Acquisizione dati da stazioni sismiche digitali tramite Earthworm in ambiente GNU/Linux

Rosario Peluso, Walter De Cesare
Istituto Nazionale di Geofisica e Vulcanologia
Osservatorio Vesuviano

Rapporto tecnico n. 8/2006

Napoli, 14 luglio 2006

Sommario

In questo rapporto viene descritto un sistema di acquisizione dati sviluppato all'interno dell'attività del Centro di Monitoraggio dell'Osservatorio Vesuviano - INGV. Tale sistema consente di acquisire dati trasmessi da una stazione sismica digitale come flusso unidirezionale su una comune porta seriale o su una data porta di una connessione TCP/IP. Come mezzo di trasporto dei dati acquisiti viene utilizzato il sistema *Earthworm*. Il sistema *Earthworm* è stato scelto per la sua estrema diffusione e la sua modularità che consente di integrare facilmente diverse soluzioni per l'analisi, la visualizzazione e la conservazione dei dati precedentemente sviluppate per sistemi di acquisizione di altro tipo. Nel rapporto viene inoltre brevemente descritto il processo di *porting* di *Earthworm* sotto un sistema operativo GNU/Linux. Viene quindi illustrata la funzionalità e la configurazione del "modulo" di *Earthworm* `lantronix2ring`, vero cuore del sistema di acquisizione.

Indice

1 Il sistema *Earthworm* e GNU/Linux.

Il progetto *Earthworm* [1] è stato sviluppato da un gruppo di ricercatori del *United States Geological Survey* (USGS) per permettere l'acquisizione, il trasporto, l'analisi e la conservazione di dati sismici. *Earthworm* fornisce uno strato di comunicazione locale ("Ring") e remoto (sia in TCP/IP che in UDP/IP) che permette a più programmi ottimizzati per un unico compito (moduli) di comunicare tra loro in modo semplice ed efficace. Grazie a questo tipo di struttura, è estremamente semplice aggiungere nuovi moduli al sistema per compiti specifici.

Il sistema *Earthworm* è però distribuito, nella sua forma ufficiale, esclusivamente per i sistemi operativi WINDOWS™ e SOLARIS™. Solo negli ultimi anni si sono avuti i primi tentativi di effettuare il *porting* anche per altre piattaforme, tra cui GNU/LINUX. Sia quest'ultimo che SOLARIS™ sono sistemi operativi che seguono lo standard POSIX [2]. Come base per il *porting* è stata dunque scelta la versione per SOLARIS™. Ciò ha richiesto solo un numero limitato di modifiche nelle parti essenziali del sistema (gestione della memoria condivisa, dei *thread*, della rete ed altro), più altre modifiche di minore importanza.

La distribuzione di *Earthworm*, tuttavia, ha un sistema di installazione e gestione di programmi e librerie estremamente essenziale. Questo sistema di compilazione ed installazione non era adatto ad una distribuzione degli eseguibili su vasta scala, vista anche la natura intrinsecamente distribuita di *Earthworm* stesso. La maggior parte dello sforzo è stata dunque indirizzata verso la necessità di utilizzare un sistema di compilazione e distribuzione che fosse più elastico di quello proposto. Si è scelto, a tal scopo, di utilizzare il sistema degli *Autotools* della fondazione GNU [3–5]. Utilizzando delle

direttive opportune, essi sono in grado di creare i ‘‘`Makefile`’’ e persino piccoli pezzi di codice in grado di rendere la compilazione indipendente dal sistema operativo e dal processore utilizzato.

Vista, però, l'estrema complessità del progetto, si è scelto di non generare un albero di compilazione estremamente generico, ma, come primo tentativo, solo per sistemi POSIX e per GNU/Linux in particolare. All'indirizzo, temporaneo ed ancora in fase di test, `http://pcvh.ov.ingv.it:8000/earthworm/` è possibile trovare il pacchetto dei sorgenti (*tarball*) che, attualmente, compila su svariate distribuzioni di GNU/Linux. In particolare è stato compilato ed utilizzato con successo su *Debian Sarge x86 e Sparc, RedHat 9.0, Fedora Core 3, 4 e 5, Mandrake 10, Mandrivia*. Utilizzando questa *tarball* come base è stato anche possibile generare i pacchetti binari per alcune delle distribuzioni indicate. Per *Debian Sarge* è anche disponibile l'archivio *apt enabled* per automatizzare l'installazione e l'aggiornamento del pacchetto stesso. Per accedere all'archivio, occorre aggiungere al *file* `/etc/apt/sources.list` la riga (indirizzo ancora temporaneo ed in test):
`deb http://pcvh.ov.ingv.it:8000/debian stable contrib`

2 Il modulo di acquisizione `lantronix2ring`.

Il modulo `lantronix2ring` è in grado di leggere un flusso di dati provenienti da una o più connessioni TCP/IP, convertirli in pacchetti *tracebuffer* di *Earthworm* e scriverli nello strato di trasmissione locale (*Ring*), da cui, poi, potranno essere usati in locale o trasmessi in remoto da altri moduli di *Earthworm* stesso.

