva, potenza reattiva di un certo insieme di componenti elettrici, oltre che valori di impedenza e/o tensioni/correnti impresse nel caso si tratti di un generatore. Ogni dispositivo virtuale, nemo o carico programmabile, possiede un riferimento ad una struttura di tipo *electricData* che costituisce quindi, l'insieme dei componenti elettrici su cui il dispositivo virtuale opera. Nel caso del Nemo virtuale, le operazioni svolte saranno solamente di lettura dei valori di tensione, corrente, potenza attiva e reattiva (che sono i campi della struttura *electricData*), necessarie al dispositivo per l'eventuale comunicazione con un modem PLC: il Nemo non modifica in alcun modo questi valori, compito che spetta al thread gestore della rete. Situazione

differente invece per il carico programmabile virtuale, schematizzato come un generatore di potenza, ossia un parallelo tra un generatore di corrente e un'impedenza. Quando il carico riceve dalla porta seriale i dati relativi a potenza attiva, power factor e crest factor che il modem comanda di impostare, esso ricava la potenza reattiva, poi la potenza complessa ed infine il corrispondente valore della corrente impressa del generatore di corrente. A questo punto provvede ad aggiornare i campi della struttura *electricData* ad esso associata che saranno utilizzati dal gestore della rete elettrica virtuale al momento della sua risoluzione.

Da quanto è appena stato detto, si può evidenziare la presenza di una possibile interferenza tra il thread gestore della rete e un thread che rappresenta un dispositivo virtuale, in quanto entrambi potrebbero accedere contemporaneamente ad una stessa struttura *electricData* per leggere e/o modificare alcuni valori. Per risolvere questo inconveniente, ogni Nemo o carico programmabile virtuale possiede un semaforo di mutua esclusione (riferimento cap. 3) che utilizza come meccanismo di sincronizzazione quando necessita di accedere alla struttura *electricData* a lui associata. Anche il gestore della rete, prima di accedere ad una determinata struttura di questo tipo, verifica la disponibilità del semaforo del dispositivo virtuale, e nel caso esso abbia un valore nullo, si blocca e attende. Va detto che le fasi di attesa sono molto brevi, dato che le operazioni di aggiornamento non vanno oltre la lettura o la modifica di una manciata di dati. Non sussistono quindi problematiche di rallentamento dell'esecuzione complessiva del codice da parte dell'elaboratore. Un altro dettaglio importante, il network manager condivide con gli altri threads solamente la struttura *electricData*, le restanti azioni che essi svolgono sono indipendenti le une dalle altre.

4.1 Visualizzazione dei dati

Gli strumenti presenti in laboratorio, il Nemo D4-L+ e il Chroma 63804 sono dotati ciascuno di un display lcd che permette la visualizzazione delle grandezze elettriche misurate nella rete, selezionabili da un elenco che comprende un buon numero di opzioni. Finora si è parlato solamente degli aspetti inerenti al funzionamento delle versioni software di tali dispositivi, senza considerare come rendere possibile appunto la visualizzazione dei dati su cui essi operano. Mentre però con gli strumenti reali si può decidere quali informazioni si vogliono misurare e mostrare a video, per i dispositivi virtuali è si è preferito visualizzare solamente i dati più rilevanti, che sono:

1. tensione, misurata in [V];

2. corrente, misurata in [A];

3. sfasamento tra tensione e corrente, misurato in radianti;

4. potenza attiva, misurata in [W];

5. potenza reattiva, misurata in [VAR].

Queste informazioni vengono riportate in un pannello grafico, che non è altro se non una semplice struttura dove i nomi delle grandezze compaiono nello stesso ordine in cui sono presenti nell'elenco qui sopra, affiancate ciascuna dal proprio valore e relativa unità di misura. Ogni pannello possiede un identificatore che consente di distinguere a quale dispositivo virtuale tali informazioni sono collegate. Il pannello grafico viene associato a ciascun thread creato, sia che esso rappresenti un Nemo virtuale o un carico programmabile virtuale: i dati visualizzati saranno quindi gli stessi.

4.1.1 Librerie grafiche Gtk+

In considerazione dell'ambiente di programmazione utilizzato, ossia il sistema operativo Ubuntu e il linguaggio C, si è scelto di utilizzare l'API Gtk+ [6] (chiamata anche GIMP Toolkit), uno strumento molto potente e versatile per la creazione di interfacce grafiche. Gtk+ è un insieme di librerie dotato di numerose tipologie di *widgets* (come finestre, pulsanti, contenitori di testo editabili, tool per la creazione di menù, ecc...) configurabili, sistemi per realizzare differenti layouts con cui creare interfacce, strumenti per la gestione di eventi e segnali che consentono di controllare l'interazione dell'utente.

Quando si vuole creare un'interfaccia grafica con gtk+, per prima cosa è necessaria l'inclusione nel codice della libreria gtk+ e l'inizializzazione dell'ambiente gtk, che si può ottenere con la chiamata alla funzione **gtk_init()**. Questo è un passaggio fondamentale per evitare errori in fase di esecuzione.

Dopo aver effettuato tali operazioni, si procedere alla creazione di tutti i widgets, i blocchi fondamentali da cui l'interfaccia è formata. Questi widgets saranno configurati in base alla loro tipologia; inoltre sarà possibile (per quelli che lo prevedono) connetterli a particolari eventi/azioni che si vuole far accadere ad esempio, quando un pulsante viene premuto dall'utente. I widgets saranno disposti all'interno della finestra del programma principale, secondo un layout creato dallo stesso programmatore sfruttando strumenti di gtk+ che consentono di "impacchettare" tra loro i widgets in differenti modalità. Si possono ad esempio utilizzare tabelle, allineare widgets in orizzontale/verticale, costruire menù ecc...

Una volta completato il codice per l'interfaccia grafica, viene aggiunta come ultima istruzione la chiamata alla funzione **gtk_main()**, che manda in esecuzione il programma creato visualizzando la finestra principale. A questo punto, il programma attende il verificarsi di determinati eventi come possono essere l'inserimento di testo in apposite caselle, il click del mouse su un bottone, un menù o un particolare widget, o semplicemente il pulsante di chiusura del programma stesso.

E' bene precisare però che la funzione **gtk_main()** è una chiamata di tipo bloccante, ossia il terminale da cui viene mandato in esecuzione il programma rimane sospeso fino a quando la finestra, a cui si era passato il controllo in precedenza, non viene chiusa. Questa caratteristica della libreria gtk (e di molte altre librerie per la creazione di interfacce grafiche), si scontra con la possibilità di utilizzare, ad esempio, alcune funzioni del programma che si è creato, che potremmo definire di background, che necessitano di essere continuamente in esecuzione per far funzionare in modo corretto il programma stesso. E' possibile però ricorrere ad alcuni strumenti, previsti da gtk, come i temporizzatori, che consentono di interrompere il main principale ad intervalli prestabiliti per eseguire la funzione o le funzioni desiderate e restituire al loro completamento il controllo alla finestra del programma.

4.2 Server Grafico

Il quadro presentato finora consente di ipotizzare differenti modalità con cui possono essere visualizzate le informazioni dei vari dispositivi della rete virtuale. La soluzione che si è scelto di adottare è in realtà più complessa della semplice visualizzazione dei dati in un'interfaccia, dato che si è deciso di separare la parte del software che realizza la rete virtuale e i threads ad essa legati dal software che implementa invece la parte grafica dove vengono mostrate le tensioni, le correnti e le potenze della rete. Il thread che gestisce la rete virtuale si occuperà allora anche di trasferire in qualche modo al software che gestisce l'interfaccia grafica i dati relativi ai dispositivi virtuali. La decisione di creare due differenti programmi, uno per la rete elettrica e uno per la grafica, ha una duplice motivazione:

- per prima cosa, i threads che vengono creati per sostituire i monitor di rete ed i carichi programmabili, necessitano di gestire ciascuno la comunicazione con la porta seriale. Non per tutti i Nemo o carichi adattativi virtuali in realtà lo scambio di informazioni è previsto ma comunque per un buon numero di essi ciò è vero. Gtk, consente come detto, di interrompere in vari modi il main principale, eseguendo le funzioni di cui si necessita ad intervalli prestabiliti. Sono inoltre presenti funzioni

per il controllo dell'I/O e la possibiltà di utilizzare anche un particolare tipo di thread, proprio di gtk, denominato *gthread* per la realizzazione di programmi multithreading. Perché allora eseguire l'interfaccia in un programma separato?

I threads che costituiscono i dispositivi virtuali utilizzano effettivamente un numero di operazioni limitato e di bassa complessità, ma necessitano di eseguirle ciclicamente e senza interruzioni. In particolare, il Nemo viene continuamente interrogato dal modem PLC per conoscere lo *status vector* (formato dai valori di tensione, corrente, potenza attiva e potenza reattiva), e deve quindi provvedere ad inviare risposte opportune tramite la porta seriale in tempi ragionevoli.

Oltre a questo, la presenza di un numero consistente di thread può produrre un overhead di sistema relativamente consistente. Per consentire quindi di gestire al meglio lo scambio di informazioni tra i dispositivi virtuali e i modem PLC, oltre che per controllare in maniera più rigida le azioni svolte all'interno della rete virtuale, si è preferito realizzare in un software separato la visualizzazione dei dati.

- un ulteriore motivazione consiste nel fatto che sarebbe desiderabile, poter disporre di un insieme di dati e informazioni relative a queste reti elettriche virtuali, che includono i risultati di test eseguiti in laboratorio. Le informazioni dovrebbero essere salvate in un server *http* dedicato e dovrebbero essere consultabili utilizzando i più comuni browser web da un qualsiasi terminale;

4.2.1 Socket

Rimane ora da capire come realizzare il software per la visualizzazione dei dati, anche perché, essendo separato dal software dove è stata creata la rete virtuale, sarà eseguito su un processo differente dell'elaboratore. Considerato questo particolare allora, è necessario disporre di uno strumento con cui è possibile far dialogare tra loro due processi distinti eseguiti all'interno o della stessa macchina o di macchine remote tra loro.

Questo strumento è nuovamente messo a disposizione dall'ambiente unix e passa sotto il nome di *socket*. Il socket consente di realizzare uno scambio di informazioni bidirezionale tra processi, sia che essi si trovino sullo stesso host, sia che essi siano in esecuzione su host diversi. Nel primo caso, si parla di *unix domain socket* (o socket di dominio unix), e si intende un canale di comunicazione interprocesso basato sulla condivisione di dati all'interno del file system. Nel caso in cui i processi siano presenti su macchine diverse,

si parla di *network socket*, e il flusso di dati tra i due processi sfrutta una rete basata su un determinato protocollo di rete. L'API per creazione di un socket non è molto differente nei due casi, nel senso che buona parte delle funzioni a disposizione sono le stesse. Le principali differenze tra socket di rete e socket di dominio locale sono invece le seguenti:

- i network socket sfruttano un protocollo di rete per le comunicazioni (di solito IP, in questo caso possono essere chiamati anche internet socket) mentre i socket di unix basano il loro scambio di dati sul file system;
- le informazioni relative ad un socket sono raccolte in una struttura dati, che per socket di rete che usano IP (i più usati) è di tipo *sockaddr_in* mentre per i socket di dominio locale è di tipo *sockaddr_un*. Le strutture differiscono nei campi che esse contengono al loro interno;
- la creazione di un socket avviene chiamando la system call **int socket(int domain, int type, int protocol)** dove:
  - *domain* specifica la famiglia di socket che si vuole creare. Ad esempio, per un internet socket, *domain* è solitamente AF_INET mentre per un socket di unix la famiglia è AF_UNIX;
  - l'argomento *type* indica invece il tipo di socket: si può scegliere tra SOCK_STREAM, che crea un socket la cui comunicazione è gestita come un flusso di dati (come per il protocollo TCP) o SOCK_DGRAM dove lo scambio di informazioni è realizzato tramite datagrammi (come il protocollo UDP). Questi due tipologie di socket sono disponibili sia per socket di rete che di dominio locale;
  - *protocol*, che indica appunto un protocollo, è un valore intero che va specificato in base al tipo di socket che si vuole creare;
  - il valore di ritorno di tipo *int* è il file descriptor collegato al socket.
- l'indirizzo associato ad un socket (contenuto nella struttura dati di tipo *sockaddr*) è formato, per i socket di rete da una coppia indirizzo IP - porta della macchina host mentre per i socket di unix è semplicemente il percorso (path) del file descriptor collegato al socket nel sistema.

I socket sono basati sull'architettura di tipo client/server, nella quale due processi, i due interlocutori della comunicazione, scambiano informazioni tra loro. Uno dei due processi, il server assume il ruolo di fornitore di un servizi, che mette a disposizione ad un altro processo, il client che agisce invece come un richiedente di servizio. Sia che si voglia creare un socket di rete che uno di

dominio locale, i passi da compiere per ciascun processo per la creazione del socket sono esattamente gli stessi. Ciascun processo crea il proprio socket, che costituisce una delle due parti del canale di comunicazione che si creerà quando la connessione sarà avvenuta. Le operazioni che consentono di stabilire questa connessione sono differenti per il client ed il server. Vediamo di analizzarle.

I passi con cui il client crea il proprio socket sono i seguenti:

1. crea il socket per la comunicazione con il server tramite la system call **socket()**, specificandone i parametri di ingresso;

2. utilizzando la system call **connect()**, il client connette il proprio socket all'indirizzo del server, che deve necessariamente conoscere. Importante nota, **connect()** restituirà un errore nel caso il server non sia ancora attivo;

3. quando la connessione è stata correttamente stabilita, il client può scambiare dati con il server. Per fare ciò, molte opzioni sono possibili, tra cui ad esempio le system calls **read()** e **write()**, già viste in quanto impiegate per leggere/scrivere dati dalla porta seriale, **send()** e **recv()**, ecc...

4. al termine delle comunicazioni con il server, il client chiude il socket con la system call **close()**.

Il server invece svolge operazioni un pò più complicate:

1. crea il socket per la comunicazione con il client con la system call **socket()**, specificandone i parametri di ingresso;

2. associa al file descriptor del socket appena creato un indirizzo nel dominio unix con la system call **bind()**;

3. tramite la chiamata a **listen()**, il server si mette in ascolto di richieste di connessione da parte di un client;

4. il server, con la system call **accept()**, accetta le connessioni provenienti dal client. La funzione **accept()** restituisce un ulteriore file descriptor, connesso appunto al client di cui è stata accettata la richiesta, mentre il file descriptor originariamente creato rimane in ascolto di altre richieste di connessione;

5. scambia le informazioni con il client tramite le funzioni **send()** e **recv()**, che costituiscono la scelta più frequente;

6. termina la connessione con il client chiudendo il socket con la system call **close()**.

4.2.2 Protocollo di comunicazione client-server

Ritornando all'analisi del software della rete elettrica virtuale, ora grazie ai socket, sappiamo come far comunicare tra loro due processi distinti. Si è detto che si vuole separare il codice che permette la visualizzazione dei dati della rete da quello che realizza la rete stessa; questo implica che al momento dell'esecuzione dei due programmi, verranno creati due processi separati.

