Implementazione della funzionalità di Autoranging per sistema UWB DecaWaveEVB1000

Corso di LM in Ingegneria Robotica ed Automazione
Sistemi di Guida e Navigazione

Studenti:
Nicola Piga
Giulio Romualdi

Supervisore:
Prof. Lorenzo Pollini

xx/10/2017

Università di Pisa

Introduzione

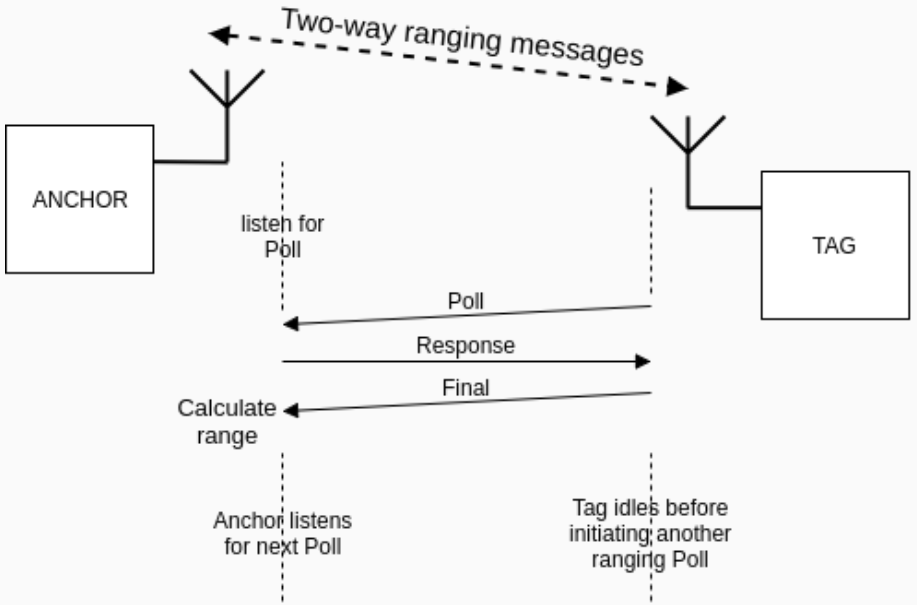
Contenuti

- Studio preliminare del Firmware della DecaWave
- Implementazione della procedura automatica di autoranging
- Implementazione di un Viewer per la visualizzazione di posizione ed assetto del Tag
- Sviluppi futuri

Descrizione della procedura di Ranging

Two-way Ranging

Durante l'interazione fondamentale vengono scambiati, tra tag ed ancora, un messaggio di Poll, uno di Risposta ed uno di Final.
I messaggi di Poll e Final vengono inviati dal tag a tutte le ancore.
Il messaggio di Risposta viene inviato dall'ancora al tag ed a tutte le altre ancore.



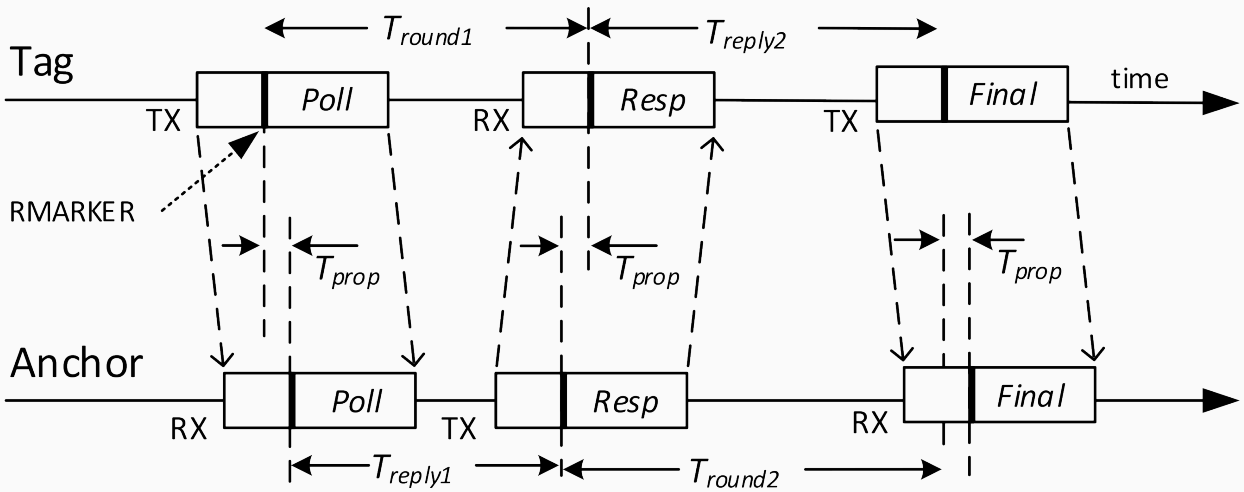
Two-way Ranging

il tag salva RMARKER di

- invio Poll (Rm_{ps})
- ricezione Risposta (Rm_{rr})
- invio Final (Rm_{fs}) (delayed transmission)

l'ancora salva RMARKER di

- ricezione Poll (Rm_{pr})
- spedizione Risposta (Rm_{rs})
- ricezione Final (Rm_{fr})