Il primo embrione del modulo è stato sviluppato per acquisire i dati della stazione digitale *Gaia* (sviluppata presso la sezione INGV di Roma): essa utilizza un semplice protocollo in cui non sono previste richieste da parte di

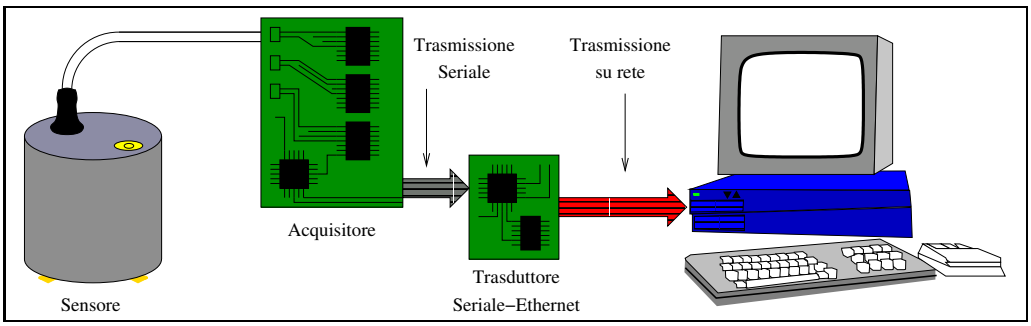

Figura 1: Acquisizione di dati seriali tramite trasduttore seriale/ethernet.

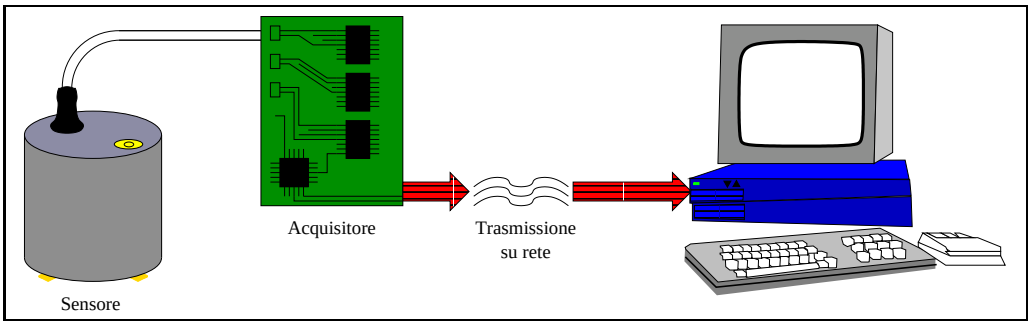

Figura 2: Acquisizione diretta TCP/IP, via cavo o wireless.

chi legge, ma i dati sono trasmessi in modo "cieco" da parte della stazione stessa. Questo flusso viene trasmesso su una porta seriale e può essere convertito poi in TCP/IP tramite un trasduttore seriale/ethernet come mostrato in figura (1).

Il nome `lantronix2ring` deriva proprio dal primo modello di trasduttore utilizzato all'Ov-ingv prodotto, per l'appunto, dalla Lantronix™. In generale, però, il programma può essere utilizzato per qualunque tipo di trasmissione su porta TCP/IP in cui non ci sia necessità di effettuare richieste, come illustrato in figura (2). È già in via di sviluppo un modulo più avanzato, in grado, oltre che di leggere i dati in modo "cieco", anche di instaurare un certo qual dialogo sulla connessione TCP/IP.

In figura 3 è schematicamente illustrato il funzionamento del modulo, che è composto da 5 *thread* così divisi:

1. **Main thread**: è quello che viene lanciato all'inizio del programma, si occupa di creare gli altri quattro *thread* e di controllarli durante tutta l'esecuzione.

2. **Connecter thread**: su richiesta invia la connessione ad ogni sorgente di dati. Se questa non riesce ci riprova dopo un tempo configurabile. Quando invece riesce, i dati della connessione vengono passati al *Reader*.

3. **Reader thread**: effettua una `select` su tutte le connessioni in suo possesso. Quando riceve dati su una di esse, questi vengono passati al *Parser* assieme alle informazioni sulla sorgente da cui provengono. Se una sorgente non manda dati per un certo tempo (configurabile), la connessione viene chiusa e viene mandata una richiesta di riconnessione al *Connecter*.

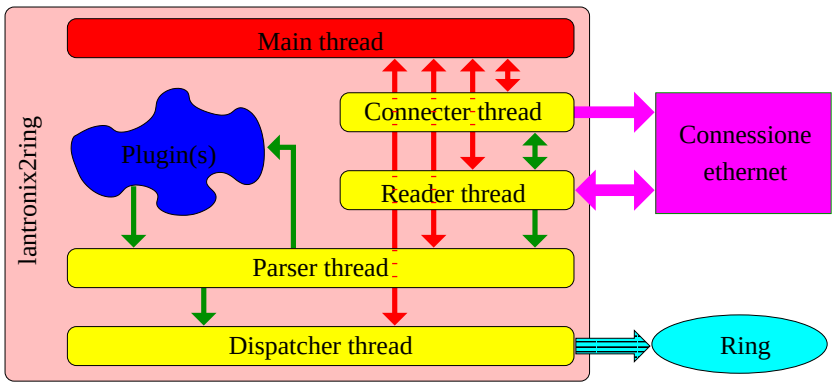

Figura 3: Schema di funzionamento di `lantronix2ring`.

4. **Parser thread**: "filtra" i dati ricevuti tramite uno o più *"plugin"* (vedi sezione 2.1) che effettuano la decodifica dei dati, ne verificano la consistenza e la correttezza e, infine, li trasformarno in pacchetti *Earthworm*. I pacchetti così ottenuti vengono poi passati al *Dispatcher*.