Dato che i socket si basano sul modello client/server, è necessario stabilire quale delle due parti software assumerà il ruolo di client e quale il ruolo di server. Si è scelto di far operare il software per la visualizzazione dei dati come server, che d'ora in poi chiameremo server grafico. A questo punto allora, il network manager sarà il client. Parliamo di network manager in quanto il socket per la comunicazione con il server grafico verrà creato dal thread gestore della rete, che come detto tra le funzioni da svolgere ha anche quella di trasferire i dati ad una entità esterna in modo da poterli mostare all'utente.

Questa decisione può sembrare in realtà un pò innaturale, in quanto solitamente, quando un client invia ad un server una richiesta, si aspetta di ricevere i risultati in un messaggio di risposta. Qui invece i risultati, che sono le tensioni, correnti e potenze della rete elettrica virtuale, sono posseduti dallo stesso client. In pratica il server agisce semplicemente come un immagazzinatore di informazioni; ricevendo periodicamente messaggi di aggiornamento dal client che provvederà a salvare, sovrascrivendo i dati precedentemente acquisiti e visualizzando i nuovi valori tramite l'interfaccia grafica realizzata con le funzionalità dell'API Gtk+.

Per gestire opportunamente la comunicazione tra network manager e server grafico, è stato ideato un protocollo di comunicazione apposito. Quando il gestore della rete vuole comunicare un nuovo dato al server, fa precedere il messaggio che conterrà le nuove informazioni da un messaggio di richiesta, contraddistinto da un particolare codice identificativo.

Il codice contenuto nel messaggio di richiesta, consente al server di distinguere quale tipologia di informazioni il client intenda inviargli, così da configurare in modo ottimale i parametri della system call **recv()** con cui i dati in arrivo vengono ricevuti dal server. I codici utilizzati hanno tutti una lunghezza fissa di 14 caratteri, nuovamente questo permette di specificare con precisione gli argomenti delle funzioni **send()** e **recv()**, che in questo modo sono noti ad entrambi gli interlocutori. Il meccanismo appena descritto si è rivelato molto utile per gestire la comunicazione tra network manager e ser-

ver, anche perché permette al gestore della rete di inviare al server quantità di informazioni differenti sia come dimensione che come tipologia e al server, grazie al codice di richiesta proveniente dal client, di prepararsi a ricevere un certo volume di dati che saprà poi anche come elaborare.

Oltre a quanto detto finora, client e server inoltre condividono l'utilizzo di una struttura dati di supporto, chiamata *communicationData*, dove sono contenute le informazioni quali:

- tensione;
- corrente;
- sfasamento tensione-corrente;
- potenza attiva;
- potenza reattiva;
- identificatore di un dispositivo virtuale a cui le precedenti informazioni sono associate; l'identificatore è formato da tipologia e numero del dispositivo, ed è esattamente lo stesso che si può trovare nei campi delle strutture *virtualNemoData* e *virtualProgrammableLoadData* di cui si è parlato nel precedente capitolo dedicato alla descrizione dei dispositivi virtuali;

Per precisare, l'identificatore che compare all'interno della struttura communicationData è suddiviso nei due campi *deviceType* e *deviceNumber*: il primo, formato da quattro caratteri, indica il tipo di dispositivo, e può assumere i valori "nemo" o "load", che indicano rispettivamente o un nemo virtuale o un carico programmabile virtuale; il secondo campo invece è formato da due soli caratteri e indica il numero del dispositivo all'interno della rete. I numeri sono assegnati in ordine crescente e consecutivo.

Una piccola nota per il campo *deviceNumber*: esso è costituito come detto da due caratteri e per i numeri da 1 a 9 il formato utilizzato prevede di antepporre al numero la cifra 0 quindi la rappresentazione del numero 3 diventa 03.

Ad ogni dispositivo virtuale quindi, è associata un ulteriore struttura di tipo *communicationData* che viene aggiornata dal network manager quando risolve la rete elettrica virtuale. Per uno specifico dispositivo virtuale, i valori che il gestore della rete inserirà nei campi delle grandezze elettriche della struttura *communicationData* saranno esattamente gli stessi della struttura *electricData* associata a quel dispositivo. Il server grafico si serve invece della

struttura *communicationData* per salvare le informazioni ricevute dal client per un particolare dispositivo virtuale.

Presentiamo di seguito i codici specifici del protocollo di scambio dati tra client e server:

- *requestInitCm* e *requestStopCm* sono rispettivamente i messaggi che contraddistinguono l'inizio e la fine di una comunicazione. Lo scambio di dati tra il client ed il server può essere infatti pensato come un insieme di comunicazioni distinte, ciascuna di durata variabile e non prestabilita. Per entrambi i messaggi, il network manager si aspetta di ricevere dal server una risposta, formata da un solo carattere. Essa può essere 'y' o 'n' (che hanno il significato di yes e no), a seconda che il server accetti la richiesta o la rifiuti;

- *requestActDev* è il messaggio che il gestore della rete invia al server quando vuole informarlo circa il numero di dispositivi virtuali operativi sulla rete elettrica. Anche per questo messaggio, il client si aspetta di ricevere una risposta dal server di un solo carattere, con lo stesso significato visto prima. Dopo aver accettato la richiesta, il server si prepara alla ricezione di un valore integer che conterrà appunto il numero di dispositivi attivi. Questo dato è molto importante in quanto consente al server di configurare l'interfaccia grafica predisponendo un adeguato numero di pannelli grafici, sulla base delle informazioni provenienti dal network manager;

- *requestDataDv* è il messaggio con cui il network manager informa il server dell'imminente comunicazione di un array di caratteri, in cui sono presenti, tipologia e numero corrispondente di ciascun dispositivo virtuale della rete. Il server, dopo aver accettato la richiesta, si prepara a ricevere i dati in un array correttamente dimensionato in base al numero di dispositivi presenti nella rete;

- *requestTypeNm* è la richiesta (il cui nome è parametrico) che il thread gestore della rete utilizza quando intende comunicare al server la struttura dati *communicationData* associata ad un determinato dispositivo virtuale, contenente i valori aggiornati delle grandezze elettriche di un certo componente della rete. La richiesta è configurabile specificando i campi *Type* e *Nm* che costituiscono rispettivamente il tipo ed il numero del dispositivo. In seguito all'accettazione della richiesta inviata dal client, il server salva la struttura *comunicationData* ricevuta in un apposito array dove sono mantenute anche tutte le altre informazioni relative ai dispositivi virtuali operanti sulla rete;

- *requestNetOff* è il messaggio di richiesta che il gestore della rete utilizza per avvisare il server sullo stato della rete virtuale. Ricevuta la consueta accettazione da parte del server, il network manager invia, tramite un valore integer, lo stato della rete (contenuto in un'apposita variabile): esso è 1 se la rete è attiva mentre è 0 se la rete o uno dei suoi dispositivi si è disattivato. Come suggerisce il nome, il client utilizza questa richiesta solamente per comunicare al server che la rete è stata disattivata e quindi la comunicazione deve essere interrotta. Il server, in seguito alla ricezione di questo messaggio, termina la sua attività.

4.2.3 Funzionamento del Server Grafico

Dopo aver introdotto tutti gli strumenti necessari per gestire la comunicazione tra processi e aver presentato un protocollo per regolare lo scambio dati tra di essi, vediamo come il server è stato creato e quali sono le operazioni che esso compie:

1. per prima cosa, il server provvede a creare il socket per la comunicazione, configurando i parametri della system call **socket()** con opportune opzioni;

2. una volta creato il socket, questo viene associato ad uno specifico indirizzo della macchina. Questo sarà poi utilizzato dal client come argomento per la funzione **connect()**. Dopo aver eseguito questa operazione(con la system call **bind()**) il server si mette in ascolto di una richiesta di connessione proveniente dal network manager, quando questa giunge al server, viene accettata ed in questo modo, la connessione tra i due processi è stabilita;

3. a questo punto, il server attende di ricevere i messaggi di richiesta dal network manager. Essi come detto hanno una lunghezza fissa di 14 caratteri. Ognuno di questi messaggi contiene uno dei codici identificativi precedentemente elencati, una volta che il codice è stato esaminato il server riconosce il tipo di richiesta inviata dal client e si predispone per la ricezione dei nuovi dati. Prima di fare questo però, il server accetta la richiesta del client inviando il carattere 'y' in risposta. Nei casi in cui i messaggi trasmessi dal client siano quelli che segnalano l'inizio o la fine di una comunicazione, il server non si aspetta ovviamente alcun nuovo dato;

4. tramite la system call **recv()**, il server riceve le informazioni tramesse dal network manager. I nuovi dati saranno salvati in apposite aree di

memoria create dallo stesso server. La comunicazione tra i due processi inizia con una prima fase di configurazione, nella quale il client informa il server del numero di dispositivi virtuali attivi sulla rete, oltre alla loro tipologia e relativo numero. Note queste informazioni, il server crea l'interfaccia grafica utente dove verranno visualizzati i dati dei dispositivi virtuali e gli array di strutture di tipo *communicationData* dove saranno contenuti i valori delle grandezze elettriche associate a ciascun dispositivo;

5. il network manager, terminata la fase di configurazione iniziale, invia aggiornamenti periodici sulle quantità da visualizzare al server;

6. nel caso il server riceva dal network manager l'informazione sullo stato della rete, esso provvede alla chiusura del socket creato in precedenza e alla sospensione di tutte le sue attività.

Il server, può decidere di non accettare le richieste che gli giungono dal client, semplicemente inviando il carattere 'n' al posto di 'y' in risposta. Questa eventualità si verifica solamente quando l'utente ha richiesto la conclusione delle attività al server, premendo un apposito pulsante presente nell'interfaccia grafica del server stesso. Se il network manager riceve in risposta il messaggio 'n', riconosce la richiesta del server di sospendere definitivamente la comunicazione, e provvede ad inviare il segnale di terminazione a tutti i thread attivi sulla rete.

Il modello di comunicazione discusso in queste pagine, consente di suddividere l'intera comunicazione tra client e server in comunicazioni minori di durata variabile. Il principale svantaggio del protocollo ideato consiste nel fatto che, se si vuole trasmettere anche un solo dato, è necessario inviare comunque i due messaggi di inizio/fine comunicazione: questo comporta un allungamento dei tempi in cui avviene lo scambio di informazioni. Ovviamente però, così facendo è possibile gestire comunicazioni di lunghezza qualsiasi e non rigidamente prestabilita. I tempi di risposta inoltre, come si è visto da test effettuati, rimangono estremamente rapidi.

4.3 Network Manager

Nelle precedenti sezioni, si è spiegato con quale modalità i dati della rete virtuale vengono visualizzati all'utente. E' stato creato un server grafico, esso riceve costantantemente valori aggiornati dal thread che gestisce la rete, utilizzando una comunicazione su socket con un formato di messaggi studiato appositamente. Si è detto che il network manager ha il principale scopo di

risolvere la rete elettrica virtuale, trovando i valori di tensione e corrente per ciascun lato della rete. Analizziamo ora nel dettaglio come si è scelto di rappresentare la rete elettrica virtuale.

4.3.1 Rete Elettrica Virtuale

All'interno del simulatore costruito in questo progetto, si è deciso di poter configurare reti elettriche non limitate ad uno schema fissato, ma reti lineari qualsiasi operanti in regime sinusoidale. L'ipotesi di linearità della rete è legata all'utilizzo di bipoli lineari, ossia componenti elettrici di cui è accessibile una coppia di morsetti che presenta una relazione tensione-corrente descrivibile tramite un'equazione lineare. All'interno della rete è previsto l'utilizzo sia di componenti adinamici o di ordine zero, ossia con relazioni tensione-corrente che non dipendono dalla dinamica temporale della tensione o della corrente sia di componenti dinamici di ordine uno, che presentano relazioni tensione-corrente che includono legami funzionali (e quindi derivate, integrali, ecc).

Con il termine topologia vengono indicate le modalità di collegamento dei bipoli che costituiscono una rete. Per descrivere la topologia di una rete, è possibile servirsi sia di un grafo, sia di apposite matrici di connessione. Quest'ultime sono particolarmente indicate quando si vuole gestire l'analisi di una rete elettrica tramite un elaboratore, come il caso che stiamo analizzando. Il grafo costituisce però uno strumento di più facile comprensione, esso è costituito da due elementi fondamentali, i nodi ed i lati; agli estremi di ciascun lato è possibile individuare una coppia di nodi. Se per ciascuno dei lati della rete è fissato un verso di percorrenza, il grafo si dirà orientato. Per un qualsiasi grafo orientato è quindi possibile individuare:

- il numero dei nodi, che indichiamo con n, numerati da 1 a n;
- il numero dei lati, che indichiamo con l, numerati con pedice da 1 a l;
- per ciascun lato, la coppia ordinata di nodi su cui il lato si appoggia.

All'interno di un grafo, ciascun lato rappresenta un bipolo della rete.

A partire dal grafo orientato e dal suo numero di nodi e lati, è possibile dedurre una matrice di dimensione n x l, quindi con n righe e l colonne, che consente di associare a ciascun lato la coppia orientata di nodi su cui esso si appoggia. Questa matrice è chiamata *matrice di incidenza*; ciascuna sua riga corrisponde ad un nodo del grafo e ogni sua colonna corrisponde invece ad un lato. L'elemento della matrice nella generica posizione (i, j) vale rispettivamente:

- 1 se il lato l_j ha come nodo di partenza (con orientazione di lato quindi uscente) il nodo i;
- -1 se il lato l_j ha come nodo di terminazione (con orientazione di lato quindi entrante) il nodo i;
- 0 se il lato l_j non si appoggia sul nodo i-esimo.

La righe della matrice di incidenza sono tra loro linearmente dipendenti, pertanto si preferisce considerare la matrice di incidenza ridotta, che indicheremo con **A**, che si ottiene dalla matrice di incidenza eliminando una delle sue righe. Solitamente, la riga da eliminare corrisponde (in numero) al nodo di massa della rete, cioè a quello con potenziale nullo.

La matrice di incidenza ridotta è utilizzata per descrivere la topologia della rete e consente di esprimere in forma matriciale le leggi di Kirchhoff alle tensioni e alle correnti, brevemente indicate con i termini LKT e LKC. Esse consentono di ricavare un sistema di l equazioni topologiche tra loro indipendenti. La soluzione di una rete elettrica consiste però nella determinazione delle l tensioni e correnti di tutti i lati della rete, pertanto, è necessario costruire un sistema di $2l$ equazioni per determinare la soluzione di una qualsiasi rete. Per fare ciò, è possibile utilizzare le relazioni tensione-corrente di ciascun bipolo della rete, che consente di di scrivere un sistema di l equazioni tipologiche.

