Capitolo 4

Descrizione dell'architettura client-server

4.1 Panoramica del sistema

La realizzazione di un server, così come quella di un client, non può essere considerata come cosa a sé stante. Il progetto di un server e quello di un client sono strettamente correlati, poiché essi dovranno comunicare ottemperando ad una serie di regole concordate, pena il mancato funzionamento di entrambi. Le regole per la comunicazione tra le due parti formano un protocollo. In questo lavoro la progettazione del lato server è naturalmente andata di pari passo con quella del lato client, e per ogni interfaccia di comunicazione è stato necessario creare il relativo protocollo.

Gli aspetti principali della creazione di un protocollo sono la definizione della struttura e del tipo delle informazioni che devono essere comunicate, e la precisazione del modo esatto in cui vengono inviate attraverso una rete

telematica. Solo in questo modo è possibile riceverle ed interpretarle correttamente. Come si è già detto, i protocolli qui realizzati si basano, con la sola eccezione di quelli per il trasferimento dei dati binari relativi alle misure, sul linguaggio di markup XML. Vista l'importanza che rivestono i protocolli di comunicazione in questo progetto, essi sono descritti dettagliatamente in un capitolo a parte (capitolo 5).

Per quanto riguarda l'instaurazione e la gestione della comunicazione tra il lato client ed il lato server, l'elemento chiave del lavoro svolto è stato la programmazione di socket. Un socket rappresenta un canale di comunicazione tra due processi, i quali possono risiedere anche su macchine diverse collegate in rete. In questo progetto si è fatto uso di socket TCP in ambiente Linux, programmati in linguaggio C. La scelta del protocollo TCP è dovuta alle sue ben note proprietà di protocollo *byte-stream* affidabile. Cercando in rete è possibile reperire una gran quantità di informazioni e tutorial sulla programmazione di socket, liberamente disponibili.

L'architettura proposta in questo progetto serve per la gestione remota e centralizzata di più sistemi di misura Metercontroller, ciascuno residente in un diverso MA-LER DS di un dominio MAID, al fine di rendere possibile il monitoring del traffico offerto all'intera frontiera del dominio. In base ai dati così raccolti, eventualmente elaborati mediante algoritmi di predizione statistica, è possibile arricchire il piano di controllo del dominio MAID di funzionalità di traffic engineering, come ad esempio basare l'admission control sull'effettivo carico della rete o sul traffico previsto tramite le stime statistiche.

Una parte del lavoro effettuato è rappresentata dall'aggiunta di un server TCP al Metercontroller, assieme alle funzioni per l'interpretazione dei messaggi e a quelle per la reazione ai messaggi stessi. Questa parte è

descritta nel paragrafo 4.2. Contestualmente alla realizzazione del server, si è creato il modulo che facesse da client, chiamato Client Interface (CI). Esso non è stato realizzato come un'applicazione indipendente, bensì come un blocco da utilizzarsi in un contesto di programmazione multi-thread. Per gestire simultaneamente più moduli CI attivi è stato poi implementato il software Client Interface Manager (CIM), un programma che ha la principale funzione di generare o cancellare dei thread corrispondenti a moduli CI, in risposta ad opportune richieste. Un'altra funzione importante del CIM è quella di smistare i messaggi destinati ai vari Metercontroller che gli vengono inviati al corretto modulo CI, affinché questo provveda all'inoltro. Il CIM non è dotato di un'interfaccia utente, ma di un server TCP, perciò ad esso si accede mediante un client che diremo User Interface (UI), o interfaccia utente. L'interfaccia utente può essere locale (eseguita sulla stessa macchina del CIM) o remota. In entrambi i casi essa può pilotare il CIM; tuttavia, come si vedrà in seguito, soltanto nel primo caso può disporre anche delle misure di traffico.
La realizzazione di una UI grafica ed *user-friendly* non rientra in questo progetto; nel paragrafo 4.5 si forniranno comunque alcune delucidazioni riguardo a come dovrebbe essere progettata. Per le prove funzionali del CIM si è fatto ricorso a un semplice programma, chiamato banalmente *tester*, creato nell'ambito di questa tesi in luogo della UI, come si vedrà nel capitolo 6.

Riepilogando, il Client Interface Manager viene eseguito su una qualsiasi macchina che possa connettersi con i MA-LER DS del dominio MAID. Su questi ultimi è attivo il Metercontroller, e vi è sempre un blocco controller disponibile per la connessione. Si esegue poi la User Interface (sulla stessa macchina del CIM oppure no) e la si fa connettere al CIM. Da questo

momento è possibile impartire ordini al CIM. Ad esempio si può chiedere di attivare un CI, fornendo l'indirizzo IP del MA-LER DS al quale si deve connettere. In seguito, è possibile, sempre tramite la UI, inviare un messaggio al MA-LER DS per il quale si è creato il modulo CI. Sarà sufficiente inviare tale messaggio (che include nuovamente l'indirizzo IP del router di frontiera) al CIM, e quest'ultimo lo passerà al CI connesso a quel MA-LER DS (se ne esiste uno). Il CI inoltrerà il messaggio al blocco controller tramite il control socket. L'architettura complessiva del sistema è mostrata in figura 4.1.

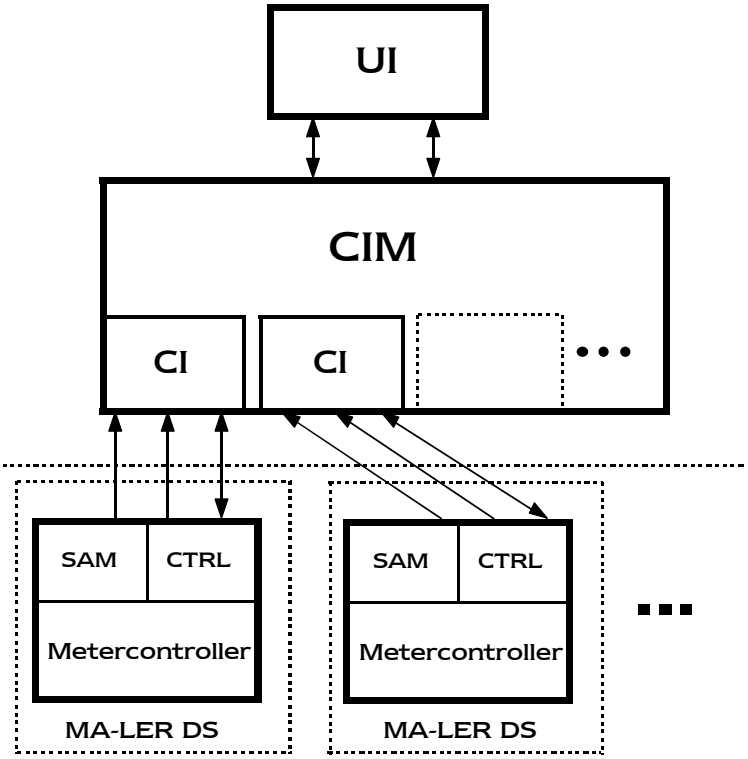

Figura 4.1 – Architettura complessiva

Il CIM funziona quindi come una sorta di demultiplexer per le richieste al Metercontroller, smistandole al corretto CI; inoltre, funziona anche come una specie di multiplexer per le repliche di tutti quanti i Metercontroller che giungono ai CI, dai quali vengono passate sequenzialmente al CIM che le inoltra alla UI.

Ogni CI riceve le informazioni desiderate riguardo al traffico, e crea in memoria un proprio database locale in cui salvarle. Ad ogni ciclo di ricezione il CI copia il database su dei file, uno per ogni LSP e PHB di cui sta ricevendo le misure. Questi file possono essere utilizzati da una UI locale o da un programma di *plot* per mostrare graficamente i dati all'utente.

Si noti, sempre in figura 4.1, come ogni CI abbia tre connessioni TCP private distinte con un Metercontroller. Una di esse serve per i messaggi di controllo, un'altra per il trasferimento dati snapshot, l'ultima (quella verso il sampler) per il trasferimento dati real-time.

Il CIM ed i moduli CI forniscono anche il supporto per le stime statistiche e per i rilevamenti di superamento di soglie sui dati di traffico. Si accede al CIM tramite due connessioni TCP, una per le operazioni di controllo del CIM stesso e dei Metercontroller, l'altra per le operazioni statistiche e di gestione soglie.

Nei prossimi paragrafi verranno esaminati con dettaglio i vari componenti dell'architettura presentata e verrà descritto il loro funzionamento.