5. **Dispatcher thread**: può effettuare due operazioni diverse a seconda di come è configurato il programma. La prima delle due consiste nello scrivere i dati in formato ASCII su di un *file*: essa ha finalità essenzialmente di debug ed è destinata a scomparire in future versioni del programma. La seconda consiste nello scrivere i dati come pacchetti *Earthworm* nel *Ring* per la loro successiva manipolazione. In questa modalità di funzionamento questo *thread* riceve anche i messaggi di shutdown dal *Ring* e, eventualmente, li passa al *Main thread*.

2.1 I *plugin* di `lantronix2ring`.

Come emerge da quanto detto in precedenza, il programma non si preoccupa minimamente di sapere quale sia il formato dei dati letti: essi vengono passati come array di byte al *plugin* il quale ha il compito di convertirli in un formato utile.

Un *plugin* altro non è che uno *"shared object"* (libreria condivisa) che viene caricata su richiesta dal programma durante la fase di partenza. Ogni *plugin* è progettato per decodificare un particolare tipo di dato ed è quindi legato al tipo di stazione digitale scelta. Esso viene caricato un'unica volta e viene utilizzato dal *Parser* per tutte le connessioni che lo richiedono.

Questo rende l'estensione delle funzionalità del modulo estremamente semplice. Aggiungere la decodifica dei dati di un nuovo tipo di stazione

si riduce allo scrivere l'apposita libreria dinamica in grado di interpretarne il formato. Nella sezione 3 si riporta l'API (Application Programming Interface) che la libreria deve possedere per essere caricata ed utilizzata con successo.

Sono attualmente disponibili tre diversi *plugin* per il modulo `lantronix-2ring`, due dei quali attualmente utilizzati per l'acquisizione della rete di monitoraggio dell'Ov-ingv. Essi sono i *plugin* utilizzati per la decodifica dei dati delle stazioni *Gaia* e *Criceta*, quest'ultima sviluppata presso i laboratori dell'Ov-ingv. Il terzo di essi è disponibile in via ancora sperimentale per la stazione *Gilda* anch'essa in via di sviluppo presso i laboratori dell'Ov-ingv.

3 L'API dei *plugin.*

In questa sezione viene descritta l'API che permette al *plugin* di interagire con il modulo `lantronix2ring`. L'API comprende due insiemi di simboli. Il primo, obbligatorio, deve essere esportato dalla libreria per renderla utilizzabile dal programma (cfr. sezione 3.1). Il secondo, invece, è un insieme di simboli esportati dal programma che possono essere opzionalmente utilizzati da chi sviluppa le librerie per facilitare l'utilizzo di alcune funzioni "standard" (cfr. sezione 3.2).

Quelli che vengono riportati nelle due sezioni 3.1 e 3.2 sono i simboli come definiti dall'attuale versione del pacchetto `earthworm-ov` (cfr. sezione 5). L'API è da considerarsi stabile per incrementi della *micro* e della *minor version* del pacchetto, mentre potrebbe subire modifiche per incrementi della *major version*.

3.1 Simboli definiti dalla libreria.

La libreria deve definire un insieme di chiamate in ANSI C che vengono risolte al momento del caricamento della stessa. Si riporta di seguito l'insieme di simboli che devono essere definiti.

`void *create_data_context( const char *name ):`

Viene chiamata una volta per ogni cliente del programma a cui viene agganciato il *plugin* selezionato. Deve restituire un puntatore ad un oggetto "opaco" che non viene utilizzato da `lantronix2ring` ma solo dalla libreria la quale è l'unica tenuta a conoscerne il contenuto ed il significato. Generalmente esso contiene tutte le informazioni riguardanti il *parsing* del flusso di dati in arrivo (da qui in poi verrà chiamato *"contesto"*). Il parametro `name` contiene il nome del canale come definito nel *file* di configurazione.

`char want_configuration( void ):`
: Questa funzione deve restituire `1` se il *plugin* richiede dei parametri di configurazione, `0` altrimenti.

`const char *add_configuration_values( const char *key, const char **values, size_t nval, void *handle ):`
: Chiamata una volta per ogni parametro del *plugin* contenuto all'interno del *file* di configurazione del programma. `key` punta al nome del parametro, `values` punta ad un vettore di `nval` stringhe corrispondenti ai valori di quel parametro, `handle` è il contesto. La funzione deve restituire il puntatore ad una stringa di caratteri in caso di errori, `NULL` in caso contrario.

`const char *check_configuration( void *handle ):`
: Viene chiamata per controllare se nella configurazione del canale non ci siano problemi. Restituisce un puntatore alla stringa contenente l'errore, `NULL` altrimenti. `handle` è il contesto su cui controllare la configurazione. Dopo la chiamata a questa funzione, il modulo `lantronix2ring` non farà ulteriori chiamate alla funzione `add_configuration_values`.

`void restart_data_parsing( void *handle ):`
: Viene chiamata ogni qual volta, dopo un timeout, c'è stata una disconnessione ed una riconnessione del canale in questione. La funzione viene chiamata immediatamente dopo la chiusura della connessione, `handle` è il suo contesto.

`parser_state_t parse_data_array( const char *buffer, size_t size, void *handle ):`
: Viene chiamata ogni qual volta ci sono dei dati disponibili per essere filtrati. `buffer` punta al buffer di caratteri contenente i `size` byte da filtrare, `handle` è il contesto della connessione. La funzione deve restituire un enumerativo di tipo `parser_state_t` contenuto nell'header `definitions.h` che definisce i seguenti simboli:

  `context_error`: Quando sia stato passato un contesto non valido.