Riassumendo, il sistema complessivo formato da $2l$ equazioni in $2l$ incognite è il seguente:

$$\begin{cases} \sum \pm \bar{V} = 0 & \text{LKT} \\ \sum \pm \bar{I} = 0 & \text{LKC} \\ \bar{V} - \dot{Z}\bar{I} = -\bar{E} & \text{equazione di lato di tipo GNTS} \\ \bar{I} - \dot{Y}\bar{V} = -\bar{J} & \text{equazione di lato di tipo GNCS} \end{cases}$$

Nel sistema sovrastante, le quantità $\bar{V}$, $\bar{I}$, $\bar{E}$, $\bar{J}$ simboleggiano i fasori di tensioni e correnti di lato ed i fasori delle tensioni e correnti impresse dei generatori ideali rispettivamente, mentre le quantità $\dot{Z}$ e $\dot{Y}$ sono degli operatori complessi. Con il termine GNTS si indica un generatore normale di tensione simbolico, ossia la serie tra un generatore ideale di tensione simbolico (GITS) ed una impedenza espresso utilizzando fasori ed operatori complessi. Dualmente, il termine GNCS indica un generatore normale di corrente simbolico, ossia il parallelo tra un generatore ideale di corrente simbolico (GICS) ed una ammettenza espresso in termini di fasori ed operatori complessi. Queste due equazioni di lato generalizzate, possono essere esplicitate nei seguenti casi

base:

$$\begin{cases} \bar{V} = -\bar{E} & \text{equazione di lato di tipo GITS} \\ \bar{V} - \dot{Z}\bar{I} = 0 & \text{equazione di lato di un bipolo passivo} \\ \bar{I} = -\bar{J} & \text{equazione di lato di tipo GICS} \\ \bar{I} - \dot{Y}\bar{V} = 0 & \text{equazione di lato di un bipolo passivo} \end{cases}$$

Si noti che in tutte le equazioni inserite nei precedenti sistemi, si è fatto uso della convenzione degli utilizzatori per quanto concerne i riferimenti di tensione e corrente dei bipoli, che prevede di considerare come riferimento per la corrente quello entrante nel morsetto contrassegnato con il riferimento di tensione + (o alternativamente con riferimento di corrente uscente dal morsetto con riferimento di tensione -). E' importante che tutti i lati della rete rechino le medesime convenzioni.

E' possibile esprimere in forma matriciale i precedenti sistemi di equazioni. Per quanto riguarda la LKC, si ottiene:

$$AI = 0$$

dove A indica la matrice di incidenza ridotta e I indica invece il vettore colonna dei fasori delle l correnti di lato dei bipoli della rete. Per la LKT si ha invece:

$$-A^{\mathrm{T}}U + V = 0$$

dove A^{T} indica la matrice di incidenza trasposta, U è il vettore dei potenziali nodali, di lunghezza n - *1*, ossia il vettore dei potenziali dei nodi della rete e V è il vettore colonna dei fasori delle l tensioni di lato dei bipoli della rete.

Anche le equazioni tipologiche possono essere espresse in forma matriciale nel seguente modo:

$$YV + ZI = W$$

dove Y è la matrice diagonale di dimensioni l x l i cui coefficienti della diagonale sono i parametri 1 e $-\dot{Y}$ dell'equazione di lato di tipo GNTS e analogamente Z è la matrice diagonale di dimensioni l x l i cui coefficienti della diagonale sono i parametri 1 e $-\dot{Z}$ dell'equazione di lato di tipo GNCS. Il vettore colonna W ha anch'esso lunghezza l ed è il vettore dei fasori delle tensioni e correnti impresse, che sono i parametri $-\bar{E}$ e $-\bar{J}$ dei sistemi descritti precedentemente.

Considerando queste informazioni complessivamente, si può ottenere un nuovo sistema formato da ($2l$ + n - 1) equazioni le cui incognite sono gli l fasori delle tensioni e gli l fasori delle correnti dei lati della rete e gli n - 1 potenziali nodali. Questo sistema si può esprimere utilizzando una formulazione matriciale compatta come riportato di seguito:

$$\begin{bmatrix} 0 & 0 & A \\ -A^{\mathrm{T}} & 1 & 0 \\ 0 & Y & Z \end{bmatrix} \begin{bmatrix} U \\ V \\ I \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ W \end{bmatrix}$$

Per un'esposizione più comprensibile, è possibile definire le matrici nel seguente modo:

$$T = \begin{bmatrix} 0 & 0 & A \\ -A^{\mathrm{T}} & 1 & 0 \\ 0 & Y & Z \end{bmatrix} \quad X = \begin{bmatrix} U \\ V \\ I \end{bmatrix} \quad N = \begin{bmatrix} 0 \\ 0 \\ W \end{bmatrix}$$

La matrice T è chiamata *matrice tableau* ed è una matrice quadrata formata da ($2l + n$ - 1) righe e colonne, il vettore X è il vettore delle incognite del sistema mentre N è il vettore degli ingressi, in quanto comprende i valori dei fasori di tensioni e correnti impresse dei generatori ideali, che nel caso in esame costituiscono appunto gli ingressi del sistema. In sintesi, il sistema si può scrivere come:

$$TX = N$$

Per risolvere il sistema, si è scelto di utilizzare il metodo di eliminazione di Gauss. Questo metodo, di facile implementazione tramite elaboratore, è adeguato per la risoluzione di reti elettriche di ridotte dimensioni, con numero di lati che non supera qualche decina di unità. Per la risoluzione di reti dimensioni più elevate, dove la matrice tableau è formata da centinaia di righe e colonne, è più indicato ricorrere ad altre metodologie di soluzione di sistemi lineari che sfruttano l'elevata sparsità della matrice T. L'algoritmo di Gauss è sensibile alla precisione finita con cui è possibile rappresentare i valori numerici all'interno di un elaboratore; inoltre tali errori divengono più rilevanti all'aumentare del numero di equazioni presenti nel sistema da risolvere: utilizzando il metodo di Gauss in alcuni casi è possibile incorrere in errori nelle soluzioni del sistema, che pur essendo in percentuale ridotti, possono essere notevolmente amplificati quando il valore di una soluzione viene sostituito nelle altre equazioni del sistema. Per cercare di limitare le problematiche appena esposte, all'interno del simulatore è stata implementata una variante dell'algoritmo di Gauss che utilizza la tecnica del *pivoting parziale*. Vediamo brevemente di seguito di cosa si tratta.

Definiamo, per facilità di trattazione, con d la quantità ($2l + n$ - 1), pertanto T è una matrice quadrata d x d. L'algoritmo di Gauss opera sia sulla matrice T sia sul vettore dei termini noti N in due fasi separate: una prima fase di eliminazione, che consente di trasformare la matrice T in una matrice triangolare superiore (detta anche a scalini) ed una seconda fase di sostituzione all'indietro, nella quale, a partire dall'ultima equazione e procedendo

verso l'alto, vengono determinate una alla volta le incognite del sistema. La fase di eliminazione può essere schematizzata nel seguente modo:

1. a partire dalla riga della matrice T con indice k pari a 1, l'algoritmo di Gauss, tramite operazioni di addizione/sottrazione di righe o moltiplicazioni di una riga per uno scalare non nullo, trasforma la matrice T di partenza in una matrice triangolare superiore. Per fare ciò nella colonna di indice k devono essere eliminati tutti i coefficienti della matrice con indice di riga maggiore di k. Ad esempio, quando k = 1, nella colonna k della matrice, ossia nella colonna 1 dovranno essere eliminati tutti i coefficienti a_{i1} con i = 2, ..., d. Prima di fare ciò è però necessario verificare che l'elemento della matrice nella posizione (k, k) sia non nullo, ossia $a_{kk} \neq 0$. In caso contrario è necessario operare uno scambio di righe della matrice;

2. si possono però verificare delle situazioni in cui l'elemento in posizione (k, k) sia si non nullo, ma abbia valore assoluto molto piccolo. Questo solitamente rende l'algoritmo di Gauss instabile, e le soluzioni ottenute presentano un errore percentuale che rispetto a quelle corrette è in certi casi estremamente rilevante. Per cercare di rendere stabile l'algoritmo di Gauss viene quindi introdotta la tecnica del pivoting parziale: essa prevede, che per ogni riga k della matrice, nella posizione di indici (k, k) della matrice sia presente l'elemento a_{rk}, con r > k, che abbia modulo massimo tra tutti i coefficienti della colonna k. Nel caso in cui siano presenti più elementi con lo stesso modulo, viene scelto quello con indice di riga r minore. Le righe di indici r e k vengono a questo punto scambiate, in modo da collocare in posizione (k, k) il termine a_{rk}, che viene chiamato pivot della riga k. Se il coefficiente di modulo massimo si trova già in posizione (k, k) non è necessario alcuno scambio di righe;

3. quando il pivot della riga k-esima è stato individuato, si può procedere alla fase di eliminazione degli elementi della colonna k con indice di riga maggiore di k. A partire pertanto dalla riga i-esima, con i = k + 1, ..., d viene individuato il moltiplicatore della riga i-esima:

$$m_{ik} = \left(\frac{a_{ik}}{a_{kk}}\right)$$

A questo punto, alla riga i-esima viene sottratta la riga k-esima moltiplicata per il moltiplicatore m_{ik}, in questo modo è possibile eliminare l'elemento della matrice in posizione (i, k). Questa operazione è estesa anche al coefficiente del vettore dei termini noti in posizione i, n_i. L'operazione di eliminazione è ripetuta per tutte le righe della matrice di indice i > k;

4. le azioni descritte nei punti 2 e 3 vengono ripetute facendo variare l'indice di riga della matrice k da 1 a d, ottenendo così una matrice di triangolare superiore. Ad ogni iterazione della fase di eliminazione, la matrice considerata avrà dimensioni sempre inferiori, e quando k ha valore d si opererà su di un solo elemento. Nel caso in cui tutti i coefficienti di una colonna della matrice siano nulli, il sistema risulta indeterminato, pertanto esso non verrà risolto.

Terminata la fase di eliminazione, l'algoritmo di Gauss può procedere alla successiva fase di sostituzione all'indietro. A partire dalla riga di indice k = d, vengono determinate le incognite del sistema. L'ultima equazione ha soluzione immediata, mentre per risolvere le altre equazioni è necessario utilizzare i risultati ottenuti dalle precedenti sostituzioni.

La procedura descritta qui sopra consente di ottenere il vettore X come risultato. Da esso si possono ricavare i vettori V e I che contengono tutte le coppie tensione-corrente dei lati della rete elettrica. L'implementazione in linguaggio C dell'algoritmo di Gauss segue gli stessi passi descritti precedentemente ed utilizza una struttura di supporto, denominata *electricNetwork* dove sono salvate tutte le informazioni relative alla rete elettrica che si vuole analizzare, come il numero di nodi e lati, il nodo di massa e contiene appositi puntatori ad aree di memoria dove vengono immagazzinate le matrici A, Y, Z, T ed i vettori V, I, U e N. Inoltre, per descrivere la tipologia di ciascun bipolo della rete, è stata creata una struttura *electronicComponent* che raccoglie informazioni quali il lato della rete dove il bipolo è collocato, i nodi su cui si appoggia, se esso è un generatore o un bipolo passivo e quindi gli eventuali valori di tensioni/correnti impresse o impedenze. All'interno della struttura di tipo *electricNetwork* è contenuto un puntatore ad un array di strutture *electronicComponent* che descrive come detto i bipoli presenti nella rete elettrica in esame, ciascuna di queste strutture viene aggiornata in seguito alla risoluzione della rete elettrica tramite l'algoritmo di Gauss.

4.3.2 Il thread Network Manager

Analizziamo in maniera più specifica le azioni svolte dal gestore della rete virtuale. Il network manager è responsabile della gestione della rete elettrica virtuale, pertanto esso provvederà ad aggiornare la struttura di tipo *electricNetwork* che descrive la rete elettrica stessa ed utilizzerà l'algoritmo di Gauss per la sua soluzione. Oltre a questo, esso provvede ad aggiornare i valori di tensione, corrente, sfasamento, potenza attiva e reattiva per tutte le strutture di tipo *electricData* associate ai dispositivi virtuali che sono stati attivati. Infine, gestisce lo scambio di informazioni con il server a cui comu-

nica i nuovi dati ottenuti al termine della risoluzione della rete. Nel dettaglio il funzionamento può essere riassunto nei seguenti punti:

1. per prima cosa, il gestore della rete crea il socket per la comunicazione con il server grafico. Il socket sarà configurato per operare come un client. Dopo aver creato il socket, il network manager chiama la funzione **connect()** per stabilire la connessione con il server: quest'ultimo deve però essere già stato attivato, altrimenti **connect()** restituirà un -1 e non verrà stabilito alcun collegamento tra i due socket. Se invece la connessione viene correttamente stabilita, il network manager inizia una fase di configurazione in cui comunica al server in un primo messaggio il numero dei dispositivi virtuali attivi nella rete, ed in seguito la loro tipologia ("nemo" o "load") e il loro numero (01, 06, 13 ecc...). Comunque è possibile decidere di inviare al server solo gli identificativi dei dispositivi di cui si vogliono visualizzare le informazioni. Il server come detto, utilizza questi dati per creare l'interfaccia grafica di supporto, che conterrà un numero di "pannelli elettrici" in base a quanti viene richiesto di impiegarne;

2. dopo le aver stabilito la connessione con il server ed aver completato la fase di configurazione, il gestore della rete si occupa del setup della rete virtuale. Il network manager non è responsabile della creazione dei threads dei dispositivi virtuali della rete, queste operazioni sono state svolte precedentemente da una funzione **main()** separata, che provvede a creare tutti i threads previsti compreso lo stesso gestore della rete.

   Dal momento in cui vengono mandati in esecuzione, i due tipi di thread, quello del nemo e del carico programmabile, come primo compito hanno quello di configurare la porta seriale con cui poi scambiare informazioni con i modem PLC. I due strumenti però hanno un comportamento differente nella gestione della comunicazione con il modem: mentre il Nemo riceve sempre quattro richieste distinte di pari lunghezza, il carico programmabile necessita di una fase di setup iniziale prima che il ciclo di comandi per la potenza attiva, il power factor e il crest factor sia attivato.

   Il network manager, prima di iniziare ad operare con la rete elettrica virtuale, si assicura che tutti i dispositivi virtuali siano operativi ed in particolare che abbiano portato a termine con successo la configurazione della porta seriale ed eventuali fasi di setup/attivazione. Per controllare ciò, nelle strutture dati che raccolgono le informazioni di ciascun dispositivo virtuale, è presente un campo *deviceStatus* che

descrive appunto lo stato del dispositivo. Questo viene utilizzato con significato differente a seconda della tipologia del thread:

- il carico programmabile utilizza il valore del campo *deviceStatus* per segnalare al thread gestore della rete che la prima fase della comunicazione, quella di inizializzazione, è terminata e quindi lo strumento è stato attivato (simbolicamente) dal modem PLC con il comando LOAD 1;
- il thread del Nemo virtuale utilizza il valore del campo *deviceStatus* come meccanismo di sospensione: infatti, attende che il thread gestore della rete provveda a modificare il valore di *deviceStatus* portandolo a 1, consentendo così al Nemo di avviare la ricezione delle richieste da parte del modem PLC.