Il tag invia Rm_{ps}, Rm_{rr} e Rm_{fs} all'ancora nel Final

Calcolo del ToF

Le seguenti quantità sono calcolate dall'ancora utilizzando i dati raccolti e ricevuti dal tag durante un'istanza di ranging

$$T_{round1} = Rm_{rr} - Rm_{ps} \qquad T_{round2} = Rm_{fs} - Rm_{rr}$$
$$T_{reply1} = Rm_{rs} - Rm_{pr} \qquad T_{reply2} = Rm_{fr} - Rm_{rs}$$

Alla ricezione del Final l'ancora calcola il nuovo Tof (T_{prop}) che viene inviato al tag nella Risposta al Poll dell'istanza di ranging successiva

$$T_{prop} = \frac{T_{round1}\,T_{round2} - T_{reply1}\,T_{reply2}}{T_{round1} + T_{round2} + T_{reply1} + T_{reply2}}$$

Delayed transmission

Attenzione

La necessità di spedire nel messaggio di Final il tempo di spedizione del medesimo richiede l'utilizzo della funzione di *delayed transmission* del DW1000.

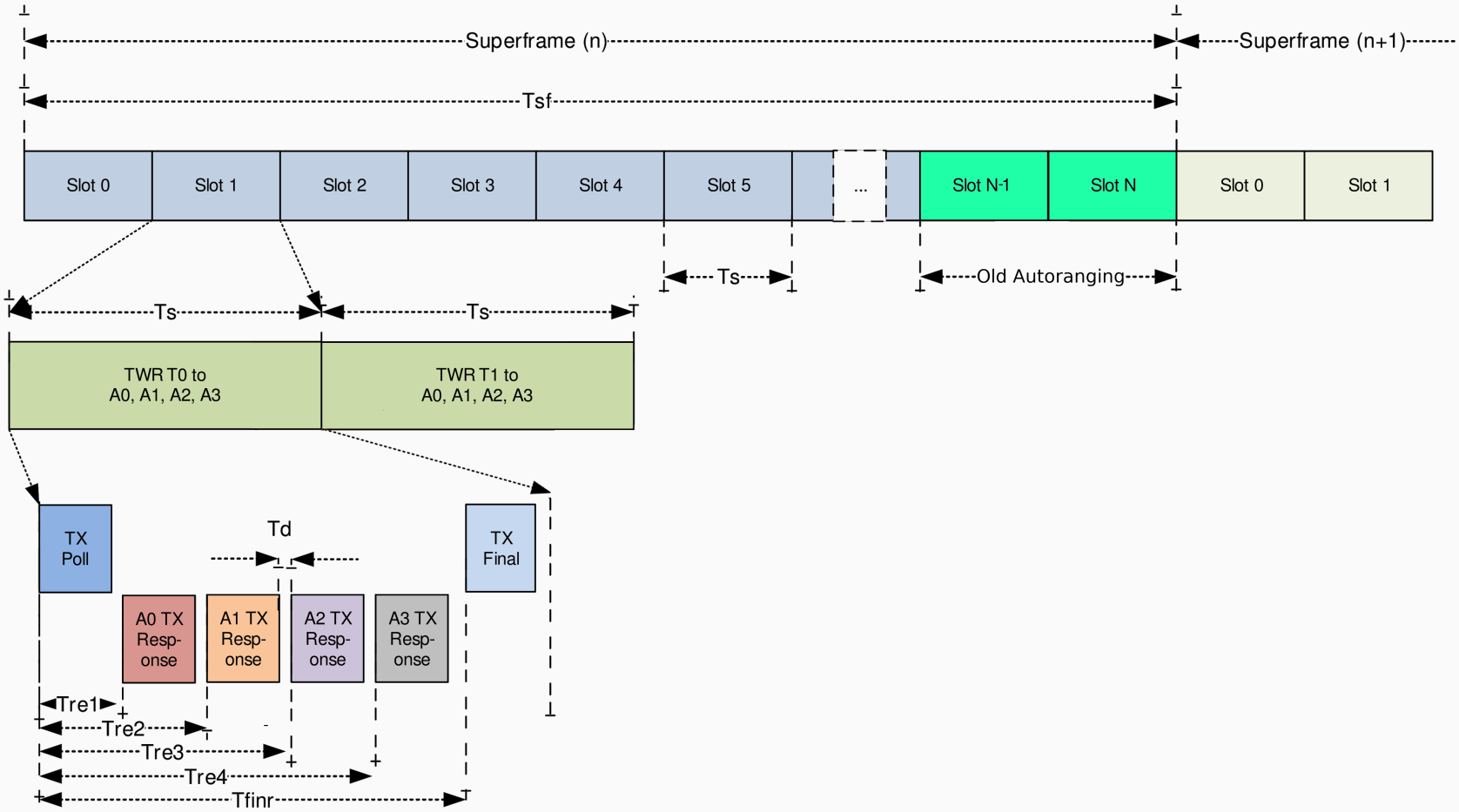
Il tempo di spedizione viene scelto a priori come