  `unknown_state`: Quando sia stato raggiunto uno stato non valido durante l'esecuzione del *parsing*.

  `first_buffer`: Questo stato può essere restituito quando si sia appena letto il primo buffer di dati del flusso. Non è però richiesto.

`parsing_data`: Il parser sta ancora analizzando i dati e non ci sono pacchetti pronti per l'estrazione.

`data_available`: Deve essere restituito quando ci sono pacchetti pronti per l'estrazione.

`parser_state_t extract_data( DataMessage_t **data, void *handle )`:

Quando `parse_data_array` restituisce `data_available` viene chiamata questa funzione per estrarre il successivo pacchetto *Earthworm* disponibile. Il parametro `data` punta ad un puntatore che deve essere inizializzato con l'indirizzo di un oggetto di tipo `DataMessage_t`, mentre `handle` è il contesto della connessione. La funzione deve restituire `data_available` se è appena stato estratto un dato, mentre `parsing_data` se non ci sono ulteriori dati da estrarre, in tal caso il puntatore `*data` può non essere definito in quanto non utilizzato da `lantronix2ring`, al limite può anche valere `NULL`. Il modulo continua a chiamare questa funzione finché essa non restituisce `parsing_data`. Il tipo `DataMessage_t` è definito in `definitions.h` nel seguente modo:

```
struct data_message_s {
  TRACE_HEADER   dm_header;
  int            dm_datasize, dm_status;
  size_t         dm_msgsize;
  char          *dm_data;
};

typedef  struct data_message_s   DataMessage_t;
```

i suoi membri sono così definiti:

`dm_header`: L'header del pacchetto *Earthworm* così come sarà poi scritto nel *Ring*. Il tipo `TRACE_HEADER` è definito negli header standard di *Earthworm*.

`dm_datasize`: La dimensione (in byte) del singolo campione all'interno del pacchetto. Non viene usato esternamente al *plugin*.

`dm_status`: Membro riservato per future implementazioni.

`dm_msgsize`: La dimensione (in byte) dei dati puntati da `dm_data`.

`dm_data`: Il vettore dei dati che verrà poi scritto nel *Ring* insieme all'header del pacchetto `dm_header`.

`void remove_data( DataMessage_t *data ):`
Quando `lantronix2ring` ha finito di utilizzare il pacchetto questa funzione viene chiamata per informare il *plugin* che può riprendere il controllo dell'area di memoria puntata da `data`.

`void remove_data_context( void *handle ):`
Questa funzione viene chiamata per distruggere il contesto puntato da `handle` e liberare le risorse da esso allocate. Viene chiamata un'unica volta per canale quando viene chiuso il programma.

3.2 Simboli messi a disposizione dal programma.

Viene di seguito riportato l'elenco dei simboli che vengono messi a disposizione dal programma `lantronix2ring` verso i *plugin*. Si tratta di simboli, funzioni e contesti che possono aiutare a scrivere nuovi *plugin* in maniera più agevole.

3.2.1 Il buffer di caratteri autoespandente.

Questo insieme di funzioni può essere utilizzato per gestire un buffer di caratteri di dimensione variabile in cui conservare e successivamente estrarre i byte necessari al *parsing* dei dati. Sono tutti definiti nell'header `buffer.h`.

Tutte queste funzioni girano intorno ad un oggetto di tipo `PluginBuffer_t` definito come:

```
struct plugin_buffer_s {
  size_t     cb_size;
  char      *cb_buffer, *cb_start, *cb_end;
};

typedef  struct plugin_buffer_s  PluginBuffer_t;
```

i cui membri sono:

`cb_size`: La dimensione totale in byte del buffer.

`cb_buffer`: Il puntatore all'inizio di tutto lo spazio disponibile per il buffer.

`cb_start`: Puntatore al primo carattere non ancora letto all'interno del buffer: si utilizza questo membro per accedere direttamente all'array di caratteri.

`cb_end`: Puntatore al primo carattere **successivo** all'ultimo carattere utilizzabile nel buffer: il buffer si può considerare vuoto quando questo puntatore coincide con `cb_start`.

`struct plugin_buffer_s *allocBuffer( void )`:
Questa funzione crea un nuovo oggetto `PluginBuffer_t` contenente inizialmente un buffer di caratteri di 1024 byte. `cb_start` e `cb_end` del nuovo oggetto vengono fatti puntare all'inizio del buffer stesso.

`void freeBuffer( struct plugin_buffer_s *buffer )`:
Libera la memoria puntata da `buffer->cb_buffer` e distrugge l'oggetto puntato da `buffer`.

`void pushDataInBuffer( struct plugin_buffer_s *buffer, const char *newdata, size_t newsize )`:
Aggiunge i `newsize` byte contenuti nell'array puntato da `newdata` all'interno del buffer di `buffer`. Il membro `cb_end` viene modificato facendolo puntare alla nuova fine dell'array di caratteri. Qualora la dimensione dell'array fosse insufficiente a contenere i nuovi dati, la memoria puntata da `cb_buffer` viene reallocata e i due membri `cb_start` e `cb_end` vengono modificati di conseguenza. Dopo una chiamata a questa funzione i precedenti valori di questi due membri non devono essere più considerati validi.