Il network manager quindi, agisce in due momenti distinti:

(a) attende che tutti i carichi programmabili presenti nella rete vengano attivati dai rispettivi modem PLC a cui sono connessi;

(b) a questo punto, vengono sbloccati tutti i Nemo virtuali in attesa del segnale del network manager, che possono quindi iniziare la comunicazione con il modem.

In pratica, tutti i threads di tipo Nemo virtuale, dopo essere stati creati e mandati in esecuzione, rimangono bloccati fino a quando tutti i carichi programmabili non hanno completato la propria configurazione con il modem PLC etichettati come master . Quando questo avviene, il network manager sblocca tutti i Nemo, che provvederanno ad aprire la porta seriale con i relativi parametri e iniziare la scambio di dati con il modem (se previsto).

Questa gestione dei threads da parte del network manager è suggerita dal fatto che i modem PLC lavorano in coppia, con un modello di tipo master-slave. Fino a quando il modem master non ha completato la configurazione del carico programmabile a lui connesso, non richiede alcuna informazione al modem slave, che pertanto invia richieste senza ottenere alcuna risposta. Quando il carico programmabile viene attivato, anche il Nemo diventa operativo, e gli strumenti possono quindi operare il consueto scambio di dati;

3. quando tutti i dispositivi virtuali sono stati correttamente inizializzati, essi gestiranno lo scambio di informazioni con i modem al fine di operare, se necessario, la compensazione della potenza reattiva della rete. Il

network manager sfrutta i nuovi dati provenienti dai carichi programmabili relativi al generatore di potenza con cui essi sono rappresentati per risolvere la rete e trovare tensioni e correnti di ciascun componente elettrico della rete.

A questo punto, il gestore della rete provvede ad aggiornare tutte le strutture di tipo *electricData* di ciascun dispositivo virtuale, in modo da fornire loro nuovi dati con cui operare. Vengono poi aggiornate con gli stessi valori anche le strutture di tipo *communicationData*; queste informazioni sono quindi trasferite al server grafico che provvederà a visualizzarle all'utente sull'apposita interfaccia. Il network manager, accede alle strutture dei vari threads solamente in mutua esclusione, sulla base della disponibilità del semaforo di sincronizzazione posseduto dallo stesso thread;

4. le azioni eseguite al punto tre sono inserite in un ciclo *while*, dove ad ogni iterazione viene inoltre controllato che non sia verificata alcuna condizione di errore o terminazione per un dispositivo virtuale. La causa per cui questo può accadere è dovuta ad un'interruzione della comunicazione seriale tra thread e modem, o perchè il cavo seriale è scollegato o perché il modem non è più attivo. Quando questo si verifica, i dispositivi virtuali, che condividono tra loro un segnale di arresto, concludono la loro attività, così come il gestore della rete e il programma termina.

Tutti i threads attivi nella rete condividono l'esame e l'aggiornamento di una condizione di terminazione, specificata dal valore di una variabile definita *stopSignal*. Se un thread riscontra, durante la sua esecuzione, un errore o un malfunzionamento, modifica lo stato della variabile *stopSignal* dal valore 0 (previsto di default) al valore 1.

La principale causa di errore per i thread di tipo Nemo o carico programmabile riguarda la disconnessione del cavo seriale o lo scollegamento dei modem PLC della rete, condizioni che forzano i thread alla chiusura della porta seriale. Il network manager, può interrompere le proprie attività nel caso un dispositivo virtuale smetta di funzionare o se il server grafico gli comunica l'intenzione dell'utente di chiudere il programma. In ognuno di questi casi, un thread aggiorna la variabile *stopSignal* portandone a 1 il valore, questo fa in modo che tutti gli altri threads della rete, esaminando la condizione di terminazione, constatino che essa è verificata e quindi interrompano la propria attività, con la conseguente chiusura del programma.

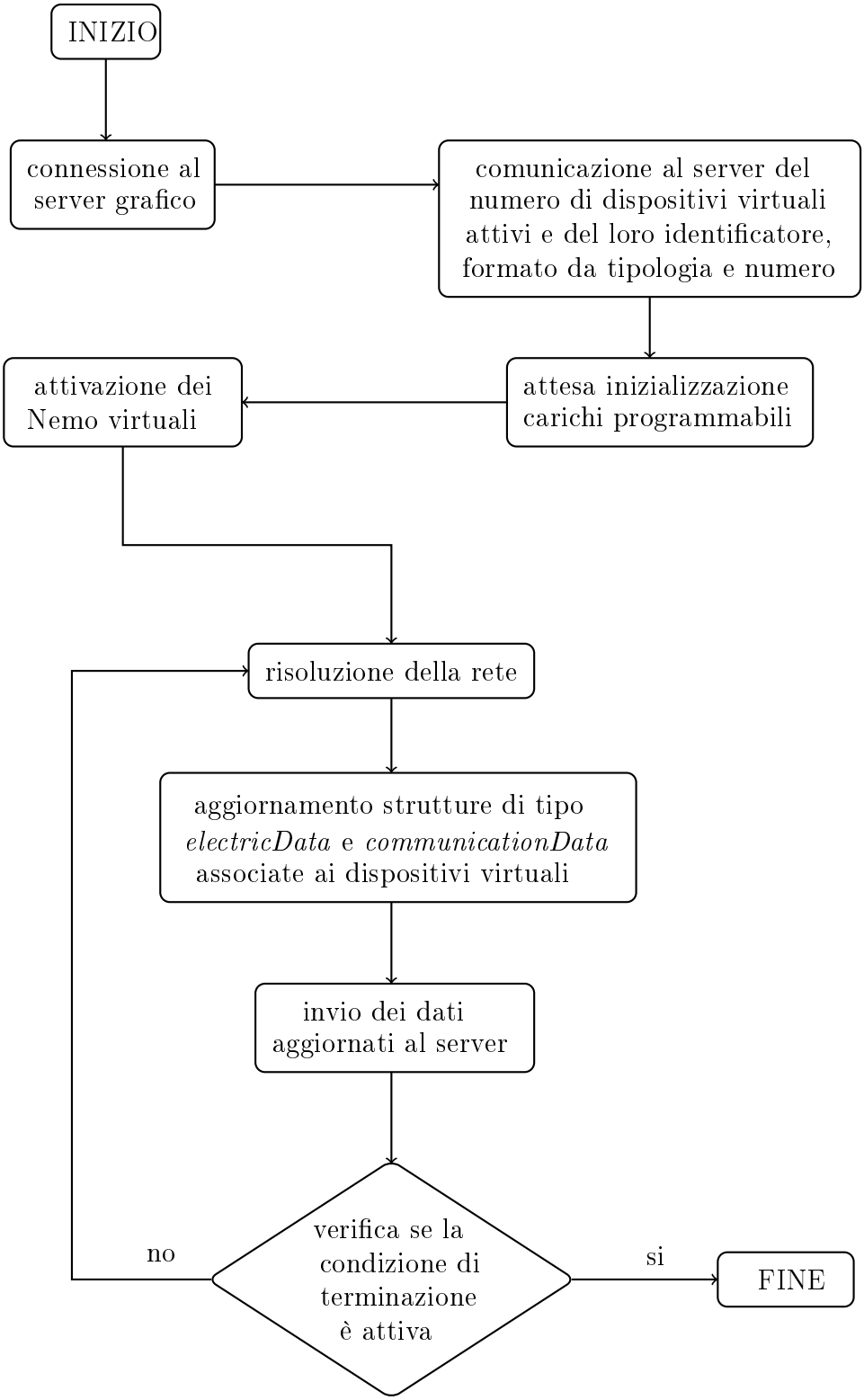

Figura 4.1: Schema di funzionamento del Network Manager

4.3.3 Schema riassuntivo

In Figura 4.1, sono riassunte le principali operazioni svolte dal thread gestore della rete durante la sua esecuzione.

Capitolo 5

Test con la rete virtuale

Nei precedenti capitoli sono state descritte le parti fondamentali che costituiscono il software della rete virtuale. Vengono ora presentate le modalità con cui sono stati eseguiti i test sulla rete virtuale, in modo da verificare il corretto funzionamento della stessa specialmente in merito alla compensazione della potenza reattiva, il principale obiettivo che si vuole raggiungere.

Il controllo dei valori di tensione, corrente potenza attiva e potenza reattiva per ciascun dispositivo viene eseguito sfruttando come detto, un server grafico che si occupa di visualizzare i dati forniti dal gestore della rete virtuale.

La rete virtuale utilizzata nei test ricalca la struttura della rete elettrica presente nel test-bed per le Smart Grid presente in laboratorio, dove gli strumenti Nemo D4-L+ e Chroma 63804 sono sostituiti dalle loro versioni software. Lo schema per la rete elettrica virtuale è riportato in Figura 5.1.

Lo schema presentato in Figura 5.1 corrisponde alla configurazione ibrida del sistema di test riportato in Figura 5.2.

Nei due schemi di Figura 5.1 e Figura 5.2 si possono individuare nel dettaglio:

- due modem PLC in configurazione master-slave con funzione analoga a quella svolta nelle reti con soli strumenti reali;

- un thread che agisce come un Nemo virtuale e che misura tensione, corrente, potenza attiva e reattiva di una sottorete di carico rappresentata dall'impedenza ($\dot{Z}_{sub}$) nella rete elettrica virtuale il cui valore è stato modificato per verificare la compensazione della potenza reattiva. I dati rilevati dal Nemo virtuale verranno comunicati al modem PLC slave cui è connesso tramite porta seriale;

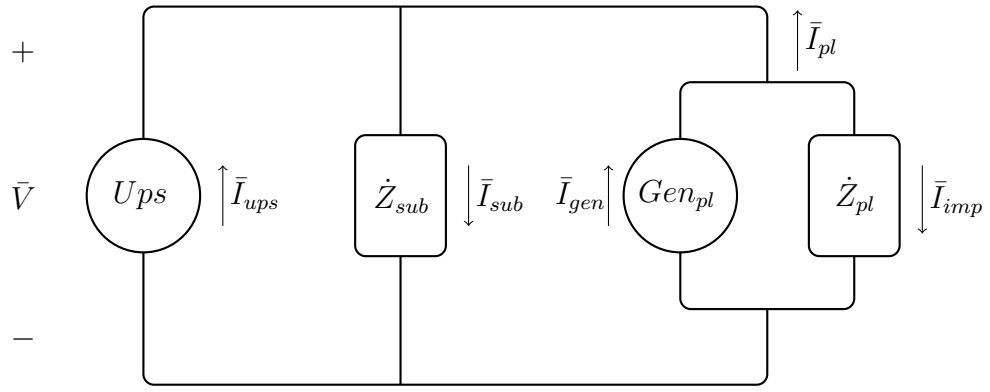

Figura 5.1: Rete elettrica virtuale di test

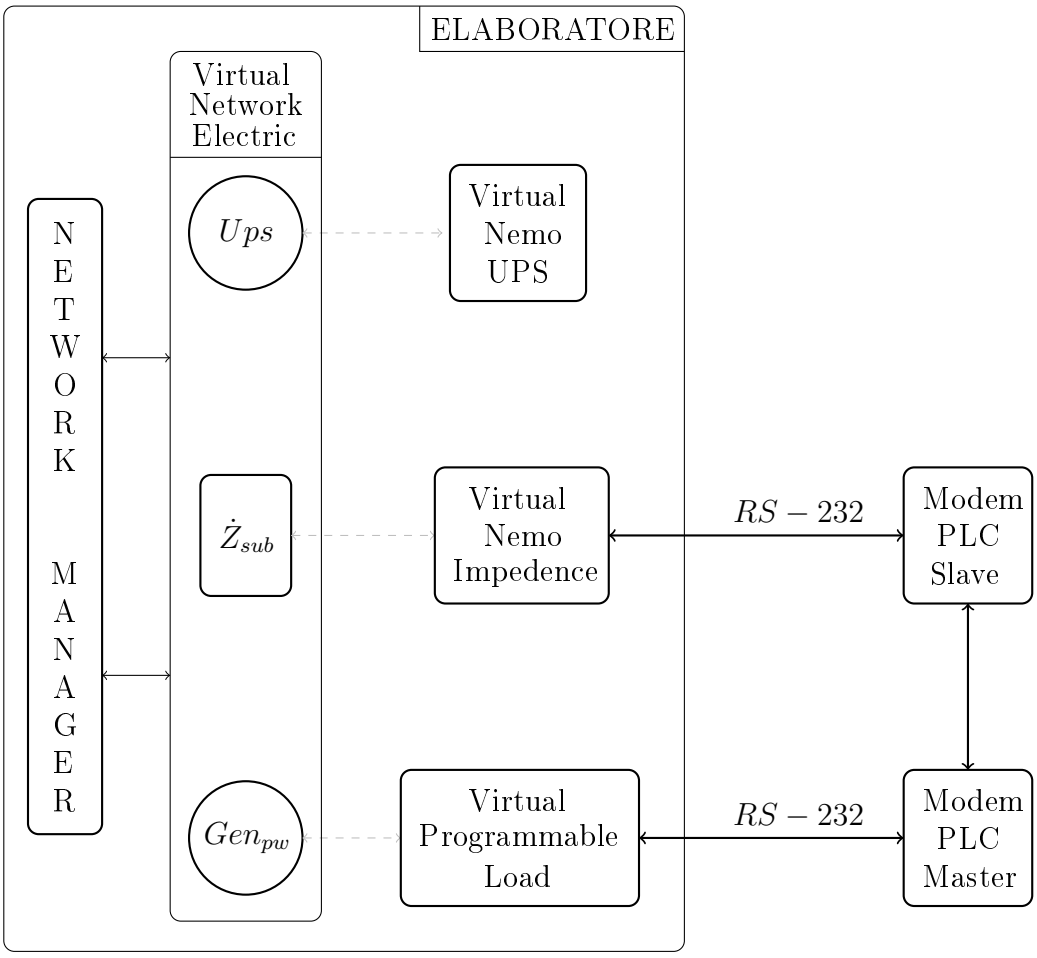

Figura 5.2: Configurazione del sistema di test per la rete elettrica di Fig. 5.1

- un thread che rappresenta un Nemo virtuale che misura tensione, corrente, potenza attiva e reattiva di un UPS con impedenza di ingresso nulla e tensione impressa fissata, utilizzato per verificare la compensazione della potenza reattiva nella rete: il valore della potenza reattiva visualizzata sul server grafico per l'UPS virtuale dovrà quindi essere nulla;

- un thread che sostituisce il carico programmabile reale, che riceve comandi ed istruzioni dal modem PLC master, e provvede ad impostare in particolare i valori di potenza attiva, power factor e crest factor che gli vengono comunicati: tali quantità sono utilizzate per calcolare la potenza reattiva da compensare e sono utili al fine di aggiornare i valori della corrente del generatore di potenza $\bar{I}_{gen}$ che è riportato nello schema della rete elettrica virtuale;

- un thread che funge da gestore della rete, che risolve la rete elettrica virtuale e aggiorna i valori di tensione, corrente, potenza attiva e reattiva per tutti i dispositivi virtuali attivi. Queste informazioni saranno poi comunicate al server grafico che provvede a visualizzarle all'utente.