$$Rm_{fs} = Rm_{ps} + \Delta$$

dove Δ è maggiore del tempo impiegato da tutte le ancore per rispondere al tag (attenzione ad aumento di ancore). La funzione *delayed transmission* garantisce che l'RMARKER di spedizione del Final coincida con Rm_{fs}

Frame & Superframe

Il sistema supporta nativamente fino 8 tag e 3/4 ancore



Gestione di più tag

Ogni tag invia un messaggio di Poll ogni activation time a_t

$$a_t = T_{sf} + T_{sr} + T_{sc}$$

- T_{sf}: durata del Superframe
- T_{sr}: alla prima iterazione vale 10 ms successivamente vale 0 ms
- T_{sc}: correzione calcolata dall'Ancora 0 ed inviata al tag i-esimo

Calcolo della correzione T_{sc}

La correzione viene calcolata in base ad un errore calcolato come la differenza tra il tempo atteso d'arrivo del Poll e quello effettivo (dal punto di vista dell'ancora 0)

$$e = t_i^a - t_{rx}$$

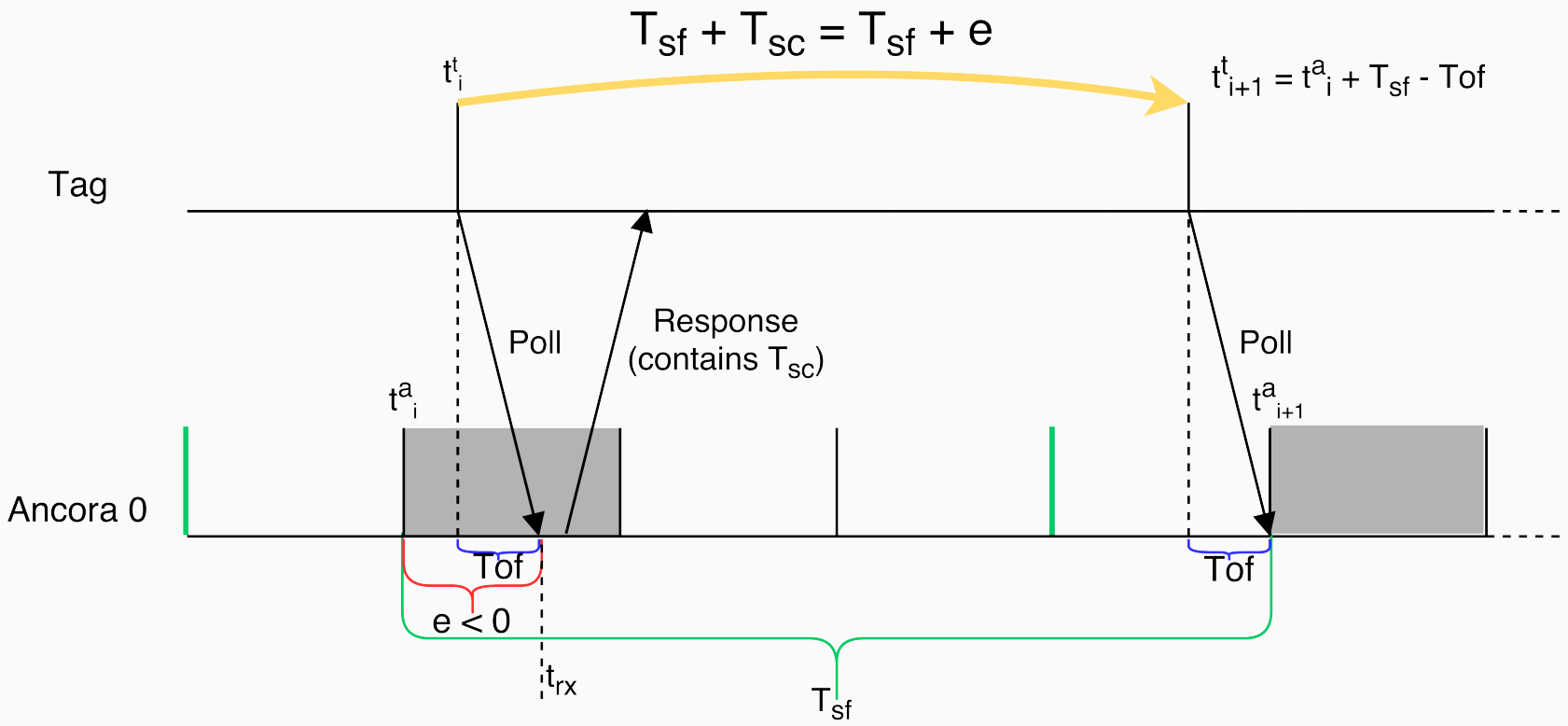
La correzione vale

$$\begin{cases} T_{sc} = e & \text{se } e < -\frac{T_{sf}}{2} \\ T_{sc} = T_{sf} + e & \text{altrimenti} \end{cases}$$

Calcolo Calcolo del tempo di attivazione $T_{sc} = e$

Il tempo di riattivazione del tag t^t_{i+1} è:

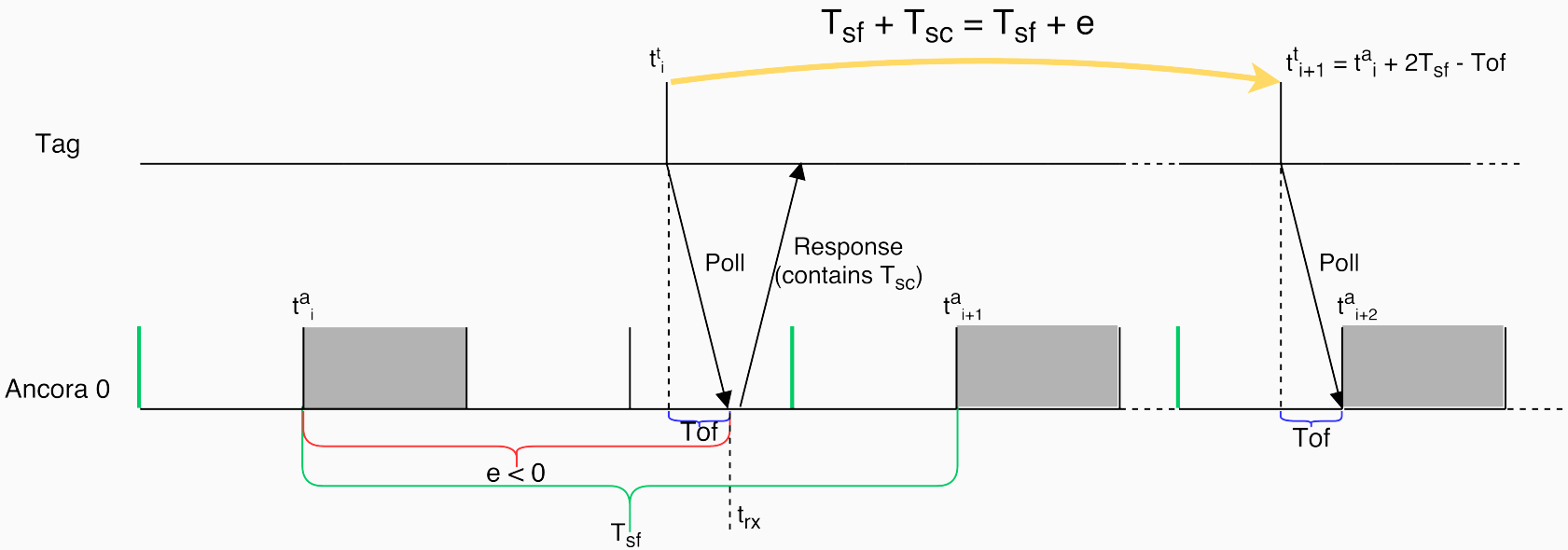
$$t^t_{i+1} = t^t_i + T_{sf} + T_{sc} = t^a_i - (\cancel{e} + Tof) + T_{sf} + \cancel{e} = t^a_i - Tof + T_{sf}$$