4.2 L'interfaccia server del controller

Il sistema di misura Metercontroller registra nel Traffic Database i dati del traffico offerto al MA-LER DS nel quale risiede, suddivisi per LSP e per

PHB. L'utente che prende visione delle misure (per esempio il *network administrator*) potrebbe essere interessato solamente ad una porzione del database: serve quindi un metodo per selezionare le informazioni desiderate. Per di più, è opportuno trasferire tali informazioni in una macchina a parte, nel caso si voglia procedere ad una loro elaborazione (ad esempio applicarvi algoritmi di predizione statistica). Queste esigenze hanno condotto all'idea di un'architettura client-server per accedere ad una parte arbitraria delle misure e trasferire questa porzione su un'altra macchina.

Nel paragrafo 2.3 sono stati descritti i tipi di accesso *real-time* e *snapshot*. L'accesso real-time consiste nel ricevere ad ogni istante di campionamento i dati in tempo reale sulle misure, dal momento della richiesta in poi; quello snapshot consiste nell'avere uno storico di un certo numero di campioni passati, fino a quello attuale. L'invio dei dati snapshot, gestito dal blocco controller, avviene quindi con una singola trasmissione, mentre per i dati real-time si ha l'invio da parte del sampler di un campione ad ogni rilievo delle misure. Si è parlato anche dei due tipi di accesso *full* e *selective*. L'accesso full consente di richiedere in una singola istanza dati relativi a tutti gli LSP presenti nel database; quello selective consente di selezionare singolarmente gli LSP desiderati.

Implementazione del server

In precedenza si è visto come è strutturato il sistema di misura nella sua componente in user space metercontroller. Il programma principale è il sampler, il quale, una volta avviato, genera gli altri thread corrispondenti alla info-unit e ai blocchi controller. È prevista la possibilità che ci siano più client connessi ad un Metercontroller, perciò deve essere sempre possibile accettare la connessione di un client e creare un controller dedicato a tale

connessione. A questo scopo, il sampler compie un cosiddetto *passive open* TCP sulla porta identificata dalla costante simbolica `DEFAULT_PORT` dichiarata in `ctrl.h` e attualmente fissata al valore 29000. Ciò significa che crea ed inizializza un socket rappresentato da una variabile globale, chiamato *start* socket, che viene posto in stato di ascolto per eventuali connessioni su quella porta. Dopodiché il sampler crea il primo blocco controller, che accetta una connessione sullo start socket. Ogni volta che viene accettata la richiesta di connessione di un client viene creato un nuovo socket, una variabile privata che la rappresenta. Esso diviene il *control* socket per quel dato controller, ossia il canale di comunicazione delle informazioni di controllo, come si vedrà meglio tra breve. Lo start socket rimane invece disponibile per ricevere altre connessioni. Dopo che ha accettato una connessione, il controller crea un altro thread controller che attende di accettare una nuova connessione sullo start socket. In questo modo è sempre possibile accettare nuovi client, e ci saranno sempre tanti thread controller attivi quanti sono i client connessi, più uno che attende nuove connessioni.

Una volta che il controller ha accettato una connessione e si è clonato, parte la procedura per completare l'interfaccia di comunicazione con il client. A questo stadio è presente soltanto il control socket. Dapprima c'è uno scambio di *hello* sul control socket per il riconoscimento dell'entità che si è connesssa; se questo scambio non è rispettato, il controller segnala errore ed esce. Questa procedura può essere in futuro ampliata per comprendere una vera e propria autenticazione con gli strumenti adeguati, per garantire anche diversi livelli di privilegio nell'accesso ai comandi del controller. Ad esempio, solo un utente autorizzato dovrebbe essere in grado di cambiare i parametri *window* e *period* di funzionamento del controller, poiché questa variazione causa l'azzeramento del Traffic DB.

Tornando al completamento dell'interfaccia, dopo lo scambio di hello il client (che ricordiamo essere rappresentato da un modulo Client Interface), compie a sua volta un passive open su una porta random. Questo passive open servirà per creare le due connessioni dati. Il CI comunica il numero di porta al controller tramite il control socket, ed il controller crea due socket e li connette a quella porta. Essi sono lo *snapshot* socket ed il *real-time* socket e vengono inizializzati come socket non bloccanti[1]. Se per qualsiasi motivo questa procedura non va a buon fine, il controller segnala errore ed esce.

Se l'operazione viene completata, ci sono tre connessioni TCP attive tra controller e CI. A questo punto il controller invia sul control socket le informazioni sul suo stato attuale, che comprendono i valori di window e period, la dimensione del database, la lista degli LSP e dei PHB attualmente presenti nel router e di cui si stanno rilevando i dati di traffico. Questo tipo di informazioni può essere richiesto successivamente dal CI in un qualsiasi momento.

Fatto questo, il controller entra nel suo ciclo principale di funzionamento, schematicamente descritto in figura 4.2 a pagina seguente.

Si parte dall'attesa di un comando sul control socket; i comandi utilizzano il linguaggio XML, e sono di lunghezza variabile. Per poter ricevere un messaggio correttamente su un socket, ci sono tre possibili metodi: il messaggio deve essere di lunghezza fissa e prestabilita, deve essere delimitato da marker conosciuti, oppure deve indicare quanto è lungo. Si è scelta la terza soluzione, più flessibile ed adatta allo scopo. Pertanto, l'attesa di un messaggio consiste nella ricezione di un intero (di dimensione costante) che indica quanti byte è lungo il successivo messaggio XML, permettendone

[1] Un socket in modalità non bloccante tenta di inviare e ricevere dati senza controllare se ne ha realmente la possibilità, con il rischio di perderne una parte. L'utilità consiste nella garanzia di non dover attendere per le operazioni di lettura e scrittura.

così la corretta ricezione. Una volta ricevuto un messaggio, se ne controlla la validità. Questo implica la conformità alla DTD ed il rispetto di alcune regole logiche per i valori degli elementi dovute al funzionamento proprio del controller. Se un comando non è valido viene scartato, quindi viene inviata una risposta contenente un codice di errore, infine si torna all'attesa di un messaggio. I comandi validi che riceve il controller sono di due tipi: da eseguire immediatamente o da accodare in una lista. Se il comando è da eseguire subito, viene eseguito e si torna all'attesa di un nuovo comando; altrimenti, si mette il messaggio nella lista corretta tra le due possibili (come si dirà tra poco) e si torna all'attesa. In entrambi questi casi, viene inviata una risposta che indica l'avvenuta corretta elaborazione del messaggio.

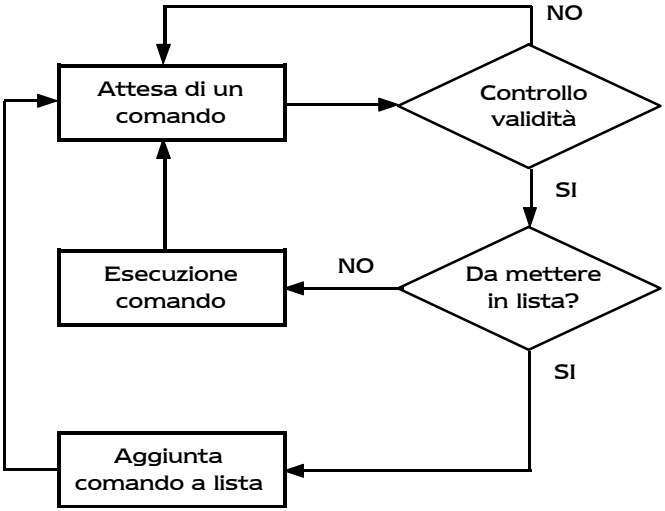

Figura 4.2 – Ciclo principale di funzionamento del controller

Con un opportuno comando si può richiedere l'uscita del controller, che, terminato il ciclo principale, reagisce chiudendo i socket, liberando le

risorse utilizzate ed infine uscendo dall'esecuzione. L'uscita di un controller non influenza il funzionamento del sistema di misura.

I comandi per il controller

I comandi che riceve il controller possono essere suddivisi in due tipi, come detto in precedenza. Il primo tipo comprende i messaggi che devono essere messi in lista. Si tratta di richieste di accesso oppure di disconnessione dal Traffic DB. Le richieste di accesso vanno in una lista diversa da quella delle richieste di disconnessione. Ogni messaggio che rientra in questa categoria viene accodato nella lista corrispondente. All'arrivo di uno specifico comando le liste vengono eseguite ed azzerate.