`void advanceBufferAt( struct plugin_buffer_s *buffer, const char *position )`:
Avanza il puntatore `buffer->cb_start` alla posizione puntata da `position` quando `position` è contenuto tra `buffer->cb_start` e `buffer->cb_end`. Se `position` cade dopo `buffer->cb_end` sia questo che `buffer->cb_start` vengono fatti puntare a `buffer->cb_buffer` (reset del buffer).

`void advanceBufferOf( struct plugin_buffer_s *buffer, size_t size )`:
Avanza il puntatore `buffer->cb_start` di `size` byte. Se la nuova posizione è successiva a `buffer->cb_end` il buffer viene resettato.

`void clearBuffer( struct plugin_buffer_s *buffer )`:
Resetta il buffer facendo puntare `buffer->cb_start` e `buffer->cb_end` a `buffer->cb_buffer`.

3.2.2 Il buffer circolare di pacchetti.

Il programma `lantronix2ring` possiede anche un insieme di utilità per poter gestire in modo semplice ed efficiente un buffer circolare, chiamato *ring* ma da non confondere con il *Ring* di *Earthworm*. Le definizioni delle funzioni e degli oggetti utilizzati si trovano nell'header `ring.h`. Il buffer circolare è fatto in modo che, quando viene completamente riempito, i pacchetti più vecchi vengono scartati ed un messaggio viene mandato tramite la funzione di *Earthworm* `logit`.

Queste funzioni utilizzano un oggetto di tipo `BufferRing_t` definito come:

```
#define  DATA_BUFFER_SIZE     1024

struct buffer_ring_s {
  DataMessage_t     *br_ring[DATA_BUFFER_SIZE];
  DataMessage_t    **br_start, **br_end;
};

typedef  struct buffer_ring_s   BufferRing_t;
```

i cui membri sono:

- `br_ring`: Il buffer di `1024` puntatori ad oggetti di tipo `DataMessage_t` (vedi sezione 3.1).
- `br_start`: Puntatore al primo pacchetto non ancora utilizzato.
- `br_end`: Puntatore al primo pacchetto **successivo** all'ultimo pacchetto non ancora utilizzato. Il buffer è da considerarsi vuoto quando questo membro coincide con `br_start`.

Tuttavia l'accesso diretto ai membri del *ring* non viene generalmente effettuato, ma è preferibile utilizzare le funzioni messe a disposizione, di seguito specificate:

`void createMessage( DataMessage_t *message ):`
: Crea l'array di dati dell'oggetto `message` (`message->dm_data`) utilizzando le informazioni contenute in `message->dm_header` e `message->dm_datasize` che quindi devono essere inizializzate correttamente.

`void freeMessage( DataMessage_t *message ):`
: Dealloca sia lo spazio dati del messaggio in `message->dm_data` che lo spazio puntato dallo stesso `message`.

`DataMessage_t **newMessages( int nchan ):`
Alloca un vettore di `nchan` oggetti `DataMessage_t`.

`void freeMessages( DataMessage_t **messages, int number ):`
Dealloca i `number` messaggi puntati dal vettore `messages` distruggendo anche lo spazio del vettore. `number` dovrebbe coincidere con il numero di oggetti allocati utilizzando la funzione `newMessages` precedentemente vista.

`void initRing( struct buffer_ring_s *ring ):`
Initializza il buffer circolare puntato da `ring` per successivi utilizzi. Un buffer circolare non può essere utilizzato se non viene preventivamente inizializzato.

`void clearRing( struct buffer_ring_s *ring ):`
Rimuove tutti i messaggi attualmente presenti nel buffer circolare.

`void pushMessage( struct buffer_ring_s *ring, DataMessage_t *newmsg ):`
Aggiunge il pacchetto puntato da `newmsg` all'interno del buffer circolare `ring`. Il pacchetto, dopo essere stato aggiunto nel buffer, **non deve essere deallocato** finché non viene estratto. Come detto, quando il buffer dovesse riempirsi, i pacchetti più vecchi vengono sovrascritti.

`DataMessage_t *popMessage( struct buffer_ring_s *ring ):`
Estrae dal buffer puntato da `ring` il successivo pacchetto (quello puntato da `ring->br_start`) e aggiorna i puntatori interni. Restituisce `NULL` se il buffer circolare è vuoto.

`int dataAvailable( struct buffer_ring_s *ring ):`
Restituisce `1` se ci sono pacchetti nel buffer circolare. `0` altrimenti.

3.2.3 Il calcolo della latenza.

Una stazione sismica digitale è in grado di associare una informazione temporale ad ogni pacchetto di dati da essa creato e trasmesso. Generalmente il tempo UTC del pacchetto viene misurato tramite un ricevitore GPS. Errori nella trasmissione dei dati della stazione o problemi nella ricezione dei dati GPS possono compromettere la corretta temporizzazione dei dati. Risulta, quindi, necessario poter essere in grado di controllare l'esattezza e la correttezza del dato temporale di un certo pacchetto.

La libreria interna di `lantronix2ring` mette a disposizione dei propri *plugin* un metodo per svolgere questo compito e per una eventuale correzione di quei pacchetti che possano risultare effetti da errori nella temporizzazione. Esso utilizza il tempo interno del calcolatore per tentare di estrapolare il tempo "migliore" del pacchetto stesso. L'orario del calcolatore possiede generalmente una precisione assai minore rispetto a quella che può essere ottenuta utilizzando un ricevitore GPS o mezzi simili. Per ovviare a questo problema quello che si fa è calcolare una media delle differenze di tempo tra quello dei pacchetti correttamente temporizzati e quello interno del calcolatore. Questa media viene poi utilizzata per estrapolare il tempo di generazione di quei pacchetti eventualmente affetti da errori.