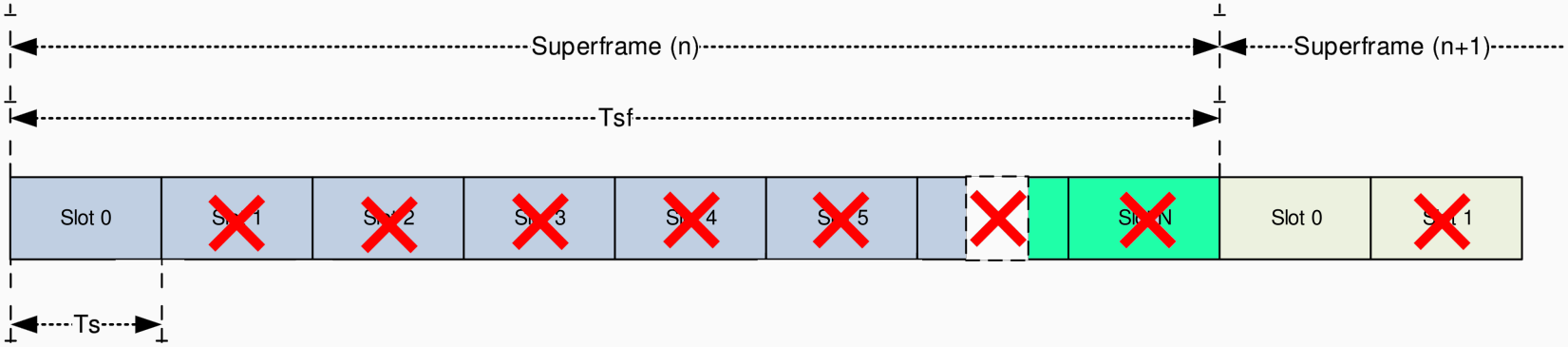
Calcolo del tempo di attivazione $T_{sc} = e + T_{sf}$

Il tempo di riattivazione del tag t^t_{i+1} è:

$$t^t_{i+1} = t^t_i + T_{sf} + T_{sc} = t^a_i - (\cancel{e} + Tof) + T_{sf} + (\cancel{e} + T_{sf}) = t^a_i - Tof + 2\,T_{sf}$$



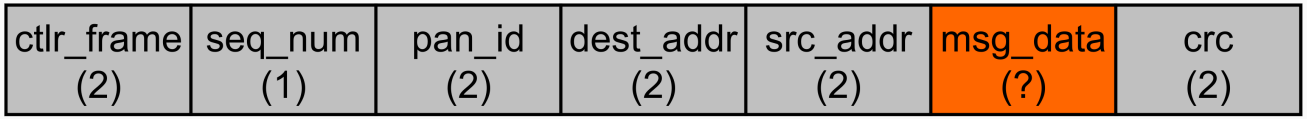
Aumento della frequenza di ranging



- contributo di Pinna, Malagoli, Giannini;
- utilizzabile con un solo tag;
- $T_{sf} = T_s = 10\,\text{ms}$;
- frequenza $f = 100\,\text{Hz}$.

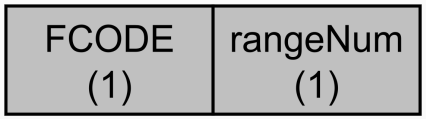
Struttura generale di un messaggio

I messaggi inviati sia dal tag che dalle ancore hanno la seguente struttura



- `ctrl_frame`: maschera nella quale, tra le varie cose, viene decisa la modalità di indirizzamento
- `seq_num`: sequence number
- `pan_id`: ID della rete
- `dest_addr`: indirizzo del destinatario
- `src_addr`: indirizzo del mittente
- `msg_data`: payload (dipende dal tipo di messaggio)
- `crc`: cyclic redundandcy check

Messaggio di Poll - `RTLS_DEMO_MSG_TAG_POLL`



- `FCODE = RTLS_DEMO_MSG_TAG_POLL`
- `rangeNum`: range number della trasmissione

Messaggio di Final - RTLS_DEMO_MSG_TAG_FINAL

FCODE (1)	rangeNum (1)	PTXT (5)	RRXT0 (5)	RRXT1 (5)	RRXT2 (5)	RRXT3 (5)	FTXT (5)	VRESP (1)

- FCODE = RTLS_DEMO_MSG_TAG_FINAL
- rangeNum: range number della trasmissione
- PTXT: tempo di invio del Poll
- RRXT0: tempo di ricezione della risposta da ancora 0
- RRXT1: tempo di ricezione della risposta da ancora 1
- RRXT2: tempo di ricezione della risposta da ancora 2
- RRXT3: tempo di ricezione della risposta da ancora 3
- FTXT: tempo di invio del Final
- VRESP: maschera della risposte valide ricevute