I comandi della prima lista sono quelli di accesso al Traffic DB:

- **Full connection:** comprende sia l'accesso snapshot che quello real time. Consente infatti di scaricare l'intero DB fino al campione attuale (inclusa cache) e connettersi alle misure in tempo reale, per tutti gli LSP (accesso full).
- **Full snapshot:** l'accesso è solo snapshot. Consente di scaricare l'intero DB (accesso full) fino al campione attuale (inclusa cache) e campione attuale, senza connettersi alle misure in tempo reale.
- **Selective connection:** accesso sia real-time che snapshot. Per un solo LSP (accesso selective), consente di scaricare lo storico e connettersi alle misure in tempo reale; può essere ripetuto in seguito per altri LSP, ma in una singola richiesta deve essercene uno soltanto. Questo comando non è lecito se è già attivo un accesso di tipo real-time full.
- **Selective snapshot:** accesso soltanto di tipo snapshot. Per un solo LSP (accesso selective), consente di scaricare lo storico inclusi cache e campione attuale, senza connettersi alle misure in tempo reale; può

essere ripetuto per altri LSP, ma in una singola richiesta deve essercene uno soltanto.

- **Full last sample:** consente di scaricare l'ultimo campione per tutti gli LSP (accesso snapshot e full).
- **Selective last sample:** consente di scaricare l'ultimo campione per un LSP (accesso snapshot e selective); può essere ripetuto per altri LSP in altre richieste.
- **Full N most recent samples:** l'acccesso è di tipo snapshot e full; consente di scaricare gli ultimi N campioni per tutti gli LSP, inclusi cache e ultimo campione.
- **Selective N most recent samples:** l'accesso è di tipo snapshot e selective. Per un LSP, consente di scaricare gli ultimi N campioni, inclusi cache e ultimo campione; può essere ripetuto in successive richieste per altri LSP.
- **Full simple connection:** l'accesso è di tipo real-time e full. Consente di connettersi alle misure in tempo reale (senza avere lo storico) per tutti gli LSP.
- **Selective simple connection:** l'accesso è di tipo real-time e selective. Consente di connettersi alle misure in tempo reale (senza avere lo storico) per un LSP; può essere ripetuto per altri LSP. Questo comando non è lecito se c'è già una connessione di tipo real-time full.

I comandi della seconda lista sono quelli relativi alla disconnessione, e riguardano solamente l'aspetto real-time dell'accesso al database:

- **Full delete:** consente di disconnettersi dall'accesso real-time selective alle misure per tutti gli LSP per cui si è attualmente connessi, oppure dall'accesso real-time full. Si noti che questa è l'unica maniera di annullare quest'ultimo tipo di accesso.

- **Selective delete:** consente di disconnettersi dalle misure per un LSP per cui si è attualmente connessi con accesso real-time. Questo comando non è lecito se è presente un accesso di tipo real-time full.

I comandi di immediata esecuzione non sono accodati in alcuna lista ma vengono eseguiti non appena si presentano.

- **Change window:** consente di modificare il parametro window, ossia la finestra temporale per il campionamento.
- **Change period:** consente di modificare il parametro period, ossia l'intervallo tra due successivi istanti di campionamento.
- **Change window and period**: consente di variare contemporaneamente entrambi i parametri.
- **Info:** consente di ottenere immediatamente informazioni sugli attuali parametri di funzionamento del controller (window, sample rate, lista dei PHB, dimensione del database) e sugli LSP attivi.
- **Clear:** consente di svuotare le due liste dei task in attesa di esecuzione, pertinenti ai messaggi di accesso alle misure e disconnessione. Questo messaggio non è valido se le liste sono vuote.
- **Commit:** dà il via all'esecuzione dei task delle due liste, innescando una procedura che sarà descritta tra poco. Questo messaggio non è valido se le liste sono vuote.
- **Stop:** blocca l'esecuzione dei messaggi in lista e successivamente svuota le liste; tra breve sarà descritto con maggior dettaglio.
- **Abort:** questo messaggio causa la chiusura del controller: annulla qualsiasi task attivo, svuota le liste ed abbatte la connessione tra il modulo CI ed il controller.

Vale la pena di esaminare con dettaglio cosa accade se ci sono dei task in lista ed arriva il messaggio commit. Per prima cosa viene creato un thread,

detto *stop thread*, che è sostanzialmente una replica del ciclo principale del controller. Esso ha la funzione di ricevere ulteriori messaggi mentre il controller vero e proprio è occupato ad eseguire i task in lista. A differenza del controller, lo stop thread rifiuta e scarta tutti i messaggi con l'eccezione dei messaggi stop ed abort. Il messaggio stop serve proprio per terminare anzitempo l'esecuzione dei task in lista. Dopo aver creato lo stop thread, il controller chiama la funzione `task_executor()`, funzione già implementata nel sistema di misura, che prende in esame le liste (prima quella di accesso, poi quella di disconnessione) ed avvia la loro esecuzione. Tra le varie cose, la `task_executor()` riceve come parametri di chamata i descrittori dei socket real-time e snapshot. La funzione accede al Traffic DB per l'invio degli eventuali dati di snapshot sullo snapshot socket, e modifica la DescriptorList affinché il sampler possa venire a conoscenza del fatto che il client è interessato alle misure real-time ed inviare così da quel momento le misure ad ogni ciclo di campionamento. Le informazioni inserite nella DescriptorList sono il descrittore del socket real-time e gli LSP per i quali inviare le misure in tempo reale a quel descrittore.

Quando la `task_executor()` ritorna, viene cancellato lo stop thread, quindi si controlla se tutti i task sono andati a buon fine e si notifica il loro esito al CI con una replica. Nel frattempo, il CI comincia a ricevere sui socket snapshot e real-time le misure di traffico sotto forma di dati binari. I protocolli per il trasferimento dei dati binari sono descritti nel prossimo capitolo, paragrafo 5.3.

La funzione utilizzata per l'invio delle repliche nel controller si chiama `send_xml_reply()`. Le funzioni sono contenute nel file `ctrl_f.c`, mentre le dichiarazioni e le costanti simboliche si trovano in `ctrl.h`.

4.3 Il Client Interface Manager (CIM)

In questo paragrafo si affronta in modo particolareggiato la descrizione del Client Interface Manager, il programma che permette la gestione simultanea di più moduli Client Interface. Si tratta di un'applicazione multi-thread, costituita da un blocco principale al quale sono demandate le seguenti funzioni:

- Presenza di un server con due connessioni TCP per l'accesso alle operazioni di controllo e di stima statistica con soglie.
- Creazione su richiesta di un modulo CI, tramite l'attivazione di un thread.
- Cancellazione su richiesta di un modulo CI, mediante l'eliminazione del thread relativo.
- Inoltro su richiesta di un messaggio ad un controller (demultiplexing), smistandolo al corretto CI.
- Inoltro verso la UI di tutti i messaggi di risposta provenienti dai controller (multiplexing).
- Inoltro ad un modulo CI di una richiesta nell'ambito delle operazioni di stima statistica o di gestione delle soglie per le misure o i predittori.

Come si può vedere nel precedente elenco, tra le funzionalità del CIM non è presente un'interfaccia diretta con l'utente, ma un'architettura server per certi aspetti analoga a quella presente nel controller. Questo significa che per pilotare il CIM è necessario un ulteriore blocco, chiamato UI (User Interface), che costruisca ed invii dei messaggi al CIM secondo il protocollo XML definito, ed in base alla richieste dell'utente. La UI può essere remota o locale rispetto al CIM. Tuttavia, se nella UI si vuole comprendere una

visualizzazione grafica delle misure di traffico ottenute dai moduli CI, la UI dovrà essere locale, poiché non è previsto l'ulteriore trasferimento di dati di traffico dal CIM alla UI stessa. Le misure si troveranno infatti su dei file, nella stessa macchina dove viene eseguito il CIM.