L'ipotesi che si fa è che il tempo di arrivo dei dati di un pacchetto nel calcolatore sia più o meno costante e dipendente solamente dal mezzo di trasporto del pacchetto stesso. In questo modo la differenza di tempo misurata dipende solamente dallo spostamento dell'orologio interno del calcolatore rispetto all'"ora esatta" e dal tempo medio di percorrenza del pacchetto stesso. La media è continuamente ricalcolata in modo da tener conto, istante per istante, dell'eventuale spostamento dell'orologio.

Attualmente il solo *plugin* di produzione che utilizza il calcolo della latenza è quello della stazione *Criceta*: esso crea un unico "contesto" per tutte le connessioni per le quali viene usato il *plugin* stesso. In questo modo nel calcolo della media vengono utilizzati i tempi di tutti i pacchetti di tutte le stazioni attualmente decodificate. L'utilizzo della latenza deve essere abilitato all'interno del *file* di configurazione (cfr. sezione 4) sia nella sezione del programma che, separatamente, in quelle dei singoli canali. Di default esso è disabilitato.

Attraverso il *file* di configurazione del modulo `lantronix2ring` si possono definire inoltre sia il numero minimo di punti necessari a calcolare la media che il numero minimo di secondi necessari. La media verrà ricalcolata ogni qual volta si hanno a disposizione un opportuno numero di tempi e sono passati un opportuno numero di secondi. Di default il minimo numero di punti è 60 ed il minimo numero di secondi è 120. I prototipi di tutte le funzioni riportate di seguito sono definiti nell'header `latency.h`.

`boolean_t latencyEnabled( void ):`

> Questa funzione restituisce `TRUE` se il calcolo della latenza è stato abilitato all'interno del *file* di configurazione, `FALSE` in caso contrario. Il tipo `boolean_t` è definito nell'header `<libraries/utilities/global_definitions.h>` nel seguente modo:
>
> ```
> typedef  unsigned char   boolean_t;
> ```

```
#ifndef TRUE
#define  TRUE     1
#endif

#ifndef FALSE
#define  FALSE    0
#endif
```

`struct latency_context_s *allocLatencyContext( void ):`
: Alloca un oggetto di tipo `struct latency_context_s` "opaco" per i *plugin* ai quali non importa il suo contenuto. Questo oggetto ("contesto") contiene tutte le informazioni necessarie ad effettuare il calcolo della media delle differenze di tempo e quindi a calcolare il tempo "migliore". Restituisce `NULL` in caso di errore.

`boolean_t pushGpsTime( struct latency_context_s *context, time_t epoch ):`
: Utilizza il tempo contenuto in `epoch` per calcolare la media delle differeze nel contesto `context`. La funzione restituisce `TRUE` nel caso ci sia una media già disponibile, `FALSE` altrimenti.

`boolean_t clockLatencyAvailable( struct latency_context_s *context ):`
: Restituisce `TRUE` se c'è una media disponibile, `FALSE` altrimenti. `context` contiene il contesto di cui si vuole conoscere la disponibilità della media.

`time_t calculatedGpsTime( struct latency_context_s *context ):`
: Restituisce il tempo calcolato quando nel contesto `context` sia presente una media utilizzabile, `0` altrimenti.

`void freeLatencyContext( struct latency_context_s *context ):`
: Ripulisce il contesto `context` allocato con `allocLatencyContext` e ne dealloca la memoria utilizzata.

4 Il *file* di configurazione del modulo `lantronix2ring`.

La configurazione del modulo `lantronix2ring` deve essere scritta in un *file*, di estensione `.d`, che va passato come unico parametro al lancio del programma stesso. Di solito questa operazione viene effettuata dal modulo standard

di *Earthworm* chiamato `startstop` che si preoccupa di preparare l'ambiente in cui gireranno gli altri moduli, di lanciarli e di controllarli.

Di seguito vengono riportate le direttive necessarie alla corretta configurazione del modulo.

`MouleId` *MODULE_ID*
: *MODULE_ID* deve specificare il codice identificativo del modulo come riportato all'interno del *file* `earthworm.d` utilizzato. Richiesto.

`RingName` *RING_NAME*
: *RING_NAME* deve specificare il nome del *Ring* su cui andare a scrivere i pacchetti decodificati. Richiesto.

`HeartBeatInt` *secondi*
: Specifica ogni quanti secondi deve essere lanciato un *"heartbeat"* sul *Ring* per confermare l'esistenza in vita del modulo. Deve essere inferiore all'intervallo specificato nel *file* `lantronix2ring.desc`. Facoltativo, il default è 60 secondi.

`LogFile` *file*
: Non utilizzato, destinato a scomparire o ad essere utilizzato in modo differente. Richiesto.

`IpAddress` *indirizzo*
: L'indirizzo IP dal quale devono essere lanciate le richieste di connessione. Deve coincidere con uno degli indirizzi della macchina. Richiesto.

`Network` *NAME*
: *NAME* specifica il network di *Earthworm* con cui devono essere generati i pacchetti. Richiesto.

`InactivityTimeout` *secondi*
: Numero di secondi dopo cui la connessione con un client inattivo deve essere chiusa. Facoltativo, il default è 10 secondi.

`LatencyEnabled` *true|false|0|1*
: Serve a specificare se si vuole abilitare il calcolo della latenza nella libreria. Facoltativo, il default è `false`.