La configurazione della rete elettrica e dei dispositivi virtuali è eseguita in una funzione **main()** principale, dove:

- viene stabilito il numero di Nemo e carichi programmabili che saranno impiegati all'interno della rete, e creati array di strutture *virtualNemoData* e *virtualProgrammableLoadData* per ciascuno strumento utilizzato;

- vengono associate a ciascun dispositivo la porta seriale dell'elaboratore per la comunicazione con il modem PLC se questa è prevista, e l'identificatore formato da tipologia di dispositivo e numero;

- i semafori di mutua esclusione per ogni dispositivo virtuale sono creati ed inizializzati al valore 1 (sono quindi attivi), viene inizializzato a 1 anche il semaforo "globale" condiviso tra i threads dei Nemo, carichi programmabili e gestore della rete;

- viene deciso quale schema di rete elettrica virtuale utilizzare, configurando opportunamente i parametri della rete stessa come impedenze, tensioni e correnti impresse;

- si procede alla creazione dei threads per ciascun Nemo e carico virtuale che si vuole utilizzare nella rete, oltre al thread gestore della rete;

- i threads vengono mandati in esecuzione e il programma evolve in maniera indipendente fino alla chiusura imposta dall'utente o al verificarsi di una condizione di terminazione.

5.0.4 Operazioni con i numeri complessi in C

Le reti elettriche virtuali che sono state utilizzate durante le fasi di test in laboratorio operano tutte in regime sinusoidale monofase. La risoluzione di tali reti prevede l'impiego dei fasori al fine di rendere più agevole il calcolo di tensioni e correnti di tutti i lati della rete stessa. I fasori sono in generale quantità complesse, così come lo sono anche le impedenze dei bipoli che sono stati rappresentati.

Per consentire di gestire le operazioni con i numeri complessi, è stata utilizzata una apposita libreria prevista dal codice C, "*complex.h*" che mette a disposizione un insieme di funzioni per lavorare con quantità complesse. Analizziamo brevemente quelle fondamentali:

- la rappresentazione di un numero complesso qualsiasi, ad esempio $z = a + jb$, avviene con la scrittura dell'istruzione seguente:

  *double complex z = a + I * b;* dove I specifica l'unità immaginaria j;

- **creal(z)** consente di calcolare la parte reale di un numero complesso z;

- **cimag(z)** permette invece di calcolare la parte immaginaria di un numero complesso z;

- **cabs(z)** è usata per calcolare il modulo di un numero complesso z;

- con la funzione **carg(z)** è possibile trovare l'argomento (detto anche fase) di un numero complesso z;

- **conj(z)** permette di calcolare il coniugato di un numero complesso, nello specifico di eseguire l'operazione $z^* = a - jb$;

5.0.5 Configurazione della rete di test e risultati ottenuti

I test sono stati eseguiti come detto utilizzando la rete elettrica virtuale presente nello schema elettrico a pagina 74 (Figura 5.1). Per l'UPS si è utilizzata una tensione di 230 V (valore efficace), che rimane tale anche per gli altri due lati della rete dove si trovano Nemo e carico programmabile dato che sono tutti e tre in parallelo. L'impedenza del carico programmabile $\dot{Z}_{pl}$ è stata

GRANDEZZA	Nemo Ups	Nemo sub	Carico Programmabile
Tensione	230.000 V	230.000 V	230.000 V
Corrente	1.872 A	1.438 A	0.435 A
Sfasamento	0.000 rad	0.000 rad	-3.142 rad
Potenza Attiva	430.625 W	330.625 W	-100.000 W
Potenza Reattiva	0.000 VAR	0.000 VAR	0.000 VAR

Tabella 5.1: Misurazioni sulla rete con impedenza $\dot{Z}_{sub} = 160.00 + j0.00$

GRANDEZZA	Nemo Ups	Nemo sub	Carico Programmabile
Tensione	230.000 V	230.000 V	230.000 V
Corrente	0.434 A	0.145 A	0.457 A
Sfasamento	-0.007 rad	-1.571 rad	-2.827 rad
Potenza Attiva	99.875 W	0.000 W	-99.875 W
Potenza Reattiva	-0.698 VAR	-33.238 VAR	-32.540 VAR

Tabella 5.2: Misurazioni sulla rete con impedenza $\dot{Z}_{sub} = 0.00 - j1591.55$

fissata ad un valore $100+j100$ mentre quella del Nemo, $\dot{Z}_{sub}$ viene modificata per verificare la corretta compensazione della potenza reattiva: per simulare questa impedenza, sono stati utilizzati dei semplici componenti elettrici (resistore, condensatore, induttore) o da soli o combinati in serie/parallelo. Tali componenti all'interno della rete elettrica virtuale sono descritti da una struttura dati di tipo *electronicComponent*.

Per il calcolo dell'impedenza, nel caso questa avesse parte immaginaria non nulla, la frequenza che si è utilizzata è $f = 50$ Hz e quindi la corrispondente pulsazione angolare si è trovata con la formula $\omega = 2\pi f$. I risultati, che sono stati registrati con differenti valori di $\dot{Z}_{sub}$, sono riportati di seguito nelle tabelle 5.1-5.5. (Importante nota: i valori di correnti e tensioni nelle tabelle sono RMS).

- sottorete di carico monitorata dal Nemo: $R = 160\Omega$ che corrisponde a $\dot{Z}_{sub} = 160.00 + j0.00$; i risultati con questa configurazione della rete elettrica virtuale sono riportati nella Tabella 5.1;

- sottorete di carico monitorata dal Nemo: $C = 2\mu F$ che corrisponde a $\dot{Z}_{sub} = 0.00 - j1591.55$; i risultati con questa configurazione della rete elettrica virtuale sono riportati nella Tabella 5.2;

- sottorete di carico monitorata dal Nemo: $L = 840mH$ che corrisponde a $\dot{Z}_{sub} = 0.00 + j263.89$; i risultati con questa configurazione della rete elettrica virtuale sono riportati nella Tabella 5.3;

GRANDEZZA	Nemo Ups	Nemo sub	Carico Programmabile
Tensione	230.000 V	230.000 V	230.000 V
Corrente	2.645 A	0.872 A	2.784 A
Sfasamento	0.001 rad	1.571 rad	2.824 rad
Potenza Attiva	608.400 W	0.000 W	-608.400 W
Potenza Reattiva	0.622 VAR	200.462 VAR	199.840 VAR

Tabella 5.3: Misurazioni sulla rete con impedenza $\dot{Z}_{sub} = 0.00 + j263.89$

GRANDEZZA	Nemo Ups	Nemo sub	Carico Programmabile
Tensione	230.000 V	230.000 V	230.000 V
Corrente	0.534 A	0.172 A	0.541 A
Sfasamento	-0.004 rad	-1.451 rad	-2.824 rad
Potenza Attiva	122.906 W	4.733 W	-118.173 W
Potenza Reattiva	-0.490 VAR	-39.274 VAR	-38.785 VAR

Tabella 5.4: Misurazioni sulla rete con impedenza $\dot{Z}_{sub} = 160.00 - j1327.65$

- sottorete di carico monitorata dal nemo: circuito RLC in serie con $R = 160\Omega$, $L = 185mH$, $C = 2\mu F$ che corrisponde a $\dot{Z}_{sub} = 160.00 - j1327.65$; i risultati con questa configurazione della rete elettrica virtuale sono riportati nella Tabella 5.4;

- sottorete di carico monitorata dal Nemo: circuito RLC in parallelo con $R = 160\Omega$, $L = 185mH$, $C = 2\mu F$ che corrisponde a $\dot{Z}_{sub} = 127.41 + j64.44$; i risultati con questa configurazione della rete elettrica virtuale sono riportati nella Tabella 5.5;

I risultati presenti nelle Tabelle 5.1-5.5 evidenziano per il carico programmabile valori negativi per la potenza attiva ed una potenza reattiva con lo stesso segno della potenza reattiva misurato dal Nemo virtuale che monitora la sottorete di carico: questo si spiega considerando che il carico programma-

GRANDEZZA	Nemo Ups	Nemo sub	Carico Programmabile
Tensione	230.000 V	230.000 V	230.000 V
Corrente	3.645 A	1.611 A	2.324 A
Sfasamento	0.001 rad	0.468 rad	2.824 rad
Potenza Attiva	838.384 W	330.621 W	-507.762 W
Potenza Reattiva	0.575 VAR	167.218 VAR	166.643 VAR

Tabella 5.5: Misurazioni sulla rete con impedenza $\dot{Z}_{sub} = 127.40 - j64.44$

bile, che come detto viene simulato nella rete utilizzando un parallelo tra un generatore di corrente ed una impedenza, sta effettivamente assorbendo potenza, pertanto la convenzione dei generatori impone che la potenza uscente sia negativa.

Come mostrano le tabelle riportate nelle pagine 77 e 78, il valore della potenza reattiva rilevato dal Nemo virtuale associato all'UPS è nullo solamente nel caso non siano presenti elementi reattivi nella rete e quindi il carico programmabile stia simulando un resistore puro, che costituisce la condizione di inizializzazione di questo strumento. In tutti gli altri casi viene visualizzato un valore per Q diverso da zero.

Questa situazione si può spiegare analizzando nel dettaglio il formato dei dati che scambiano tra loro dispositivi virtuali e modem PLC. Il thread che emula il monitor di rete, risponde alle richieste del modem PLC con i valori di tensione, corrente, potenza attiva e potenza reattiva. Per le prime due quantità, il numero di cifre decimali massimo che viene utilizzato nella rappresentazione (indicato con il termine "precisione") è pari a 3, mentre per le potenze è solamente di due cifre. Questi valori così costituiti vengono inviati al modem PLC slave che li usa per comporre lo *status vector* che comunicherà successivamente al modem PLC master se esso li richiederà.

A questo punto, il modem PLC master, utilizza il valore della potenza reattiva che gli è giunto tramite connessione power line per calcolare i valori della potenza attiva, del power factor e del crest factor che comanderà al carico programmabile di impostare. Il dato della potenza attiva comunicato dal modem master è però una quantità di tipo *unsigned int*, ossia un valore intero senza segno: l'assenza di cifre decimali comporta una perdita di informazioni significativa, che influenza anche il calcolo della potenza reattiva che il carico programmabile ricava proprio dai valori di P e PF che gli sono giunti dal modem master. Il dato così ottenuto si discosta leggermente da quanto originariamente comunicato dal Nemo virtuale al modem PLC slave, la differenza è sempre in difetto. Quanto spiegato giustifica la presenza di un errore, seppur marginale, sulla compensazione della potenza reattiva visualizzata dal Nemo virtuale che monitora l'UPS.

Capitolo 6

Modem PLC Virtuali

Al fine di ampliare le possibilità offerte dal simulatore di reti elettriche finora descritto, si è deciso di realizzare le versioni virtuali dei modem PLC. I due modem PLC presenti all'interno della rete di test, operano in configurazione master-slave:

- i due modem PLC, scambiano tra loro informazioni tramite una struttura di supporto, denominata *status vector* composta da quattro campi: tensione (misurata in volt [V]), corrente (misurata in ampere [A]), potenza attiva (misurata in watt [W]) e potenza reattiva (misurata in volt-ampere reattivi [VAR]);

- il modem slave, connesso ad un Nemo D4-L+ tramite una porta RS-232, interroga il monitor di rete con comandi appositi, al fine di ottenere informazioni circa la tensione, la corrente, la potenza attiva e la potenza reattiva misurate dallo strumento. Le richieste sono effettuate in successione, una dopo l'altra. Non appena sono disponibili i nuovi dati, essi vengono inseriti nella struttura dati condivisa, lo *status vector*;

- il modem master è connesso ad un carico programmabile tramite una porta RS-232. Esso, per prima cosa, inizia una fase di configurazione con il carico programmabile, i cui specifica le modalità operative richieste ed imposta i registri dello strumento. Segue poi una fase ciclica in cui, il modem master periodicamente interroga il modem slave per conoscere lo *status vector*: questi dati sono utilizzati successivamente per calcolare i valori della potenza attiva, del power factor e del crest factor da comunicare al carico adattativo.

Al pari di quanto fatto nel capitolo 3 per i dispositivi Nemo D4-L+ e Chroma 63804, si è scelto di rappresentare anche i modem PLC come dei

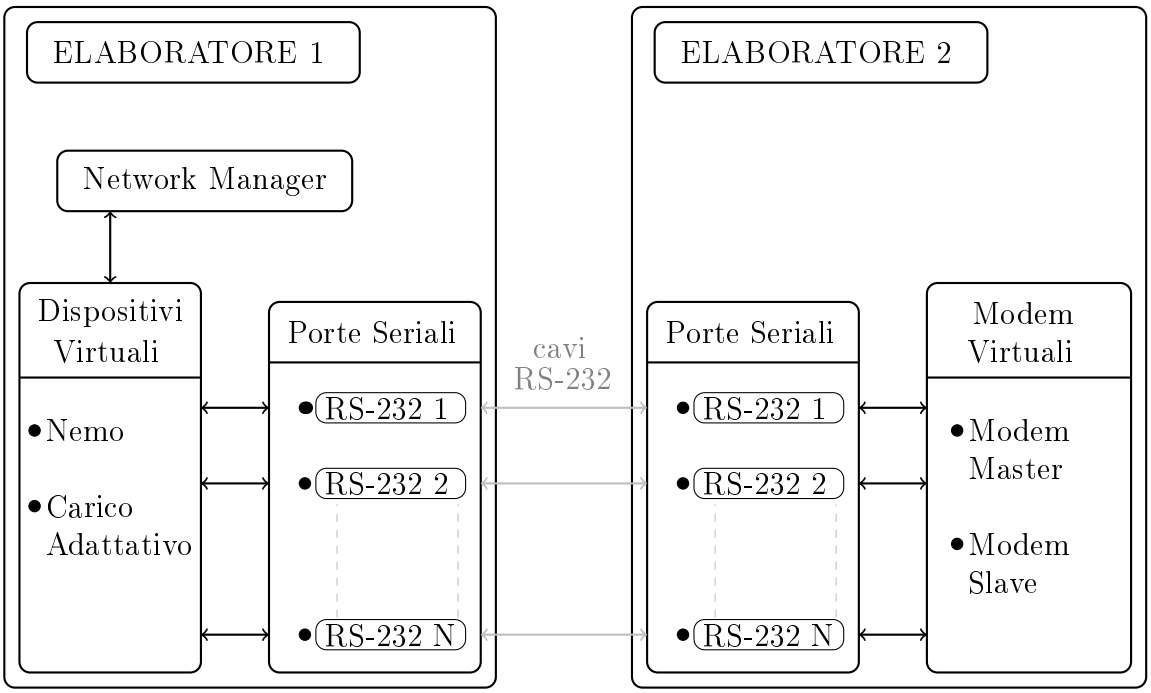

Figura 6.1: Struttura del sistema di test con modem virtuali

threads all'interno dell'elaboratore. I modem virtuali saranno in grado di gestire, al pari delle controparti reali, la comunicazione tramite porta seriale; lo scambio di informazioni avviene o con un Nemo o un carico programmabile virtuale. Non è stata invece emulata la comunicazione PLC tra i due modem; lo scambio di dati tra i due dispositivi avviene tramite la lettura/scrittura di una struttura di supporto, denominata *statusVector* che è la stessa per ciascuna coppia di modem, ovvero di thread. Dato che la struttura è comune ai due thread, essi dovranno accedervi in mutua esclusione, cosa possibile impiegando un semaforo binario (rif. sezione 3.1). Le versioni software dei modem PLC sono eseguite in una macchina separata rispetto a quella dove sono in esecuzione la rete elettrica e annessi dispositivi virtuali; le porte seriali dei due elaboratori sono collegate tramite cavi RS-232 esattamente come avveniva per le porte seriali dei modem PLC reali. Nel complesso, la struttura di test può essere schematizzata come riportato in Figura 6.1. Per la realizzazione delle versioni software dei modem PLC, si è preso come riferimento il codice contenuto nella ROM dello strumento stesso. Il codice per l'emulazione dei modem PLC, è stato scritto sempre in linguaggio C ed utilizza, per la gestione della porta seriale, la già citata libreria *serial_port_functions.h*.