4.3.1 Struttura del CIM

Il Client Interface Manager è un programma multithread in cui i vari thread necessitano di un cospicuo e continuo scambio reciproco di informazioni. Il codice è dislocato su diversi file, ciascuno pertinente ad un componente diverso del programma. Il listato della funzione `main()` si trova nel file `cman.c`; il modulo Client Interface si trova nel file `c_int.c`. Tutte le dichiarazioni di funzioni e strutture e gli include sono nel file `clnt.h`; le variabili comuni sono invece dichiarate in `comvar.h`. Le altre funzioni sono raccolte nel file `clnt_f.c`, mentre i parser real-time e snapshot, di cui si parlerà nel paragrafo 4.4.2, sono rispettivamente in `r_t.c` e `snap.c`.
Per rendere possibile lo scambio di informazioni tra i diversi thread si è deciso di utilizzare una serie di strutture visibili globalmente in cui ogni thread possa scrivere o dalle quali ogni thread possa ricavare i dati desiderati. L'accesso alle strutture comuni deve in qualche modo essere arbitrato, onde evitare l'accesso concomitante di due thread alla stessa struttura, che causerebbe un blocco del programma. A questo scopo si è fatto uso dei *mutex* (*MUTual EXclusion locks*), dei semafori per i thread in ambito GNU/Linux. Un mutex è un semaforo che concede l'accesso ad un thread alla volta, mettendo gli altri in attesa. È sufficiente quindi regolare l'accesso alle strutture comuni con un mutex per avere la garanzia che soltanto un thread alla volta cercherà di accedervi. Nel file `comvar.h` i mutex sono

dichiarati immediatamente prima delle variabili comuni alle quali regolano l'accesso.

Il numero massimo di CI contemporaneamente attivi è fissato in fase di compilazione mediante la costante simbolica `MAX_INTERFACES`, definita in `clnt.h`. A livello concettuale, il CIM è concepito con un numero di slot pari a `MAX_INTERFACES` per ospitare altrettanti CI. Nelle variabili comuni sono definiti dei vettori di `MAX_INTERFACES` strutture, che vengono utilizzati man mano che vengono creati nuovi CI.

La struttura più importante di cui si dichiara il vettore è quella che contiene le informazioni di funzionamento dei moduli Client Interface, ed è definita come segue (la definizione si trova in `clnt.h`):

```
struct clnt_interface_info
        {
        int busy;
        pthread_t interf_id;
        struct in_addr ctrl_ad;
        int t_write;
        char *task_dialog;
        int r_read;
        char *reply_dialog;
        int est_write;
        char *est_dialog;
        };
```

Il vettore di `MAX_INTERFACES` strutture di questo tipo è dichiarato in `comvar.h`, con il nome di `cii`:

```
struct clnt_interface_info cii[MAX_INTERFACES];
```

Ogni elemento di questo vettore rappresenta uno slot per la creazione di un modulo CI. Il campo `busy` è un flag che indica se lo slot è libero (0) oppure

già occupato da un CI (1). Quando tutti gli slot sono occupati, non è possibile creare altri CI, a meno di non cancellarne prima uno. Nel campo `interf_id` viene memorizzato il *tid* del modulo CI che va ad occupare quello slot, mentre nel campo `ctrl_ad` si registra l'indirizzo IP del controller al quale il modulo CI si deve connettere. Gli altri campi servono per le operazioni di multiplexing e demultiplexing della messaggistica. I puntatori a carattere vengono allocati dinamicamente durante l'esecuzione del programma per ospitare i messaggi XML di lunghezza variabile che transitano dal CIM ai moduli CI e viceversa. Ogni puntatore è accompagnato da un flag che indica la presenza (1) o meno (0) di un messaggio da inoltrare. I campi `t_write` e `task_dialog` riguardano i messaggi diretti ai controller. I campi `r_read` e `reply_dialog` sono pertinenti ai messaggi provenienti ai controller e diretti alla UI. Infine, i campi `est_write` e `est_dialog` servono per le richieste nell'ambito delle operazioni di stima, ricevute sullo *estimate* socket.
Esistono altri vettori globali di `MAX_INTERFACES` elementi, utilizzati per gestire in modo pulito la cancellazione o l'uscita dei thread CI. Essi sono tutti dichiarati in `comvar.h`, assieme ai mutex per la loro gestione.

Il Client Interface Manager ha una struttura di server TCP. All'avvio si compie un passive open sulla porta fissata in fase di compilazione mediante la costante simbolica `MASTER_PORT` (attualmente il valore è 28000). Il socket in ascolto ha il descrittore di nome `sock_start`. Si attendono quindi due connessioni da parte della UI, delle quali la prima rappresenta il *master* socket, la seconda lo *estimate* socket. I descrittori relativi si chiamano nel CIM rispettivamente `sock_cgi` e `sock_est`. Il primo è una variabile privata della funzione `main()` del CIM, poiché essa soltanto vi deve leggere e scrivere; il secondo è invece una variabile globale. Mentre soltanto la

funzione `main()` legge sullo estimate socket, quasi tutti i thread devono avere la possibilità di scriverci, come si vedrà in seguito. L'accesso al descrittore `sock_est` è quindi regolato da un mutex. Nel caso la procedura delle connessioni non vada a buon fine, il CIM segnala errore ed esce. Una volta instaurate le connessioni TCP, il CIM esegue delle inizializzazioni e poi entra nel ciclo principale di funzionamento, descritto nel paragrafo 4.3.5, che consiste in una sequenza di operazioni ripetuta periodicamente, senza far uso di attese attive. La struttura complessiva del CIM è mostrata in figura 4.3.

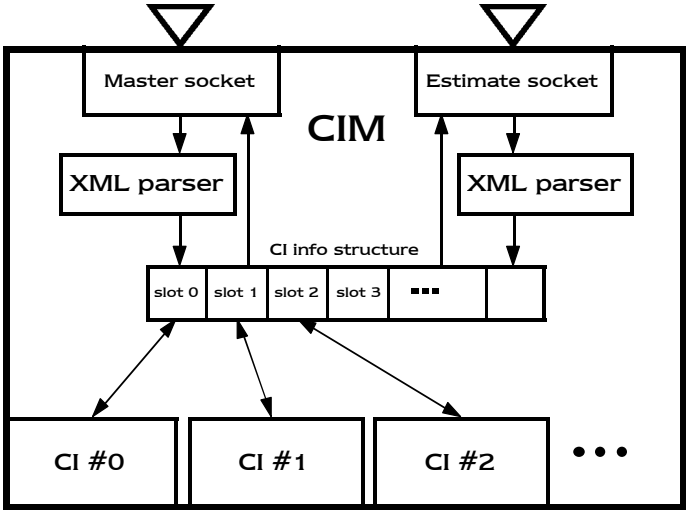

Figura 4.3 – Struttura del Client Interface Manager

4.3.2 Interfacce di controllo

Le interfacce di controllo del Client Interface Manager sono i due socket TCP master ed estimate. L'accesso ad entrambi prevede l'utilizzo del

linguaggio XML, anche se ad ogni socket compete un differente protocollo con messaggi di tipo diverso. Come si è spiegato per il server TCP del controller, per ricevere un messaggio su un socket è necessario conoscerne in qualche modo l'estensione. Anche nel CIM si è scelto di utilizzare il metodo di far precedere all'invio del messaggio l'invio di un intero che ne indichi la lunghezza. Questa è la soluzione generalmente migliore, specialmente nei casi come questo, in cui i messaggi possono essere di lunghezza variabile. Questo espediente è utilizzato in entrambi i socket ed in entrambi i sensi di trasmissione: il CIM si aspetta di ricevere prima un intero (la cui dimensione è fissa), poi un messaggio di lunghezza in byte pari al valore dell'intero. Analogamente, ogni volta che il CIM invia un messaggio, invia prima un intero indicante la lunghezza del messaggio stesso. Quanto detto vale sia per il master che per lo estimate socket. Si noti che il progetto della UI deve obbligatoriamente tener conto di queste regole.

La ricezione e l'invio sui due socket di accesso al CIM vengono effettuati in maniera non bloccante. Per quanto riguarda la ricezione sul master socket, viene utilizzata la funzione `select()` con un timeout configurabile in fase di compilazione, mediante le costanti `RECV_WAIT_SEC` (per i secondi) e `RECV_WAIT_USEC` (per i microsecondi), definite in `clnt.h`. Questo meccanismo di timeout è cruciale per tutto il funzionamento del CIM, in quanto scandisce la temporizzazione del suo ciclo principale. Il programma sta in attesa (non attiva) sulla `select()` per la durata del timeout, quindi passa oltre. Nel caso ci sia un messaggio in arrivo sul master socket, la `select()` segnala il fatto e si procede alla ricezione. Esiste un meccanismo di timeout anche per il completamento del messaggio in corso di ricezione, implementato nella funzione `nonblocking_receive_n()`, nel file `clnt_f.c`. Scaduto il timeout (sempre rappresentato dalle due costanti simboliche `RECV_WAIT_SEC` e `RECV_WAIT_USEC`), la ricezione non viene completata ed il

CIM segnala un errore, sia localmente che alla UI. Il motivo del timeout in ricezione è il fatto che il CIM, per poter assolvere correttamente alle proprie funzioni, non deve trovarsi ad attendere indefinitamente il completamento di un messaggio.