`LatencySampleNumber` *number*
: Minimo numero di punti con i quali calcolare la media delle latenze. È facoltativo e deve essere ≥ 30, il default è 60.

`LatencyRecalculateTime` *secondi*
: Minimo numero di secondi dopo cui calcolare la media delle latenze. È facoltativo e deve essere ≥ 10, il default è 120.

`Filter` *`nome libreria`*

Definisce il nome e la libreria di un *plugin* da caricare. *name* verrà utilizzato nel *file* di configurazione come nome del *plugin*. *libreria* specifica il nome del *file* o il *path* completo della libreria dinamica. Quando il nome del *file* viene specificato senza il *path* completo, esso viene cercato nel percorso di default `/usr/share/earthworm-ov/plugins/`. È necessario a specificare quali *plugin* caricare.

`LantronixServer` *`nome indirizzo porta filtro`*

Serve a specificare una sorgente di dati a cui connettersi. *name* sarà il nome del server all'interno del *file* di configurazione, *indirizzo* e *porta* specificheranno l'indirizzo TCP/IP e la porta della sorgente di dati, mentre con *filtro* si specifica il nome del filtro (*plugin*) da agganciare alla sorgente. *filtro* deve essere già stato definito con la direttiva `Filter`. È richiesto per ogni stazione cui ci si vuole connettere.

`ServerConfiguration` *`name`*

Marca l'inizio della sezione di configurazione del server *name* come specificato dalla direttiva `LantronixServer`. Da questo momento in poi tutti i parametri incontrati verranno mandati alla funzione `add_configuration_values` del *plugin* (cfr. sezione 3.1). Il contenuto della sezione è strettamente dipendente dai parametri di configurazione richiesti dal *plugin*. È richiesto per tutti quei *plugin* per i quali la funzione `want_configuration` restituisce `1`.

`EndConfiguration`

Serve a terminare la sezione di configurazione aperta dal precedente `ServerConfiguration`. Richiesto.

`DispatchMode` *`File|Ring`*

Utilizzato per specificare il tipo di output richiesto per il modulo (cfr. sezione 2). Specificando `File` i dati vengono scritti su *file* (s) in formato ASCII, mentre con `Ring` i dati vengono scritti sul *Ring* come pacchetti *Earthworm*. Questo parametro è destinato a scomparire quando verrà eliminata la possibilità di `lantronix2ring` di scrivere i dati su *file* attualmente presente per operazioni di debug. Richiesto.

4.1 I parametri di configurazione del *plugin* della stazione *Gaia*.

Vengono di seguito riportate le direttive accettate nella sezione di configurazione di ogni server che utilizzi il *plugin Gaia*.

`LogLevel` *livello*
: Specifica un livello di log del *plugin*. Deve essere un intero tra 0 e 6 inclusivi o una delle parole chiave (tra parentesi l'equivalente formato numerico): `nothing (0), fatal (1), severe (2), error (3), warning (4), info (5), debug (6)`. Facoltativo, il valore di default è `error (3)`.

`RecoverJumps` *true/false/0/1 secondi*
: Quando il primo parametro risulta vero, il *plugin* tenta di recuperare quei dati per i quali la differenza nel contatore del pacchetto corrente rispetto al contatore dell'ultimo pacchetto accettato risulta minore dei *secondi* specificati. Il recupero avviene se il pacchetto era "in sequenza" rispetto all'ultimo pacchetto accettato e viene effettuato sommando al tempo di questo la differenza tra i due contatori ed assegnando il nuovo tempo al pacchetto corrente. Facoltativo, il default è `false`.

`Components` *est nord verticale*
: Serve a specificare i nomi delle tre componenti che la stazione trasmette in modo da identificare a quale componente il pacchetto stesso si riferisca. Facoltativo, i default per i tre nomi sono: `EHE EHN EHZ`.

`MaxCounterError` *numero*
: Serve a specificare il massimo numero di pacchetti fuori sequenza da non accettare prima di considerare sbagliato il primo della serie e resettare il contatore interno di controllo. Facoltativo, il default è 10.

4.2 I parametri di configurazione del *plugin* della stazione *Criceta*.

Vengono di seguito riportate le direttive accettate nella sezione di configurazione di ogni server che utilizzi il *plugin Criceta*.

`Channels` $channel_1$ $channel_2$ ... $channel_n$
: La stazione *Criceta* non manda nel pacchetto i nomi dei canali che sta acquisendo: questa direttiva serve a specificarli. Se ne possono specificare un numero qualunque ma solo i primi tre sono necessari ed utilizzati: essi corrispondono, nell'ordine, ai primi tre canali del digitalizzatore. Richiesto.

`LogLevel` *livello*
: Ha lo stesso significato ed utilizzo della corrispondente direttiva del *plugin Gaia*.

`Firmware` *1|2*
: La stazione *Criceta* esiste con due diverse versioni di *firmware*. Sebbene la prima versione non sia più utilizzata, il *plugin* è ancora in grado di decodificare il vecchio formato. Potrebbe scomparire in future versioni oppure supportare ulteriori versioni del *firmware*. Facoltativo, il default è 1.

`OverrideName` *name*
: La stazione *Criceta* trasmette in ogni pacchetto il nome della stazione stessa. Questo nome viene utilizzato per generare il pacchetto *Earthworm*. Quando si voglia utilizzare un nome diverso per la stazione oppure essa trasmetta un nome sbagliato, occorre specificare questa direttiva. Il default è di utilizzare il nome trasmesso dalla stazione.