Vengono ora presentate e descritte nel dettaglio le due tipologie di modem PLC che sono state realizzate. E' bene precisare che i modem PLC reali sono identici a livello hardware tra loro, quindi quando si parla di tipologia, si intende il differente comportamento mostrato dai modem in base al codice contenuto all'interno della loro memoria ROM.

6.1 Modem PLC Slave Virtuale

Il modem PLC slave è un modem PLC che gestisce la comunicazione tramite porta seriale con un dispositivo di rete Nemo D4-L+. Il modem necessita di reperire informazioni relative ai valori di alcune grandezze elettriche misurate dal monitor di rete; per ottenere tali dati il modem formula opportune interrogazioni strutturate secondo il protocollo di comunicazione adottato dal Nemo D4-L+ (rif. sezione 3.3.1) e le invia come sequenze di caratteri esadecimali attraverso la porta RS-232 di cui è provvisto. Le richieste hanno tutte la medesima lunghezza, pari ad 8 byte, ed i campi dei messaggi di cui sono composte sono gli stessi. Riportiamoli brevemente nell'ordine in cui compaiono nei messaggi di interrogazione:

- *Instrument Address* formato da un 1 byte, contenente l'identificatore dello strumento Nemo D4-L+;
- *Functional Code* formato da 1 byte, che specifica la tipologia di operazione che si vuole effetture, lettura o scrittura di dati;
- *First Word Address* formato da due bytes, che indica la posizione di partenza della memoria nel Nemo D4-L+ da cui si vuole iniziare a leggere/scrivere dati;
- *Words Number* formato da due bytes, indica il numero di parole da leggere/scrivere;
- *CRC16* formato da 2 bytes, è il risultato del calcolo del CRC a 16 bit.

I messaggi inviati dal modem PLC al Nemo sono relative alla lettura di informazioni (quindi il valore del *Functional Code* è invariato, 0x03); le differenze tra le quattro tipologie di richieste formulate dal modem riguardano i valori del campi *First Word Address* e *Words Number*. Nel dettaglio:

- richiesta per conoscere la tensione, V: *First Word Address* è 0x301 mentre *Words Number* vale 6;

- richiesta per conoscere la corrente, I: *First Word Address* è 0x30d mentre *Words Number* vale 6;

- richiesta per conoscere la potenza attiva, P: *First Word Address* è 0x102c mentre *Words Number* vale 9;

- richiesta per conoscere la potenza reattiva, Q: *First Word Address* è 0x1035 mentre *Words Number* vale 9.

Anche il campo *CRC16* varia per ciascuna richiesta, ma esso viene inserito per la verifica della correttezza del messaggio quando questo viene ricevuto dal destinatario ed analizzato. Le informazioni fornite dal Nemo D4-L+ per ognuna delle quattro grandezze elettriche (tensione, corrente, potenza attiva e reattiva) sono relative a sistemi trifase, quindi per la tensione ad esempio, sono forniti tre distinti valori, uno per ciascuna delle tre fasi. Nel caso in cui la rete elettrica in cui il Nemo opera sia sinusoidale monofase (come quella utilizzata nei test), solo l'indicazione della prima fase è significativa. Di seguito, considereremo come valore fornito dal Nemo solamente quello relativo alla prima fase per ciascuna grandezza elettrica.

Nel caso in cui la comunicazione seriale non presenti problemi, il modem slave invia i messaggi di interrogazione al Nemo uno di seguito all'altro; non appena il messaggio di risposta per la richiesta corrente viene ricevuto, il modem provvede ad inviare la richiesta seguente. Il funzionamento del modem è quindi ciclico, nel senso che esso invia e riceve i messaggi di richiesta e risposta per ciascuna delle quattro grandezze elettriche (V, I, P, Q); quando tutti i dati necessari sono stati ottenuti, il modem inizia un nuovo ciclo di richieste. Le informazioni ricevute dal Nemo ad ogni nuovo ciclo di richieste, vengono salvate in una struttura denominata *statusVector*. Essa è formata da quattro campi di tipo *float* dove vengono memorizzati i valori di tensione, corrente, potenza attiva e potenza reattiva relativi ad uno stesso ciclo di richieste e altri campi necessari alla sincronizzazione dei due threads che condividono l'utilizzo della struttura, tra cui menzioniamo il campo *new_status_vector* che consente di verificare se i dati presenti nella struttura sono aggiornati o sono già stati letti: *new_status_vector* è posto ad 1 dopo ogni operazione di scrittura mentre è posto a 0 dopo ogni operazione di lettura. E' importante sottolineare il fatto che valori delle grandezze elettriche relativi a cicli di richiesta diversi sono da considersi informazioni inconsistenti, dato che la tipologia della rete può essere stata modificata durante quell'intervallo temporale, per questo motivo i valori presenti nello *status vector* sono sempre relativi ad uno stesso ciclo di richieste.

6.1.1 Funzioni del Modem Slave Virtuale

Il software che emula le azioni compiute dal modem PLC slave si trova in un file denominato "virtual_modem_slave_functions.h": questo file contiene tutte le funzioni e le strutture dati previste per funzionamento del modem slave virtuale. Analizziamole nel dettaglio:

- *struct virtualModemNMemory*: è una struttura dati dove vengono salvate informazioni di differente tipologia relative al modem virtuale. Esse sono ad esempio l'identificatore del modem virtuale, la porta seriale dell'elaboratore utilizzata per le comunicazioni e annessa velocità di funzionamento, array di caratteri dove vengono salvati i messaggi di richiesta da inviare e i messaggi di risposta ricevuti dal Nemo, un puntatore alla struttura di tipo *statusVector* dove vengono salvate le informazioni ricevute e un puntatore ad un semaforo binario che consente di accedere in maniera esclusiva alla struttura *statusVector*;

- **void initializeModemData(struct virtualModemNMemory* memory)**: questa funzione permette di inizializzare alcuni campi della struttura di tipo *struct virtualModemNMemory* passata come parametro di chiamata, come gli array per l'invio/ricezione dei messaggi da porta seriale e l'identificatore del dispositivo;

- **void send_delay(int milliseconds)**: questa funzione permette di simulare l'attesa di una quantità di tempo in millisecondi indicata dal parametro *milliseconds*;

- **uint16_t calculate_crc(uint8_t *buffer, uint16_t buffer_dimension)**: questa funzione permette di calcolare il crc (cyclic redundacy check) a 16 bit per un buffer, ossia un'area di dati di dimensione specificata. Nel dettaglio:

  - *uint8_t *buffer* è il puntatore al buffer su cui viene effettuato il calcolo del crc;
  - *uint16_t buffer_dimension* specifica la lunghezza del buffer;
  - la funzione **calculate_crc** restituisce un unsigned integer a 16 bit che costituisce il risultato del calcolo del cyclic redundacy check.

- **void makeNewRequest(struct virtualModemNMemory* memory)**: questa funzione consente di creare un nuovo messaggio di richiesta sulla base del valore della variabile *cycleOfRequest* contenuta nella struttura di tipo *virtualModemNMemory* puntata da *memory*.

Il possibili valori della variabile *cycleOfRequest* sono indicativi della grandezza elettrica misurata dal monitor di rete che si vuole conoscere: 0 per la tensione, 1 per la corrente, 2 per la potenza attiva e 3 per la potenza reattiva. La richiesta da inviare viene salvata in un apposito array denominato *messageRequest* della struttura *virtualModemNMemory* stessa;

- **int sendRequestToNemo(struct virtualModemNMemory* memory)**: questa funzione consente di inviare il messaggio di richiesta contenuto nel campo *messageRequest* della struttura di tipo *virtualModemNMemory* puntata da *memory*. La funzione restituisce il valore di ritorno della system call **write()**, ossia il numero di caratteri scritti in caso di successo, o -1 in caso di errore;

- **int receiveAnswerFromNemo(struct virtualModemNMemory* memory)**: questa funzione consente di ricevere il messaggio di risposta proveniente dal Nemo in seguito all'invio di un'opportuna richiesta. La quantità di dati da ricevere è relativa al valore del campo *cycleOfRequest* della struttura di tipo *virtualModemNMemory* puntata da *memory*; le nuove informazioni ricevute sono salvate in un array di caratteri opportunamente dimensionato contenuto sempre nella struttura di tipo *virtualModemNMemory*; La funzione si aspetta di ricevere la risposta dal Nemo entro un intervallo temporale di lunghezza prefissata, nel caso venga ricevuto un messaggio di lunghezza inferiore rispetto a quella attesa o non venga ricevuto nessun carattere, la funzione restituisce un valore *integer* pari a 0, se invece il messaggio proveniente dal Nemo ha dimensione pari a quella prevista, il valore di ritorno della funzione è 1;

- **int analyzeAnswerFromNemo(struct virtualModemNMemory* memory)**: questa funzione consente di analizzare il messaggio ricevuto tramite la funzione **receiveAnswerFromNemo()**. Viene innanzitutto controllata la correttezza della risposta, verificando che la funzione **calculate_crc()** restituisca zero come valore di ritorno; se ciò è verò vengono esaminati i campi *Instrument Address*, *Functional Code* e *Bytes Number* del messaggio ricevuto controllando per i primi due, che siano uguali ai corrispettivi campi del messaggio di richiesta corrente, mentre per il terzo che il suo valore sia pari al doppio del valore del campo *Words Number* del messaggio di richiesta corrente. I messaggi di risposta che segnalano errori nella richiesta non vengono gestiti, o per meglio dire, sono trattati come se fossero errori di ricezione in cui il numero di caratteri letti dalla porta seriale è inferiore a

quanto previsto, pertanto il messaggio è ignorato. Nel caso in cui tutte le verifiche precedenti abbiano esito positivo, il messaggio è corretto e si procede all'estrazione dei valori delle fasi ed eventuali segni delle stesse(se si tratta di potenze) della grandezza elettrica che costituisce l'informazione del messaggio di risposta. In questa operazione sono previste conversioni da dati originari in formato esadecimale a valori di tipo *floating point*.

6.1.2 Il thread VirtualModemSlave

Mentre le funzioni presentate nella precedente sezione consentono di emulare le azioni svolte dal modem PLC slave, il funzionamento vero e proprio del modem viene realizzato tramite un thread cui è connessa una struttura di tipo *virtualModemNMemory*. Esaminiamo i dettagli dell'esecuzione del thread per il modem slave virtuale:

1. per prima cosa, la struttura di tipo *virtualModemNMemory* associata al thread del modem viene inizializzata chiamando la funzione **initializeModemData()**. Segue poi una fase di configurazione della porta seriale, in cui la porta RS-232 dell'elaboratore designata per il modem virtuale viene aperta associandola ad un file descriptor, vengono poi impostati i parametri della comunicazione (baud-rate, bit di parità, input canonico o raw, minimo numero di caratteri da leggere e tempo inter-carattere) che sono compatibili con quelli utilizzati dal Nemo virtuale. Sono eseguite delle verifiche per controllare che le opzioni specificate siano state applicate nel modo corretto; se nessun errore viene riscontrato la porta seriale può essere utilizzata senza problemi;

2. terminata la fase iniziale di setup, il thread entra in una fase di funzionamento ciclico in cui esegue le seguenti operazioni:

   (a) crea una nuova richiesta da inviare al Nemo tramite la funzione **makeNewRequest()**: i campi della richiesta sono compilati in base al valore della variabile *cycleOfRequest* della struttura *virtualModemMemory* collegata al thread;

   (b) invia la nuova richiesta creata al punto precedente con la funzione **sendRequestToNemo()**. Viene controllato il valore di ritorno della funzione per assicurarsi che il messaggio sia stato inviato correttamente;

   (c) dopo aver inviato il messaggio al Nemo, viene ricevuta la risposta chiamando la funzione **receiveAnswerFromNemo()**. Il valore di ritorno della funzione viene salvato in una variabile temporanea;

(d) nel caso in cui la funzione **receiveAnswerFromNemo()** restituisca 1, il messaggio è stato correttamente ricevuto e può essere analizzato con la funzione **analyzeAnswerFromNemo()**. Il valore di ritorno della funzione è salvato in una variabile temporanea: se esso è 1, la risposta è corretta e il valore della grandezza elettrica contenuto nel messaggio è salvato nella memoria del modem virtuale (che è la struttura di tipo *virtualModemNMemory*);

(e) se i valori restituiti dalle funzioni **receiveAnswerFromNemo()** e **analyzeAnswerFromNemo()** sono entrambi 0 o anche solo uno dei due vale 0, il modem virtuale individua una condizione di errore, incrementa un apposito contatore di errori ed azzera il valore della variabile *cycleOfRequest* della struttura *virtualModemNMemory*. Quando il contatore supera la soglia massima prevista, l'esecuzione del ciclo si interrompe. Gli errori vengono registrati solamente nel caso non vengano ricevute risposte alle interrogazioni inviate al Nemo, situazione che indica un malfunzionamento della comunicazione seriale. Nel caso in cui la comunicazione seriale ricominci a funzionare dopo un periodo di inattività, il contatore degli errori viene azzerato.

3. quando un ciclo di richieste viene completato correttamente, ossia tutti i messaggi di risposta vengono ricevuti senza interruzioni ed errori e la variabile *cycleOfRequest* raggiunge il valore 3, i valori della grandezze elettriche misurati dal Nemo vengono inseriti nella struttura di tipo *statusVector* che il modem slave virtuale condivide con il modem master virtuale, notificando la disponibiltà di valori aggiornati ponendo ad 1 il campo *new_status_vector* della struttura stessa. A questo punto, un nuovo ciclo di richieste viene iniziato. Il funzionamento finora descritto (punti 2 e 3) si interrompe solamente quando il contatore degli errori in ricezione supera un limite prefissato. Come spiegato in precedenza, una condizione di errore si verifica a causa di un malfunzionamento della comunicazione seriale tra i due dispositivi: essendo essi entrambi in esecuzione su un elaboratore, la principale causa di errore è dovuta al parziale o totale disconnessione del cavo RS-232 dalle porte seriali degli elaboratori. Un ulteriore causa è ovviamente relativa all'utilizzo di un codice C non adeguato alle problematiche che devono essere gestite.