Quando la ricezione di un messaggio non va a buon fine, sia nel caso di timeout scaduto che nel caso di errori della funzione `recv()`, parte un meccanismo che tenta di recuperare il funzionamento dell'interfaccia, potenzialmente compromesso dall'errore. Nel caso di timeout scaduto a causa di ritardi nella consegna di una parte del messaggio, ci si deve aspettare che la parte mancante arrivi da un momento all'altro, facendo perdere il sincronismo lunghezza – messaggio. La procedura di recupero, assolutamente empirica e senza pretesa di rigore, è implementata nella funzione `descriptor_recover()`, di cui si fornisce qui il prototipo:

```
void descriptor_recover(int desc, int s, int us);
```

I parametri di ingresso sono il descrittore del socket da recuperare `desc`, un numero di secondi `s` e di microsecondi `us` per costruire un timeout. La procedura consiste nell'attesa di un tempo pari a `s` secondi e `us` micro-secondi, quindi allo svuotamento totale del buffer di ricezione TCP. Dal momento che l'inizio della procedura viene segnalato alla UI, questa può cessare di trasmettere per un tempo sufficientemente maggiore del timeout, nella speranza che il canale di comunicazione risulti ripulito e si possa riprendere col normale protocollo.

Per quanto riguarda la scrittura sul master socket da parte del CI, anche questa è configurata non bloccante, tramite l'apposito flag `MSG_DONTWAIT` della funzione `send()`. Questo significa che una UI non sufficientemente veloce può perdere alcune repliche del CIM.

Il funzionamento dello estimate socket è del tutto analogo a quello del master socket. L'unica differenza è la temporizzazione in ricezione: la funzione `select()` in questo caso è impostata con un timeout nullo, ossia ritorna subito. Si procede alla ricezione soltanto se essa segnala la presenza di un messaggio in arrivo.

Entrambi i descrittori sono dotati di un meccanismo che consente loro di capire se l'altro capo della connessione è stato chiuso, per evitare eventuali errori. In tal caso, poiché sarebbe impossibile continuare il normale iter operativo, il CIM reagisce abbattendo tutti i moduli CI ed uscendo.

I comandi possibili per il master socket sono i seguenti:

- **Create CI**: assieme a questo comando si specifica l'indirizzo IP di un MA-LER DS sul quale è residente ed attivo il Metercontroller, ed il CIM crea un thread CI facendolo connettere al controller specificato. Non è possibile creare all'interno di un CIM due CI verso lo stesso controller. Ad una simile richiesta il CIM reagirà segnalando errore.
- **Delete CI**: assieme a questo comando si specifica l'indirizzo IP di un MA-LER DS sul quale è residente ed attivo il Metercontroller. Se c'è un CI connesso a quel controller, esso verrà cancellato, altrimenti il CIM segnalerà errore.
- **Forward task**: assieme a questo comando si specifica l'indirizzo IP di un MA-LER DS sul quale è residente ed attivo il Metercontroller. Se c'è un CI connesso a quel controller, il CIM gli passerà il messaggio corrente affinché sia inoltrato al controller. Se non c'è un CI connesso al controller desiderato, il CIM segnalerà errore.
- **Close**: in risposta a questo comando il CIM cancella tutti i CI attivi ed esce.

Per i comandi dello estimate socket si rimanda alla sezione relativa alle operazioni di stima statistica, paragrafo 4.3.4.

L'invio di segnalazioni, notifiche e repliche del CIM alla UI è demandato alla funzione `cm_notify()`, della quale si riporta il prototipo e si illustra il funzionamento:

```
int cm_notify(int desc, int code, int id, char *ip);
```

I parametri in ingresso sono il descrittore del master socket (`desc`), il codice di replica `code` (che indica il significato del messaggio), il *request id* (vedi capitolo 5) del messaggio al quale si sta replicando (`id`), ed un puntatore ad una stringa contenente l'indirizzo IP del controller di interesse per la replica (`ip`). Se non si desidera che venga riempito il campo dell'indirizzo IP, è sufficiente passare `NULL`. La funzione crea un messaggio XML coi dati in ingresso e lo invia in ottemperanza al protocollo sul descrittore specificato. Restituisce 0 in caso di successo, -1 in caso di errore.

4.3.3 Il parser XML

I comandi ricevuti dal Client Interface Manager sui socket master ed estimate sono in realtà dei documenti XML, dai cui elementi il CIM deve estrarre le informazioni necessarie a compiere l'opportuna operazione. Il parsing dei messaggi XML è demandato a due funzioni, una creata per il master socket – `xml_request_parseandvalid()` – e l'altra per lo estimate socket – `estimate_request_parseandvalid()` – , che provvedono anche alla convalida secondo DTD. È possibile trovarle nel file `clnt_f.c`.
Le funzioni contengono l'elemento XML di dichiarazione `DOCTYPE`, con la

definizione del modello di documento secondo DTD. Ad esso appongono il messaggio XML (che quindi deve essere inviato dalla UI privo del tag `DOCTYPE`), quindi procedono con il parsing. Nel caso di errori di parsing o di convalida il messaggio sarà scartato ed il CIM segnalerà un errore sia localmente che alla UI.

Le informazioni sono ricavate dal contenuto degli elementi mediante le funzioni di conversione da stringhe, presenti nelle librerie standard. Si eseguono anche dei controlli di congruenza sui valori riscontrati, come ad esempio che il codice operativo sia tra quelli previsti o che un indirizzo IP sia valido. Per le istruzioni dettagliate sui messaggi XML da inviare sui due socket e sulle notifiche inviate in risposta da parte del CIM si rimanda al capitolo 5, riguardante i protocolli di comunicazione.

4.3.4 Il supporto per le operazioni di stima

Il Client Interface Manager fornisce un supporto flessibile per le operazioni di stima statistica sui dati di traffico ricevuti dai moduli CI. L'idea è quella di poter supportare qualsiasi tipo di stimatore modellabile con un'opportuna funzione. Ogni stimatore viene gestito come plug-in, ed il CIM all'avvio genera una lista di tutti gli stimatori che correntemente supporta.

A mettere in atto le operazioni di stima sono però i moduli CI, che hanno accesso ciascuno al proprio database dove vengono salvati i dati di traffico ricevuti dal metercontroller. Il CIM provvede a passare i messaggi ricevuti sullo estimate socket al CI indicato dall'indirizzo IP del controller al quale è connesso, sempre che ci sia effettivamente un CI connesso a quel controller. In caso contrario, il CIM riporterà errore sia localmente che alla UI (sullo estimate socket).

I comandi possibili nell'ambito delle stime statistiche e della gestione delle soglie sono i seguenti:

- **Estimators list**: con questo comando si richiede al CIM di fornire la lista degli stimatori supportati, assieme alle informazioni necessarie per il loro utilizzo. Il CIM stesso replica a questa richiesta.
- **Add estimator**: questo messaggio viene passato dal CIM ad un modulo CI per il quale si fornisce l'indicazione dell'indirizzo del controller a cui è connesso. Si richiede di creare uno stimatore di un certo tipo fra quelli supportati e di agganciarlo alle misure di un dato LSP. Nel messaggio si specificano i parametri di funzionamento per lo stimatore stesso. Il CIM segnala errore se non esiste un CI attivo verso il controller richiesto, notifica il successo dell'inoltro se il passaggio del messaggio al CI va a buon fine. Il CI poi provvede a compiere le dovute operazioni (vedi paragrafo 4.4.4), tra le quali invia alla UI un messaggio contenente un identificatore univoco (una stringa) che serve a indicare il particolare stimatore appena creato in successive richieste.
- **Remove estimator**: fornendo l'identificatore univoco di uno stimatore agganciato in precedenza, se ne richiede la rimozione. Questo messaggio è preso in consegna dal corretto modulo CI.
- **Adjust estimator**: fornendo l'identificatore univoco di uno stimatore agganciato in precedenza, consente di variarne i parametri di funzionamento. Questo messaggio è preso in consegna dal corretto modulo CI.
- **Set threshold**: in questo messaggio si deve specificare al solito l'indirizzo di un controller verso il quale sia attivo un modulo CI, e si richiede a quel CI di configurare le soglie per le misure su uno LSP. Per disattivare il meccanismo delle soglie è sufficiente passare un valore negativo.
- **LSP estimator/threshold info**: con questo messaggio si richiedono le

informazioni per un dato LSP riguardo alle soglie fissate per le sue misure e agli eventuali stimatori ad esso agganciati.