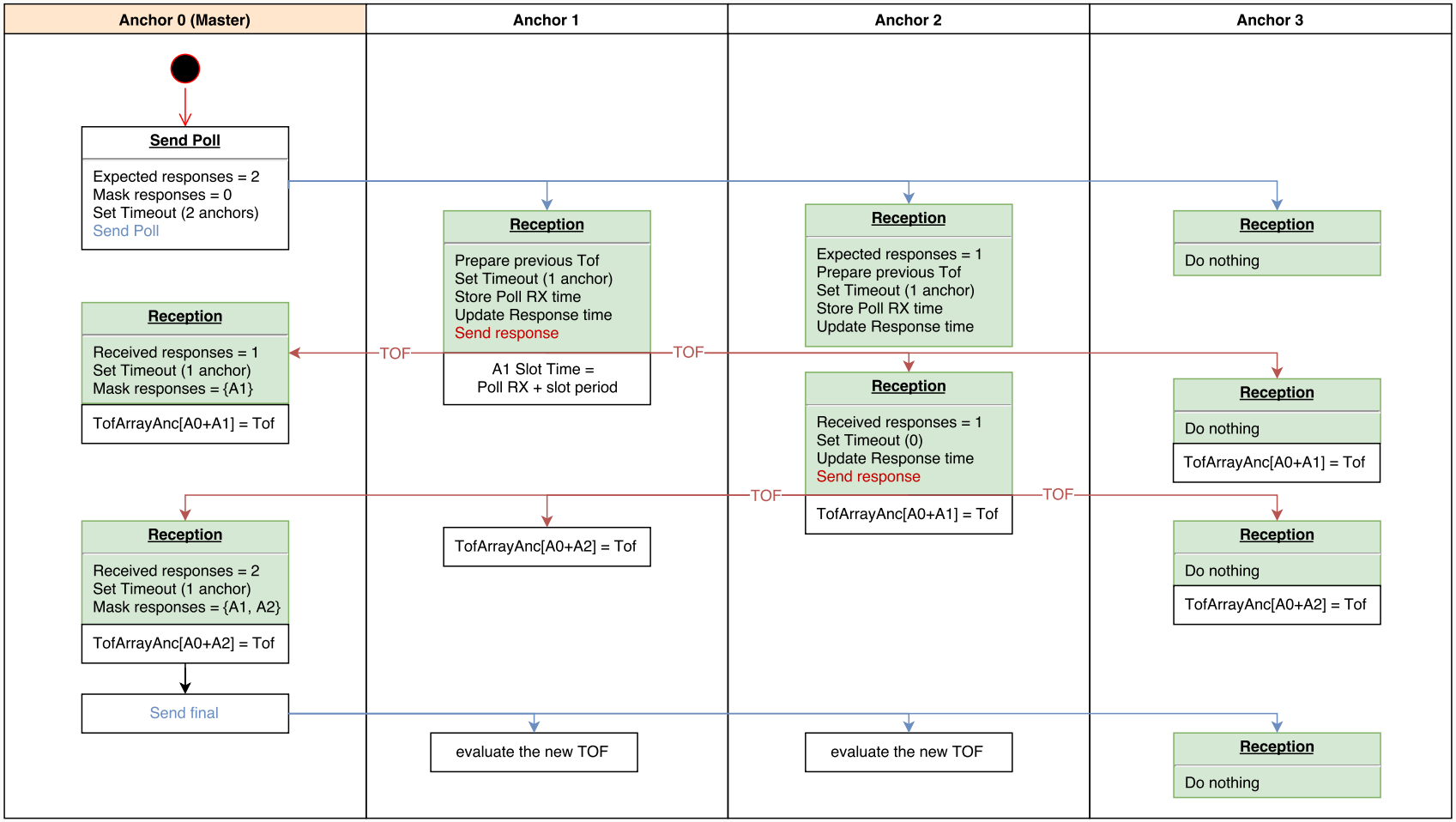
Messaggio di Risposta - RTLS_DEMO_MSG_ANCH_RESP

FCODE (1)	RES_TAG_SLP0 (1)	RES_TAG_SLP1 (1)	PREVIOUS_TOF (4)	rangeNum (1)