6.1.3 Schema riassuntivo

In Figura 6.2 è riportato lo schema di funzionamento del modem slave virtuale, dove sono evidenziate le operazioni principali svolte durante la sua

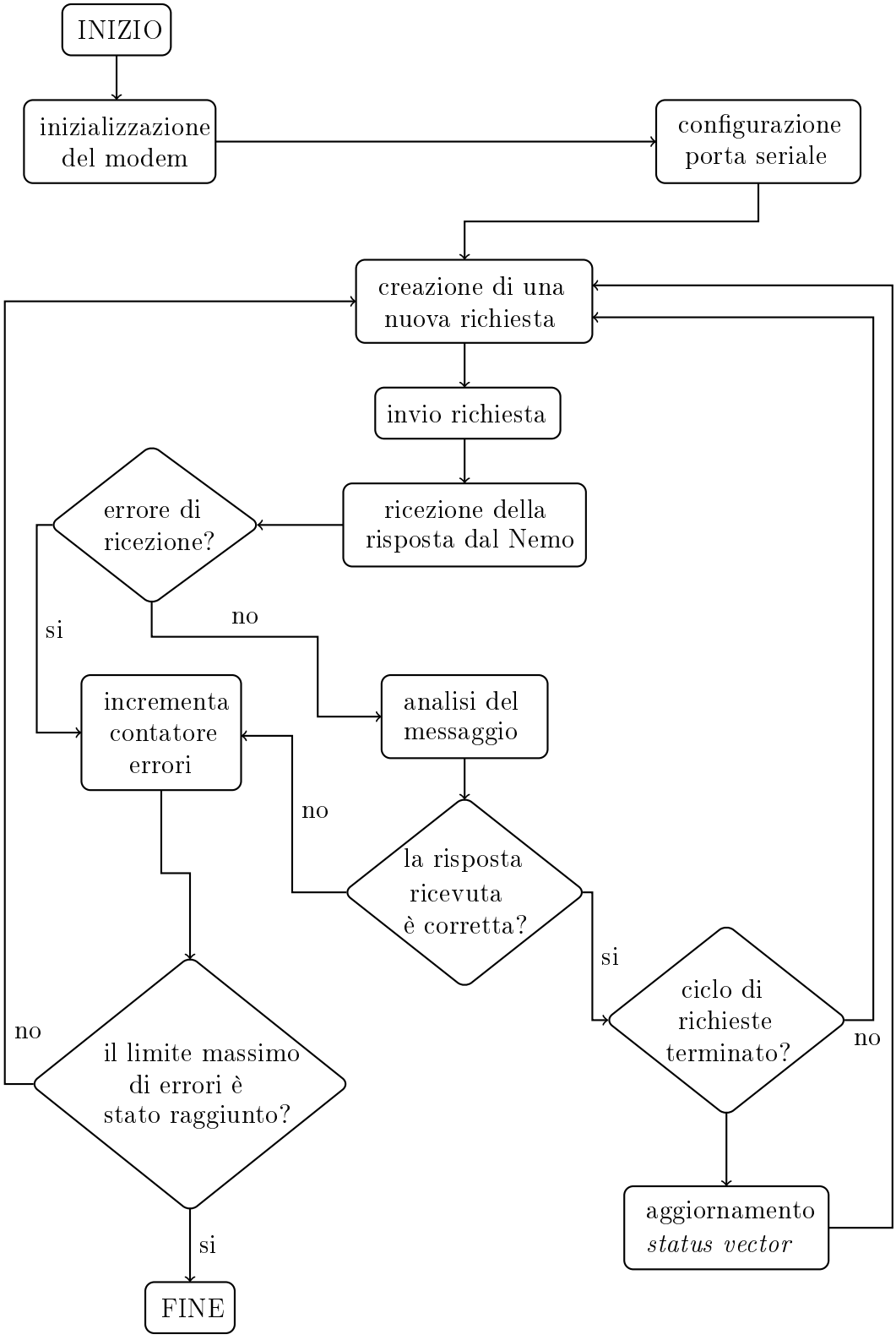

Figura 6.2: Schema di funzionamento del modem slave virtuale

esecuzione.

6.2 Modem PLC Master Virtuale

Il modem PLC master è un modem PLC che gestisce la comunicazione tramite porta seriale con un carico programmabile Chroma 63804. Il modem, tramite appositi comandi strutturati secondo il protocollo SCPI, istruisce il carico programmabile sui valori che devono essere impostati relativamente a modalità operative, registri dello strumento o grandezze elettriche che consentono al dispositivo di simulare un determinato tipo di carico all'interno della rete. I parametri da specificare per la simulazione del carico nel Chroma 63804 sono tre: il valore della potenza attiva (P), il valore del power factor (PF) ed il valore del crest factor (CF). L'obiettivo del modem PLC master è quello di calcolare i valori di P, PF e CF che consentono di compensare la potenza reattiva di un componente elettrico presente all'interno della rete i cui valori di tensione, corrente, potenza attiva e reattiva sono misurati da un Nemo D4-L+; queste informazioni vengono raccolte da un modem slave che le rende disponibili al modem master in modo che questo possa realizzare la compensazione di Q con opportuni comandi impartiti al carico adattativo.

La comunicazione tra modem master e carico programmabile può essere suddivisa in due fasi distinte: una prima fase di configurazione, nella quale il modem interroga il carico per conoscere il suo identificatore e lo stato di alcuni registri e provvede, nel caso i messaggi di risposta provenienti dal carico adattativo siano corretti, ad impostare le modalità operative dello strumento e ad inizializzarlo richiedendo una potenza attiva di 100 W, un fattore di potenza pari ad 1 ed un crest factor di 1.414. Essi corrispondono alla simulazione di un carico puramente resistivo. Terminata la fase di setup, inizia una fase ciclica il cui funzionamento è il seguente: il modem master legge i valori di tensione, corrente, potenza attiva e potenza reattiva inviatigli dal modem slave tramite una struttura dati chiamata *status vector* di cui si è già discusso in precedenza; i valori presenti nella struttura consentono al modem master di calcolare i valori di potenza attiva, power factor e crest factor che verranno successivamente comunicati al carico programmabile con tre comandi distinti. Il modem non si aspetta di ricevere alcuna risposta dal carico adattativo, pertanto esegue queste azioni ciclicamente fino a quando la sua esecuzione non viene interrotta.

Il modem virtuali adottano in realtà un meccanismo di comunicazione di tipo indiretto (la comunicazione PLC come detto non viene simulata), scambiando informazioni tramite una struttura dati comune ad entrambi i tipi di modem, denominata *statusVector* i cui dettagli sono già stati presentati

nella sezione precedente. Pertanto, a differenza dei modem PLC reali, il modem master virtuale non effettua alcuna richiesta al modem slave per conoscere lo *status vector* né tantomeno il modem slave invia dati al modem master.

6.2.1 Funzioni del Modem Master Virtuale

Il software che emula le azioni compiute dal modem PLC master si trova in un file denominato "virtual_modem_master_functions.h": questo file contiene tutte le funzioni e le strutture dati previste per il funzionamento del modem master virtuale. Analizziamole nel dettaglio:

- *struct virtualModemPLMemory*: è una struttura dati dove vengono salvate informazioni di differente tipologia relative al modem virtuale. Esse sono ad esempio l'identificatore del modem virtuale, la porta seriale dell'elaboratore utilizzata per le comunicazioni e annessa velocità di funzionamento, un puntatore alla struttura di tipo *statusVector* dove vengono salvate le informazioni ricevute e un puntatore ad un semaforo binario che consente di accedere in maniera esclusiva alla struttura *statusVector*. Inoltre sono presenti tre campi salvare i valori di potenza attiva, power factor e crest factor;
- **void initializeModemPLData(struct virtualModemPLMemory *memory)**: questa funzione consente di inizializzare alcuni campi della struttura di tipo *struct virtualModemPLMemory* passata come parametro di chiamata, come l'identificatore del dispositivo ed i campi per potenza attiva, power factor e crest factor posti tutti e tre a zero;
- **void sim_delay(int milliseconds)**: questa funzione permette di simulare l'attesa di una quantità di tempo in millisecondi indicata dal parametro *milliseconds*;
- **void sendCommandToPL(int fileDescriptorSerialPort, char *messageToSend, int lengthOfMessage)**: questa funzione consente di inviare un messaggio specificato come parametro d'ingresso al carico programmabile. I tre parametri della funzione sono utilizzati come argomenti di ingresso per la system call **write()** che consente di inviare caratteri tramite porta seriale. In particolare:
  - *int fileDescriptorSerialPort*: file descriptor della porta seriale utilizzata nella comunicazione;
  - *char *messageToSend*: un puntatore al messaggio che si vuole inviare;

  - *int lengthOfMessage*: la lunghezza del messaggio da inviare.

- **int check_answer_from_PL(char *modemID, int fileDescriptorSerialPort, char *answerToMatch, int lengthOfAnswer)**: questa funzione consente di verificare la correttezza della risposta proveniente dal carico programmabile in seguito ad un comando di interrogazione inviato precedentemente dal modem master.

  - *char *modemID* è l'identificatore del modem master, utilizzato solamente per stampe a video di diagnostica per l'utente;
  - *int fileDescriptorSerialPort* è il file descriptor della porta seriale associata al modem master, da cui viene ricevuto il messaggio di risposta del carico adattativo;
  - *char *answerToMatch* è la risposta che il modem master si aspetta di ricevere. Essa viene confrontata con il messaggio letto dalla porta seriale proveniente dal carico programmabile;
  - *int lengthOfAnswer* è la lunghezza della risposta prevista. Il valore di questo parametro viene utilizzato per stabilire il numero di caratteri da leggere dalla porta seriale. Il modem attende un intervallo di tempo prefissato per la ricezione della risposta, se al termine dell'intervallo il numero di caratteri letti è inferiore rispetto a quanto prefissato, una condizione di errore viene rilevata. Se invece il numero di caratteri ricevuto è corretto, la stringa risultante viene confrontata con la stringa che il modem si attende di ricevere utilizzando la funzione C **strcmp()**;
  - il valore di ritorno della funzione è un intero pari a 0 se le due stringhe (ricevuta e prestabilita) sono uguali, 1 in caso contrario.

- **void initialize_programmable_load(struct virtualModemPLMemory *memory)**: questa funzione consente di configurare il carico programmabile impostando tramite appositi comandi le modalità operative e i valori di alcuni registri dello strumento. Sono previste due interrogazioni, formulate dal modem master, per conoscere l'identificativo del carico adattativo e lo stato di alcuni registri: nel caso le risposte non siano ricevute o siano errate, la fase di configurazione viene interrotta e il modem master virtuale termina la sua attività;

- **void set_programmable_load(struct virtualModemPLMemory *memory)**: questa funzione consente di impostare i valori di potenza attiva, power factor e crest factor del carico adattativo mediante l'invio di tre comandi in formato SCPI. La funzione viene eseguita

solamente se la fase di setup iniziale del carico programmabile è andata a buon fine. Prima dell'invio dei tre messaggi il modem master verifica se è disponibile un aggiornamento dei valori delle grandezze elettriche nella struttura *statusVector* condivisa con il modem slave; questo è eseguito tramite un'operazione di lettura del campo *new_status_vector* della struttura stessa: se vale 1, i dati sono stati aggiornati, se invece vale 0 i dati non sono stati modificati dall'ultima operazione di lettura, effettuata sempre dal modem master. In quest'ultimo caso, il modem master provvede ad inviare al carico programmabile i valori della potenza attiva, del power factor e del crest factor salvati nella struttura di tipo *struct virtualModemPLMemory* puntata da *memory*: in pratica, se non sono disponibili nuove informazioni, il modem master invia i dati più recenti di cui dispone. Se sono invece presenti dati aggiornati, il modem master necessita di calcolare i nuovi valori della potenza attiva, del power factor e del crest factor da comunicare al carico programmabile. Le azioni svolte in questa parte della funzione **set_programmable_load()** sono le seguenti:

- per prima cosa, viene verificato se la potenza reattiva contenuta nella struttura *statusVector* è nulla: in caso affermativo, il componente elettrico di cui sono state misurate tensione, corrente, potenza attiva e reattiva è un componente non reattivo, pertanto non c'è alcuna potenza reattiva da compensare. Pertanto i valori specificati per il carico adattativo sono quelli di default ossia $P =$ 100 W, $PF = 1.00$ e $CF = 1.414$;
- se invece la potenza reattiva è non nulla, il suo valore viene utilizzato per calcolare la corrispondente potenza attiva da comunicare al carico. La formula di calcolo è la seguente:

$$P = |Q| \frac{PF}{\sqrt{1 - PF^2}}$$

Dato che il valore in modulo del power factor è noto, pari a 0.97 (le ragioni di tale valore sono già state spiegate nei capitoli 2 e 3) è possibile calcolare la frazione nella precedente equazione, riducendo la formula in questo modo:

$$P = |Q| \, 3.99$$

A questo punto il calcolo della potenza attiva è immediato. Per quanto riguarda il power factor, è importante ricordare che il modulo è fissato ad un valore di 0.95: esso è stato deciso in seguito a

numerosi esperimenti effettuati con il carico programmabile Chroma 63804, ed è lo stesso utilizzato dal carico adattativo virtuale. Rimane a questo punto da stabilire il segno del PF: esso è positivo se la potenza reattiva è negativa, viceversa è negativo se la potenza reattiva è positiva. Per il crest factor, il valore da impostare nel carico adattativo è di 1.55. Per una spiegazione più approfondita del perchè tali valori di PF e CF siano impiegati, si rimanda alla sezione 3.4.5. Si noti che il valore della potenza attiva contenuto nello *statusVector* è ininfluente ai fini del calcolo dei dati di P, PF e CF;

- nei casi precedenti non sono da considerarsi valide situazioni in cui sia la potenza attiva che quella reattiva sono entrambe nulle: ciò corrisponderebbe infatti ad un componente che non assorbe alcuna potenza, cosa non desiderata;
- quando i valori dello *statusVector* vengono utilizzati dal modem master, è necessario porre a zero il campo *new_ status_ vector* dello *statusVector* stesso: questo consente al modem master in un momento successivo, di verificare se i dati presenti nella struttura sono stati aggiornati o sono invariati dalla precedente lettura.

- quando tutti i valori al punto precedente sono stati calcolati, essi vengono salvati all'interno della struttura di tipo *struct virtualModemPLMemory* puntata da *memory* in apposite variabili. Sono questi i valori comunicati al carico programmabile, in questo modo, se non sono disponibili nuovi dati nello *statusVector*, vengono inviati i più recenti valori di P, PF e CF.