Per quanto riguarda l'invio dei messaggi sul socket estimate da parte del CIM o dei moduli CI, esiste una funzione che copre tutte le possibili casistiche di messaggi e provvede al loro invio. La funzione si chiama `est_notify()` e si trova in `clnt_f.c`. Il prototipo è il seguente:

```
int est_notify(int code, int id, char *ip, unsigned long lsp,
               unsigned long phb, char *data);
```

Mediante questa funzione viene creato ed inviato un messaggio XML (preceduto al solito da un intero indicante la sua lunghezza) secondo il protocollo di notifica delle operazioni di stima e gestione soglie. I dati in ingresso sono il codice di notifica `code`, che determina il significato della notifica; il request id della richiesta alla quale si sta replicando (`id`); un puntatore ad una stringa contenente l'indirizzo IP che caratterizza il CI dal quale parte la notifica (se necessario, altrimenti si passa `NULL`), gli identificatori `lsp` e `phb` di LSP e PHB di interesse (vedi paragrafo 5.5); un puntatore ad una stringa dove devono trovarsi eventuali dati aggiuntivi (vedi paragrafo 5.5.2). L'accesso a questa funzione deve essere regolato dal mutex `estimate_mutex`, poiché essa fa uso del descrittore dello estimate socket, che è una variabile globale. Il motivo consiste nel fatto che tutti i blocchi del Client Interface Manager devono scrivere sullo estimate socket. Il thread principale del CIM infatti vi notifica eventuali errori di ricezione o parsing dei messaggi, oppure errori dovuti al fatto che non esiste il CI richiesto (si ricorda che un CI viene sempre caratterizzato univocamente nei messaggi dall'indirizzo della macchina sulla quale risiede il Metercontroller

al quale è connesso). I moduli CI devono invece notificare il successo o gli errori nelle operazioni vere e proprie, come l'agganciamento o la rimozione di uno stimatore. Le notifiche del superamento di una soglia sono invece gestite dai parser dei dati binari, che sono dei thread figli del thread CI, come si vedrà nel paragrafo 4.4.

4.3.5 Il ciclo principale

Il Client Interface Manager ha un funzionamento ciclico, temporizzato dal *polling* per la ricezione sul master socket, come si è detto nel paragrafo 4.3.2. Il valore del timeout dovrebbe essere scelto in fase di compilazione in modo da conciliarsi con i valori del periodo di campionamento che si intende utilizzare nei controller, e con la rapidità con la quale si prevede di richiedere le operazioni al CIM. Ad ogni ciclo si elabora infatti una singola richiesta XML sul master e sullo estimate socket. Valori del timeout inferiori ad un secondo sono senz'altro ragionevoli, considerando che la fase di richiesta operazioni è comunque sporadica e probabilmente concentrata all'inizio della connessione. Il valore impostato in fase di progetto è pari a 200 millisecondi, e si è dimostrato un compromesso più che soddisfacente tra utilizzo della CPU (il CIM risulta praticamente ininfluente) e velocità di opera. Si noti tuttavia che si è utilizzato per l'invio delle richieste una UI costituita da un *tester* con input da tastiera per la creazione dei messaggi XML, il che implicava un intervallo di alcuni secondi tra due richieste successive. Nell'ottica di una UI grafica, dove un clic su un pulsante sia un *trigger* per numerosi comandi inviati contemporaneamente, potrebbe forse essere necessario un valore minore (anche se si ritiene che con buona probabilità le circa cinque richieste al secondo

smaltite andranno ancora bene). Per di più, si noti che le richieste vengono accodate nei buffer di ricezione TCP e non vanno perdute. Per il funzionamento proprio del protocollo TCP, esiste un controllo di flusso che impedisce l'invio di dati nel caso il buffer di ricezione dall'altro capo non sia in grado di accoglierli[2]. Sarà quindi sufficiente gestire opportunamente le procedure di invio dal lato UI, controllando che il buffer di trasmissione non sia pieno, per aver la garanzia di non perdere messaggi.

Le operazioni compiute in un ciclo di funzionamento del Client Interface Manager sono le seguenti:

- Inizializzazione o pulizia delle strutture che ospitano dati sui messaggi ricevuti e delle variabili che sovrintendono al funzionamento del ciclo.
- Ricezione non bloccante sul master socket, tramite utilizzo della funzione `select()`: il processo sta in fase di *sleeping* per la durata del timeout, a meno che non arrivi un messaggio, nel qual caso procede alla ricezione, al parsing ed alla convalida, e la variabile `operation` viene settata al valore relativo all'operazione da svolgere. Per questi valori sono utilizzate delle costanti simboliche definite in `clnt.h`. Se non vi sono errori di ricezione o convalida, il CIM procede, a seconda del valore della variabile `operation`, con l'esecuzione dell'operazione opportuna. In seguito all'esecuzione, notifica sia localmente che alla UI il successo o eventuali errori.
- Scansione del vettore di informazioni sui CI per capire se ci sono messaggi provenienti da qualche controller che devono essere inoltrati alla UI; se ve ne sono, il CIM provvede all'inoltro in modalità non bloccante, senza però controllare l'effettiva possibilità di invio (se la UI non è pronta alla ricezione, il messaggio va perduto, poiché non è stato

[2] Si tratta del meccanismo di *sliding window* TCP per il *flow control*.

previsto un metodo per accodare più messaggi in attesa di inoltro di un solito controller).

- Ricezione non bloccante sullo estimate socket, che funziona in modo analogo a quella sul master socket, con la differenza che il timeout impostato per la `select()` è nullo. La temporizzazione del ciclo è quindi determinata solo dal polling sul master socket. Se la funzione `select()` segnala che il descrittore è pronto in lettura, si procede alla ricezione del messaggio con le solite modalità viste nel paragrafo 4.3.2. Il messaggio ricevuto viene passato alla funzione per il parsing e la convalida; se non vi sono errori il codice dell'operazione da svolgere nell'ambito delle stime statistiche o delle soglie viene salvato nella variabile `est_operation`. A seconda di questo valore, il CIM esegue l'operazione opportuna, che, con l'eccezione della richiesta della lista degli stimatori (alla quale provvede direttamente), consiste nel passare il messaggio al corretto modulo CI.
- Scansione del vettore globale di interi `exit_interface`, organizzato con `MAX_INTERFACES` slot, nel quale uno slot posto a uno indica che il corrispondente CI è uscito per errori sui socket (ad esempio se il controller è stato chiuso per un errore interno). In tal caso, il CIM pulisce le risorse relative allo slot e segnala sia localmente che alla UI l'uscita del CI con un apposito messaggio di errore. Si noti che le funzionalità del CIM non sono influenzate da un simile evento.
- Controllo della variabile globale `parser_synchro_warning` e del vettore globale di strutture `pars_err`, per capire se all'interno di qualche CI è in corso la procedura di sincronizzazione di un parser in seguito ad un errore sui dati binari (vedi paragrafo 4.4.2). L'eventuale presenza di questo tipo di procedura in corso in un CI viene segnalata alla UI con un apposito messaggio.

Se sul master socket si riceve il comando *close*, così come se vengono riscontrati errori sui socket master o estimate (segnalati nel processo mediante assegnazione del valore 0 alle variabili `sock_err` ed `est_sock_err`), il CIM esce dal ciclo principale, libera i buffer allocati, cancella i thread CI eventualmente presenti, chiude le proprie due connessioni TCP ed esce.

4.4 Il modulo Client Interface (CI)

Il modulo Client Interface è la controparte (lato client) del server presente nel controller. È implementato da un thread principale che esegue la funzione `client_interface()`, presente nel file `c_int.c`, e da due thread figli corrispondenti ai parser per i dati binari real-time e snapshot, che eseguono le funzioni `rt_parse()` e `snap_parse()`, che si trovano rispettivamente nei file `r_t.c` e `snap.c`. Il thread principale è quello creato dal Client Interface Manager, ed ha un funzionamento di tipo ciclico. Quando la UI richiede la cancellazione di un CI, il thread principale del CI stesso non viene cancellato direttamente dal CIM mediante la funzione `pthread_cancel()` (soluzione meno pulita ed elegante), ma legge su un'apposita struttura globale (il vettore `kill_interface`) che il CIM gli sta richiedendo di uscire, quindi esce autonomamente.