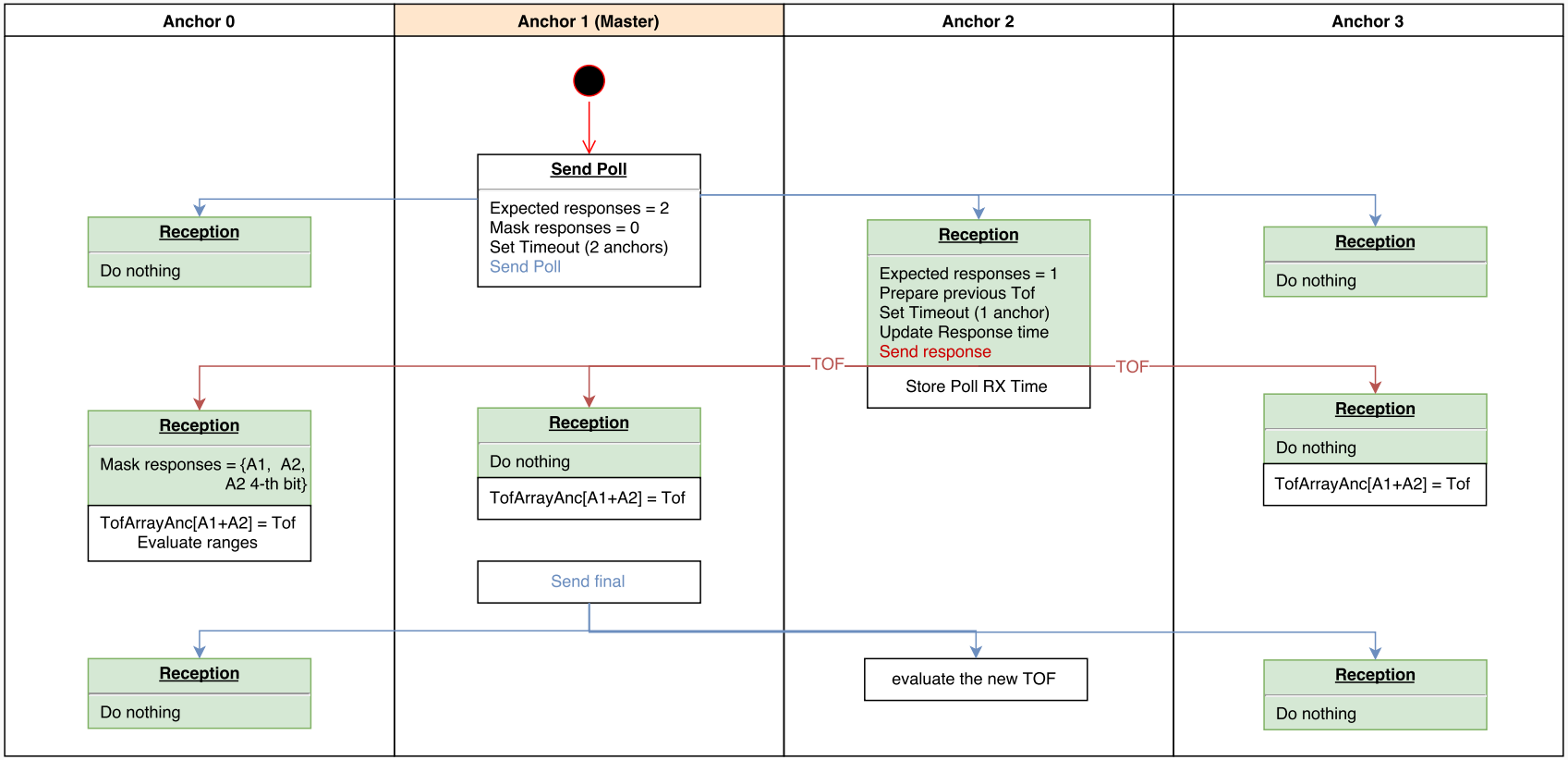
- FCODE = RTLS_DEMO_MSG_ANCH_RESP
- RES_TAG_SLP0: parte alta dello sleep correction calcolato dall'ancora 0
- RES_TAG_SLP1: parte bassa dello sleep correction calcolato dall'ancora 0
- PREVIOUS_TOF: tof calcolato al passo precedente dall'ancora
- rangeNum: range number della trasmissione

Descrizione procedura di Autoranging presente nel Firmware

Modalità di Autoranging inclusa nel Firmware



Modalità di Autoranging inclusa nel Firmware



Considerazioni

La procedura di autoranging presente nel firmware:

- calcola solo 3 su 6 dei range richiesti per valutare la posizione delle ancore;
- non spedisce i range calcolati / la loro media al tag;
- è implementata mediante parti di codice specifiche per ogni ancora quindi non facilmente estensibile al caso di più ancore;
- a parita di scelta del superframe period e dello slot period riduce di N il numero massimo di tag utilizzabili poiché la procedura avviene negli ultimi N[1] slot.

[1] nel caso del firmware $N = 2$

Nuova procedura di autoranging

Nuova procedura di autoranging

A differenza della procedura originale la nuova procedura non avviene durante la fase di ranging portata avanti dal tag ma avviene in una fase preliminare durante la quale il tag sta in attesa.