6.2.2 Il thread VirtualModemMaster

Le funzioni presentate nella precedente sezione consentono di emulare le azioni svolte dal modem PLC master, ma il funzionamento vero e proprio del modem viene realizzato tramite un thread cui è connessa una struttura di tipo *virtualModemPLMemory*. Esaminiamo i dettagli dell'esecuzione del thread per il modem master virtuale:

1. per prima cosa, la struttura di tipo *virtualModemPLMemory* associata al thread del modem viene inizializzata chiamando la funzione **initializeModemPLData()**. Segue poi una fase di configurazione della porta seriale, in cui la porta RS-232 dell'elaboratore designata per il modem virtuale viene aperta associandola ad un file descriptor, vengono poi impostati i parametri della comunicazione (baud-rate, bit di parità,

input canonico o raw, minimo numero di caratteri da leggere e tempo inter-carattere) che sono compatibili con quelli utilizzati dal carico programmabile virtuale. Sono eseguite delle verifiche per controllare che le opzioni specificate siano state applicate nel modo corretto; se nessun errore viene riscontrato la porta seriale può essere utilizzata senza problemi;

2. terminata la fase di setup, il modem master configura il carico programmabile tramite la funzione **initialize_programmable_load()**: nel caso in cui la configurazione non venga completata correttamente, il modem riconosce la condizione di errore e termina la sua esecuzione, non essendo certo dello stato del carico programmabile;

3. se la fase di configurazione del carico adattativo viene completata correttamente, il modem continua la sua esecuzione con un funzionamento ciclico che consiste semplicemente nel chiamare la funzione **set_programmable_load()**. Dopo aver inviato i tre messaggi contenenti potenza attiva, power factor e crest factor, il thread del modem virtuale simula un'attesa temporale di durata prefissata con la funzione **sim_delay()**. A questo punto il ciclo riprende la sua esecuzione;

4. si noti che il modem master non ha alcun modo di verificare lo stato della comunicazione seriale con il carico programmabile, dato che fatto salvo per i due messaggi ricevuti nella fase di configurazione, è solamente il modem master ad inviare informazioni: la comunicazione è in pratica unidirezionale, pertanto il modem master esegue il ciclo descritto al punto precedente indefinitamente. Si è deciso, per risolvere questa situazione, di sincronizzare il modem master con la condizione di terminazione del modem slave: in pratica, ad ogni ciclo il thread del modem virtuale verifica se il modem slave ha identificato una condizione di errore (che abbiamo detto è quasi sempre dovuta ad un collegamento non corretto tra porte seriali e cavo RS-232), in caso affermativo sospende la comunicazione con il carico programmabile.

6.2.3 Schema riassuntivo

In Figura 6.3 è riportato lo schema di funzionamento del modem master virtuale, dove sono evidenziate le operazioni principali svolte durante la sua esecuzione.

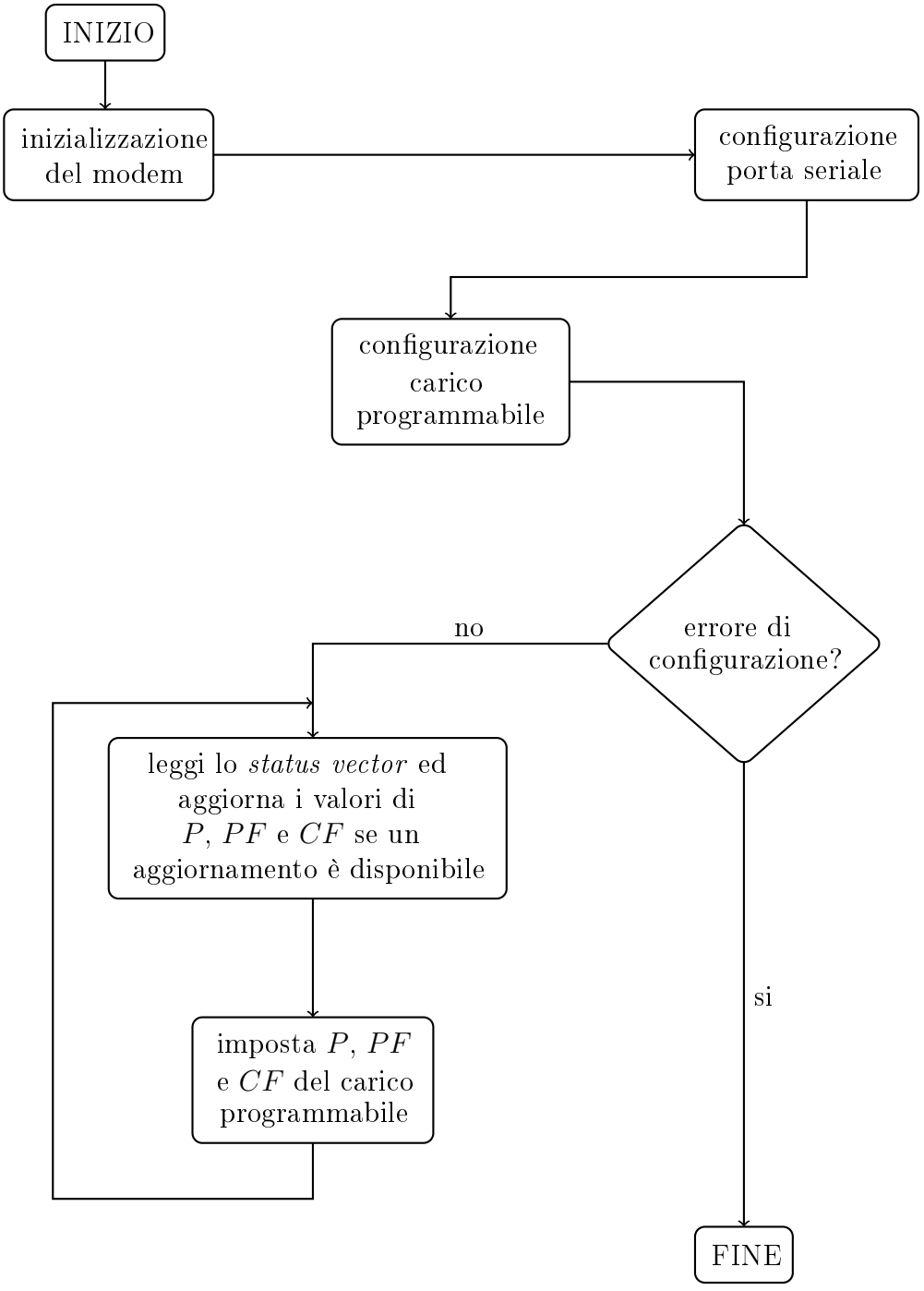

Figura 6.3: Schema di funzionamento del modem master virtuale

6.3 Test con i modem virtuali

Per verificare la correttezza del funzionamanto dei modem slave e master virtuali, sono stati eseguiti alcuni test con una configurazione della rete virtuale riportata nello schema di Figura 5.1 a pagina 74. Si noti che i thread dei modem virtuali sono in esecuzione in un elaboratore separato rispetto a quello dove operano i thread dei dispositivi di tipo Nemo e carico programmabile virtuali ed il thread gestore della rete. I risultati forniti dai test sono gli stessi riportati nelleTabelle 5.1-5.5 a pagg. 77-78 dato che non sono stati apportate modifiche alle topologia della rete elettrica virtuale e il valori dell'impedenza di carico monitorata dal Nemo utilizzati sono esattamente gli stessi.

Appendice A

Note sulla compilazione

Il simulatore di rete sviluppato in questo progetto è composto da 10 file scritti in linguaggio C, strutturati nel modo seguente:

- "serial_port_functions.h" è il file che raccoglie tutte le funzioni necessarie per l'utilizzo della porta seriale;

- "virtual_nemo_functions.h" è il file che contiene le funzioni che sono state implementate per emulare le azioni svolte dallo strumento Nemo D4-L+;

- "virtual_programmable_load_functions.h" è il file che contiene le funzioni che sono state implementate per emulare le azioni svolte dallo strumento Chroma 63804;

- "virtual_electric_network.h" è il file che contiene le strutture dati e le funzioni con cui è possibile simulare, gestire e risolvere la rete elettrica virtuale;

- "network_electric_client.c" è il file che contiene le realizzazioni virtuali dei dispositivi di rete, che sono rappresentati da due threads distinti. Inoltre, nel file è contenuta l'implementazione di un ulteriore thread, denominato Network Manager, che si occupa di gestire la rete elettrica virtuale;

- "server_gtk.c" è il file che contiene la realizzazione di un server grafico, basata sulle librerie Gtk+, dove vengono visualizzati i risultati delle grandezze elettriche presenti su dispositivi virtuali operativi sulla rete elettrica;

- "modem_status.h" è il file che contiene la struttura denominata *status vector*, utilizzata dai modem slave e master per lo scambio (indiretto) di informazioni;

- "virtual_modem_slave_functions.h" è il file che raccoglie le funzioni che sono state implementate per emulare le azioni svolte dal modem PLC slave;

- "virtual_modem_master_functions.h" è il file che raccoglie le funzioni che sono state implementate per emulare le azioni svolte dal modem PLC master;

- "virtual_modems.c" è il file che contine le realizzazioni virtuali delle due tipologie di modem PLC, rappresentati da due threads distinti.

I file elencati sopra possono essere separati in due gruppi distinti:

- un primo gruppo comprendente i files "virtual_electric_network.h", "virtual_nemo_functions.h", "network_electric_client.c", "virtual_programmable_load_functions.h" e "server_gtk.c" che consentono di simulare la rete elettrica virtuale e i dispositivi di rete;

- un secondo gruppo formato dai files "modem_status.h", "virtual_modem_master_functions.h", "virtual_modems.c" e "virtual_modem_slave_functions.h" che consentono di simulare il funzionamento dei modem PLC.

Nei due gruppi non è stato inserito il file "serial_port_functions.h", per il motivo che esso è in realtà utilizzato in entrambi i gruppi, dato che sia i dispositivi di rete che i modem fanno uso della porta seriale per le comunicazioni. E' bene precisare che i due gruppi di file hanno un funzionamento tra di loro autonomo: infatti, è possibile ad esempio, configurare il sistema di test simulando la rete elettrica, i Nemo ed i carichi adattativi, ma impiegare modem PLC reali.

A.1 Preparazione dell'ambiente di test

Prima di poter operare con il simulatore, è necessaria una fase di configurazione dell'ambiente di test. I passi da seguire vengono brevemente descritti di seguito:

1. per prima cosa, si devono predisporre 1/2 elaboratori in cui eseguire il software del simulatore. Il secondo elaboratore è consigliato nel caso si voglia utilizzare la simulazione dei modem virtuali. Le due macchine dovranno essere dotate di un numero di porte seriale adeguato al numero di dispositivi di rete e coppie di modem che si intendono impiegare. Nel caso l'elaboratore non sia dotato di un numero di porte sufficiente, è possibile ricorrere ad adattatori USB-Seriale. Le porte seriali che si vogliono mettere in comunicazione dovranno essere opportunamente connesse con cavi RS-232;

2. in base al numero di dispositivi di rete (Nemo, carichi adattativi) da utilizzare, all'interno della funzione **main()** del file "network_electric_client.c" è necessario predisporre un numero adeguato di strutture dati per i vari dispositivi virtuali, in modo da garantire il funzionamento degli stessi. Per ogni dispositivo è inoltre richiesto di specificare l'eventuale porta seriale che esso utilizzerà durante le comunicazioni, e la relativa velocità (baud-rate);

3. nel caso si utilizzino modem virtuali, anche per essi è necessario predisporre un adeguato numero di strutture dati all'interno della funzione **main()** del file "virtual_modems.c" dove sarà inoltre richiesto di specificare le porte seriali utilizzate ed annessa velocità per ciascun modem virtuale. Si ricordi inoltre che con la versione attuale del software, il funzionamento dei modem virtuali è a coppie, ossia per ciascun modem slave è previsto l'utilizzo di un modem master.

A.2 Compilazione

Dopo aver configurato opportunamente l'ambiente di test, ed in particolare le funzioni **main()** dei due file "network_electric_client.c" e "virtual_modems.c", si può procedere alla fase di compilazione. Con il termine "Elaboratore" ci riferiremo ad un generico elaboratore dove è in esecuzione il software per la simulazione della rete elettrica e i dispostivi virtuali, mentre con "Elaboratore 2" ci riferiremo ad un generico elaboratore dove è in esecuzione il software per la simulazione dei modem PLC. Di seguito sono riporati i principali comandi da utilizzare:

- nell'Elaboratore 1, aprire una nuova finestra di terminale e digitare il seguente comando: *gcc network_electric_client.c -o network_electric_client -lm -pthread* per compilare il file "network_electric_client.c";

- sempre nell'Elaboratore 1, aprire una nuova finestra di terminale e digitare il seguente comando: *gcc server_gtk.c -o gtk_server $(pkg-config --cflags --libs gtk+-2.0)* per compilare il file "server_gtk.c";

- nell'Elaboratore 2, aprire una nuova finestra di terminale e digitare il seguente comando: *gcc virtual_modems.c -o virtual_modems -pthread* per compilare il file *virtual_modems.c*.

A.3 Esecuzione

Al termine della fase di compilazione, è possibile passare alla fase di esecuzione. I comandi utilizzati sono i seguenti:

- nell'Elaboratore 1, nella finestra del terminale in cui si è prima compilato il file "network_electric_client.c", digitare *sudo ./network_electric_client* per eseguire il file *network_electric_client* creato in precedenza. Sono necessari i permessi di superuser considerato l'utilizzo delle porte seriali;

- nell'Elaboratore 1, nella finestra del terminale in cui si è prima compilato il file "server_gtk.c", digitare *./server_gtk* per eseguire il file *server_gtk* creato precedentemente;

- nell'Elaboratore 2, nella finestra del terminale in cui si è prima compilato il file "virtual_modems.c", digitare *sudo ./virtual_modems* per eseguire il file *virtual_modems*. Sono necessari i permessi di superuser considerato l'utilizzo delle porte seriali.

E' importante ricordare che il file *network_electric_client* deve essere eseguito solamente dopo aver mandato in esecuzione il server gtk. Questo funzionamento è dovuto all'utilizzo dei *socket*, pertanto il client, rappresentato dal gestore della rete (che "si trova" nel file *network_electric_client*), non può connettersi al server se il server non è ancora operativo. Il file *virtual_modems* viene di norma mandato in esecuzione qualche secondo dopo aver digitato il comando per eseguire il file *network_electric_client*.

Bibliografia

[1] Microgrid Control Via Powerline Communications: Network Synchronization Field Test With Prime Modules, Massimo Gallina, Michele Tasca, Tomaso Erseghe, Stefano Tomasin, Padova, 2012

[2] Francesco Trentini, Powerline Communications: An Implementation Of A Real Time Control Architecture For Smart Grid, Padova, 2012

[3] Programmable AC/DC Electronic Load 63800 Series Operation and Programming Manual, version 1.2, 2009, `www.chromaate.com`

[4] IME Nemo D4-L+, Multifunction Meter Communication Protocol (MF6H.docx), rev. 1, 2012, `http://www.imeitaly.com/`

[5] IEEE Standard for Information Technology—Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7, 1003.1-2008/Cor 1-2013, 2008

[6] The GTK+ Project `http://www.gtk.org/`