Le funzioni che svolge un modulo Client Interface sono le seguenti:

- Interfaccia di client TCP con tre connessioni distinte (*control*, *real-time*, *snapshot* socket) per la comunicazione con un sistema di misura Metercontroller residente su un MA-LER DS.
- Comunicazione bidirezionale con il CIM mediante le strutture globali dinamiche di cui si è già parlato durante la descrizione del CIM stesso.
- Smistamento dei messaggi di controllo da e verso il control socket.

Quelli provenienti dal CIM (il quale a sua volta li ha ricevuti sul master socket dalla UI) vengono inoltrati al controller; quelli provenienti dal controller vengono passati al CIM che li inoltrerà alla UI.

- Mantenimento di una copia parziale del Traffic DB del Metercontroller, in base ai dati real-time e snapshot che da esso arrivano.
- Attivazione di due thread adibiti a ricevere, interpretare e collocare nel DB locale i dati in arrivo sui socket real-time e snapshot.
- Parsing XML delle richieste di operazione nell'ambito delle stime statistiche e della gestione soglie, e supporto per tali operazioni.

La struttura di un modulo Client Interface è mostrata in figura 4.4.

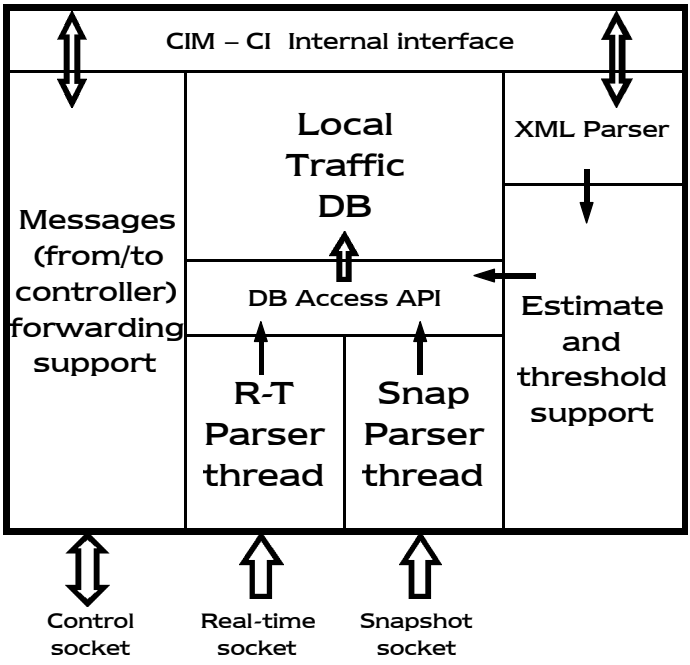

Figura 4.4 – Struttura del modulo Client Interface

4.4.1 L'interfaccia client verso il controller

Quando un modulo CI viene creato dal CIM, per prima cosa riconosce qual è lo slot che gli compete, in base al proprio *thread identifier* (tid), e ricava dal vettore globale di strutture `cii` le informazioni riguardo all'indirizzo IP del controller al quale deve connettersi. Fatto questo, inizializza un socket TCP e lo connette (*active open*) all'indirizzo ricavato in precedenza, alla porta `DEFAULT_PORT`. A questo punto c'è una procedura di scambio di hello con il controller (di cui si è già parlato nel paragrafo 4.2), che può in futuro essere integrata in un meccanismo di security. Se il tutto va a buon fine, il CI compie un passive open su una porta random, scelta sommando al valore `DEFAULT_PORT` le quattro cifre meno significative del tid, quindi comunica questo numero di porta al controller (sul control socket), affinché questo possa connettere i due socket di dati, real-time e snapshot. Il CI accetta le due connessioni, terminando in questo modo l'instaurazione dell'interfaccia di comunicazione. Nel caso di errori in qualsiasi parte della procedura, il CI chiude tutte le connessioni, segnala al CIM (tramite le strutture globali) ed alla UI (tramite un apposito messaggio) che sta uscendo per un errore di socket, pulisce le risorse dello slot utilizzato ed esce. Il funzionamento del CIM non ne sarà influenzato.

Terminata questa procedura, il CI inizializza le strutture informative da passare in ingresso ai parser e crea due thread per i parser stessi, uno in ascolto sul real-time socket, l'altro sullo snapshot socket. Successivamente, entra nel suo ciclo principale di funzionamento, che prevede le seguenti operazioni:

- Inizializzazione e pulizia delle strutture utilizzate per la ricezione che lo necessitino.

- Controllo delle strutture informative (vettore `cii`) per capire se c'è un messaggio da inoltrare al controller, e in caso affermativo inoltro del messaggio stesso sul control socket.
- Ricezione non bloccante sul control socket di eventuali messaggi del controller; anche qui si effettua il polling mediante l'uso della funzione `select()`, con un opportuno timeout che serve a temporizzare il ciclo principale. Le costanti per i secondi e microsecondi del timeout sono le solite viste per il CIM. Qualora il descrittore risulti pronto in lettura, il messaggio viene ricevuto e passato al CIM mediante le strutture globali. Il CIM lo invierà poi alla UI sul master socket. Anche sul control socket del CI i messaggi fanno uso del linguaggio XML, e il loro invio è preceduto da quello di un intero indicante la loro lunghezza in byte. Nel caso di perdita della connessione sul control socket, il CI segnale l'errore ed esce autonomamente, come è spiegato in seguito.
- Controllo delle strutture informative (vettore `cii`) per capire se c'è una richiesta operativa per la stima o per le soglie da prendere in esame. Nel caso che ci sia, il messaggio viene copiato su un buffer locale ed inviato alla funzione per il parsing XML, che lo analizza e ne ricava una struttura in cui memorizza i dati sull'operazione da svolgere (vedi paragrafo 4.4.4. Sia in caso di errore che di successo nello svolgimento della richiesta, il CI invierà una notifica direttamente sullo estimate socket.
- Controllo del vettore globale di strutture `pars_err`, per capire se uno dei due parser ha segnalato che si è verificato un errore sui socket di dati e si è persa la connessione. In tal caso, poiché il CI non può più ricoprire il proprio incarico, segnalerà errore al CIM ed alla UI ed uscirà.
- Controllo del vettore globale `kill_interface`, dove un valore 1 nello

slot appropriato segnala al CI che il CIM gli sta chiedendo di uscire. In questo caso si forza l'uscita dal ciclo pricipale del CI mediante la variabile `kill_state`, quini il CI esce.

Quando il CI deve uscire, provvede a cancellare i parser (a meno che l'uscita non sia stata causata da una loro segnalazione), chiude le tre connessioni TCP, compie le dovute pulizie per rendere nuovamente disponibile lo slot utilizzato ed esce. Nel caso l'uscita non sia stata richiesta dal CIM, il CI segnala al CIM che sta uscendo di propria iniziativa. Il CIM invierà un messaggio di errore alla UI in cui indica che un CI è uscito per errori sui socket, e specifica l'indirizzo IP del controller verso cui era connesso il CI che è uscito.

4.4.2 I parser real-time e snapshot

Il parser real time è un thread figlio del CI che sta sempre in ascolto sul real-time socket (la ricezione qui è bloccante) per interpretare i dati binari in arrivo e inserirli nel database locale di traffico. Il protocollo di trasferimento dei dati binari non utilizza il linguaggio XML, ma dei marker di dimensione fissa (fissata nella costante simbolica `MARKER` in `clnt.h`, attualmente a 6 caratteri) per delimitare le varie parti del messaggio.
Man mano che i dati binari vengono ricevuti, il parser real-time aggiorna il database locale di traffico, residente in memoria, con i nuovi valori dei timestamps e delle misure di traffico suddivise per LSP e PHB. Ad ogni completamento di messaggio, il parser real-time crea o aggiorna, a partire dal database, una serie di files su disco fisso contenenti le misure di traffico in un formato pronto per il plotting, o per l'accesso da parte di un'eventuale

UI grafica che voglia visualizzare la situazione del carico della rete.
Il parser snapshot è un altro thread figlio del CI, ed ha un funzionamento del tutto analogo al parser real-time, salvo che sta in ascolto sullo snapshot socket e non aggiorna i files sul disco rigido. Il totale dei thread per ogni modulo CI attivo è quindi tre.