`ConsecutiveDecoded` *numero*
: Minimo numero di pacchetti correttamente decodificati richiesti per l'inizializzazione del contatore interno di controllo del *plugin*. Facoltativo, il default è 10.

`ConsecutiveErrors` *numero*
: Massimo numero di pacchetti decodificati in maniera non corretta prima di considerare sbagliato il primo della sequenza ed accettare l'attuale come corretto. Facoltativo, il default è 10.

`UseLatency` *true|false|0|1 secondi*
: Abilita l'utilizzo della latenza all'interno del *plugin* quando il primo parametro è `true`. Il secondo parametro specifica il massimo numero di secondi che deve intercorrere tra il tempo del pacchetto corrente ed il tempo calcolato utillizzando la latenza. Se questa differenza risulta maggiore, il pacchetto viene scartato. Il controllo viene effettuato alla fine di tutti i tentativi di recupero del pacchetto che vengono svolti dal *plugin*.

  Quando abilitata, la latenza viene utilizzata anche per tentare di recuperare i pacchetti di dati che possano essere stati generati o trasmessi dalla stazione con errori nel tempo GPS o nel contatore del pacchetto stesso.

5 Il pacchetto `earthworm-ov`.

Il modulo `lantronix2ring` viene distribuito in un pacchetto chiamato `earthworm-ov` disponibile sia in forma sorgente che in forma binaria. Esso contiene

sia il modulo `lantronix2ring` che i tre *plugin* precedentemente descritti per le stazioni *Gaia*, *Criceta* e *Gilda*. Attualmente l'ultima versione stabile del pacchetto è la `1.2.9`.

In questo pacchetto sono contenuti anche gli altri moduli *Earthworm* sviluppati ex-novo all'Ov-ingv per l'acquisizione e la gestione dei dati sismici. Esso non contiene le versioni dei moduli standard di *Earthworm* modificati per le esigenze interne dell'Ov-ingv: questi possono essere trovati in un ulteriore pacchetto chiamato `earthworm-addons`.

Si riportano i programmi attualmente contenuti nel pacchetto `earthworm-ov`:

`adterm, array_init, arrayd e array2ring:`
: Utilizzati per acquisire i dati e per la gestione di un *Array* sismico a 48 canali in trasmissione continua installato al Vesuvio. Verranno descritti in un ulteriore rapporto tecnico.

`ring2suds:`
: Utilizzato per creare *file* in formato `DMX` a partire dai dati contenuti sul *Ring*. Attualmente utilizzato in produzione, esso è destinato ad essere sostituito da un altro modulo che utilizzerà una più potente libreria di gestione di *file* sismici.

`ring2day:`
: Modulo attualmente incompleto e non funzionante, avrebbe dovuto avere lo stesso compito del modulo `ring2suds` ma per produrre i *file* `DAY` utilizzati all'Ov-ingv. La sua funzionalità verrà però implementata dallo stesso modulo destinato a sostituire il `ring2suds`.

I programmi e le librerie del pacchetto `earthworm-ov` sono scritti parzialmente in `C` e parzialmente in `C++`. Il pacchetto è in grado di compilare su molte distribuzioni GNU/Linux recenti: è stato utilizzato con successo su *Debian Sarge, RedHat 9.0, Fedora Core 3* e *4*. Esso è attualmente disponibile solo per macchine con processore *x86* compatibili o comunque con la medesima endianess: sono perciò escluse, attualmente, le macchine *Sparc* e similari.

Esso dipende, oltre che dalle librerie di *Earthworm*, anche da alcune librerie esterne:

ClassAd [6]:
: Utilizzata dai moduli dell'*Array Vesuviano* per la configurazione. Questa dipendenza verrà probabilmente eliminata in future versioni del pacchetto.

BOOST [7]:

> Ampiamente utilizzate, in particolare, quelle per l'accesso al *filesystem*, quelle per la gestione dei *threads* e quelle per l'utilizzo delle *Regular Expression*. Esse consentono di utilizzare questo tipo di costrutti in modo assolutamente indipendente dall'ambiente di sviluppo e dal sistema operativo: sono perciò state scelte pensando a futuri *porting* del pacchetto `earthworm-ov` verso altre piattaforme.

Viene inoltre utilizzata una ulteriore libreria sviluppata internamente all'OV-INGV per la gestione delle *socket* in `C++`. Chiamata `libsocket++` essa, come il pacchetto `earthworm-ov`, è disponibile all'indirizzo:
`http://pcvh.ov.ingv.it:8000/debian/dists/stable/contrib/source/`

Tutti i pacchetti fin qui descritti sono inoltre disponibili in forma binaria per la distribuzione GNU/LINUX *Debian Sarge* in un archivio *apt-enabled* accessibile (come descritto nella sezione 1) all'indirizzo:
`deb http://pcvh.ov.ingv.it:8000/debian stable contrib`

Riferimenti bibliografici

[1] `http://folkworm.ceri.memphis.edu/ew-doc/`

[2] `http://standards.ieee.org/catalog/olis/posix.html`

[3] `http://sourceware.org/autobook/autobook/autobook_toc.html`

[4] `http://www.gnu.org/software/autoconf/`

[5] `http://vipe.technion.ac.il/~shlomif/lecture/Autotools/`

[6] `http://www.cs.wisc.edu/condor/classad/`

[7] `http://www.boost.org/`