Se l'altro capo delle connessioni dati cade per qualche motivo, i parser se ne accorgono, segnalano l'errore al CI ed escono. Il CI, non potendo più esplicare le proprie funzionalità, segnala errore di socket sia al CIM che alla UI, quindi esce. Nel caso in cui si verifichino invece errori di ricezione (come ad esempio un marker errato o dati binari quando è atteso un marker) i parser danno il via ad una procedura con la quale tentano di recuperare il sincronismo di ricezione. Viene ricevuto un carattere alla volta, finché sei caratteri successivi non formano un marker di fine messaggio, cosicché si possa ricominciare con la corretta ricezione immediatamente dopo. La procedura si protrae per un numero totale di bytes configurabile in fase di compilazione mediante la costante simbolica `SYNCHRO_CHARACTERS`, definita in `clnt.h`. Se non viene trovato un marker di fine messaggio entro lo scadere di questo numero di bytes, il parser dichiara sincronismo perso ed esce segnalando al CI errore sul socket, causando come detto prima anche l'uscita del CI stesso.

4.4.3 Il database locale delle misure di traffico

Ogni modulo Client Interface mantiene un proprio database locale per le misure di traffico ed i relativi timestamp, che in sostanza è la copia di una parte del Traffic DB mantenuto dal controller al quale il CI è connesso. La

quantità di dati trasferiti dipende dalle richieste dell'utente; nel caso di una full connection, ad esempio, nel database locale del CI si trovano in realtà le misure di tutti gli LSP e PHB.
La struttura del database è già stata descritta nel capitolo 2 (nel paragrafo 2.3.2). La profondità temporale del database dei CI è la stessa di quella del database del Metercontroller; entrambe sono fissate in fase di compilazione mediante la stessa costante simbolica `HISTORY_LENGTH`.

L'accesso al database per l'inserimento dei dati di traffico è effettuato dai due parser real-time e snapshot, mediante una libreria completa di funzioni messe a punto nell'ambito del progetto del Metercontroller, che mascherano la complessità delle strutture dati del DB. Anche per la creazione dei files su disco rigido si sono utilizzate delle funzioni preesistenti, sempre create assieme al Metercontroller. Tutte queste componenti già implementate e testate hanno semplificato notevolmente il lavoro per la messa a punto del modulo Client Interface e dei parser stessi.

4.4.4 Il supporto per le operazioni di stima

Il modulo Client Interface svolge la maggioranza delle attività nell'ambito delle operazioni per le predizioni statistiche e per le soglie. Con l'eccezione della richiesta *estimators list*, gestita dal CIM, il CI ha il compito di mettere in opera tutti gli altri task. A questo scopo è dotato di un parser XML (solo per i messaggi di stima e soglie) che prende in esame le richieste passate dal CIM e recupera le informazioni sull'operazione da svolgere creando una struttura di cui si riporta la definizione (definita in `clnt.h`):

```
struct threshold_estim_struct
        {
        double upper_th;
        double lower_th;
        };

struct add_estim_struct
        {
        char estim_name[MAX_EST_NAME];
        char estim_proto[MAX_EST_PROTO];
        struct threshold_estim_struct estim_th;
        };

struct adjust_estim_struct
        {
        char estim_id[MAX_EST_NAME];
        char estim_proto[MAX_EST_PROTO];
        struct threshold_estim_struct estim_th;
        };

struct remove_estim_struct
        {
        char estim_id[MAX_EST_NAME];
        };

struct estimate_task_struct
        {
        int estim_requestid;
        int estim_opcode;
        unsigned long estim_lspid;
        unsigned long estim_phbmask;
        int estim_errorcode;
        union {//Struttura diversa a seconda dell'opcode
                struct add_estim_struct estim_add;
                struct adjust_estim_struct estim_adj;
                struct remove_estim_struct estim_rmv;
                struct threshold_estim_struct lsp_th;
                } estim_opstruct;
        };
```

Se ci sono errori logici o formali nel messaggio, il CI notifica la cosa alla UI direttamente sullo estimate socket. Se tutto va bene, creata la struttura il CI esegue il dovuto task e notifica il successo. Le funzioni per l'esecuzione delle procedure di stima non rientrano nel lavoro previsto per questo progetto, e sono al momento presente in fase di sviluppo. L'agganciamento di uno stimatore è stato già realizzato, e nel capitolo riguardante le prove sperimentali (capitolo 6) si mostrano i grafici relativi.
Il CI fornisce comunque un supporto flessibile che può essere facilmente esteso per comprendere altre funzionalità che si dovessero richiedere in seguito.

4.5 L'interfaccia utente

L'utilità principale dei lavori contigui rappresentati dal sistema di misura Metercontroller e dell'architettura client-server sviluppata in questo progetto consiste nel monitorare l'effettiva condizione di carico del dominio MAID, misurando in tempo reale il traffico offerto all'intera frontiera. Tuttavia, questo sistema è incompleto senza un'interfaccia per la presentazione dei dati all'utente. Quest'interfaccia, che rappresenta il punto di partenza per le future espansioni del sistema, dovrebbe avere un certo numero di utili caratteristiche, prima tra tutte l'approccio grafico alla presentazione dei dati ed alla creazione delle richieste XML per il Client Interface Manager e per i Metercontroller. La UI o User Interface si configura come un software CGI (Common Gateway Interface). Tramite il CGI grafico dovrebbe essere possibile passare con semplici operazioni dai dati di un CI a quelli di un altro CI, dai grafici per un LSP a quelli per un altro. Anche per la composizione di messaggi di richiesta dovrebbero esserci strumenti per un

accesso rapido, intuitivo e visivamente gradevole, con l'utilizzo di *checkbox* e pulsanti. I file scritti dal real-time parser rappresentano una base per l'elaborazione off-line o per il plotting in tempo reale dei dati di traffico, sia all'interno del CGI che a parte.

Qualsiasi sarà il design dell'interfaccia utente, non si potrà prescindere nella implementazione della UI da una serie di regole costitutive. Essa, infatti, dovrà tenere conto della struttura del Client Interface Manager, innanzitutto comprendendo una componente di client con due connessioni TCP attive. Le connessioni saranno effettuate entrambe verso la porta `MASTER_PORT`; la prima sarà il master socket, la seconda lo estimate socket. L'invio e la ricezione dei messaggi dovranno rispettare rigidamente le regole definite nei protocolli d'accesso descritti sia in questo che nel successivo capitolo. In particolare si dovrà far precedere l'invio di ogni messaggio XML da quelo di un intero indicante la lunghezza del messaggio stesso. Analogamente, in ricezione si dovrà attendere prima un numero intero (`sizeof(int)` bytes), quindi un messaggio XML di lunghezza pari al valore dell'intero. Si dovrà altresì tenere di conto delle procedure presenti nel CIM in caso di errori, come ad esempio i tentativi di sincronizzazione della ricezione sui socket descritti in precedenza. Questi eventi sono segnalati alla UI mediante apposita messaggistica, e la UI dovrà reagire opportunamente o quantomeno renderne conto all'utente. Come si vedrà nel prossimo capitolo, i protocolli per le repliche da parte dei controller e del CIM forniscono informazioni esaustive e dettagliate riguardo a quanto accade nell'elaborazione delle richieste. Si potrà scegliere di mostrare nel CGI tutte queste informazioni o solo una parte, in base alle effettive necessità dell'utente, trovando un compromesso tra ricchezza informativa e snellezza dei dati presentati.

I messaggi XML che saranno creati ed inviati dovranno ottemperare alle DTD previste (vedi capitolo seguente), senza però includere il tag `DOCTYPE`.

Questo sarà aggiunto dai parser del CIM e del controller in fase di ricezione e convalida, ed una sua ripetuta presenza darebbe luogo ad errore.
Si ricorda infine che i protocolli scritti nel linguaggio XML sono facilmente ampliabili, ed i protocolli presenti possono essere facilmente rivisitati ed estesi. Qualora fossero necessarie modifiche alle DTD, esse si trovano nelle funzioni di seguito elencate:

- `xml_request_parseandvalid()`
- `estimate_request_parseandvalid()`

in `clnt_f.c`, per il Client Interface Manager, e:

- `xml_validate()`

in `ctrl_f.c`, per il controller.

Nel prossimo capitolo si descriveranno dettagliatamente i protocolli di comunicazione utilizzati, spiegando in quale modo sono trasportate le informazioni nei messaggi. Questo deve rappresentare senz'altro una guida per la progettazione dell'interfaccia utente, che dovrà costruire ed inviare messaggi conformi ai protocolli.