L'intera procedura si divide in 3 parti. Ogni ancora a turno:

1. valuta M volte i range di interesse, i.e. l'ancora j-esima raccoglie i range $r_{j,j+1}$ ... $r_{j,N-1}$ con N il numero di ancore e $j < N$;
2. calcola la media dei range raccolti;
3. invia i range medi al tag ogni volta che risponde ad un suo Poll.

Simmetria

In linea di principio quando l'ancora j-esima esegua la sua fase di raccolta dei range non sarebbe necessario che le ancore con indice $i < j$ partecipino alla procedura poiché sono di interesse solo i range $r_{j,k}$ con $j < k < N$. Tuttavia facendo partecipare sempre tutte le ancore è possibile raccogliere per M volte ciascun range eseguendo $M/2$ istanze di range per ogni ancora invece che M.

Numero di istanze di range richieste

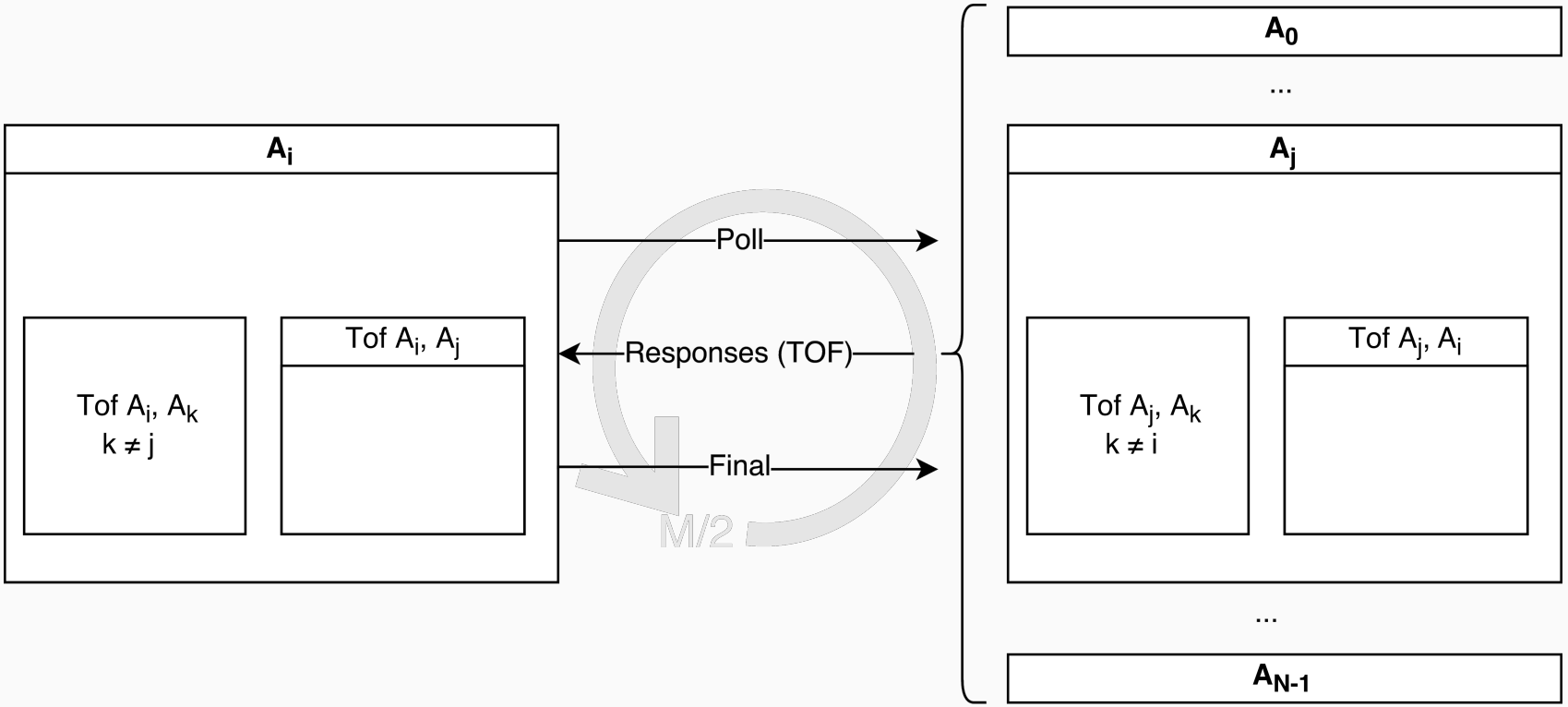
Date N ancore per raccogliere M volte ciascun range di interesse sono richieste

$$N\frac{M}{2}$$

istanze di ranging

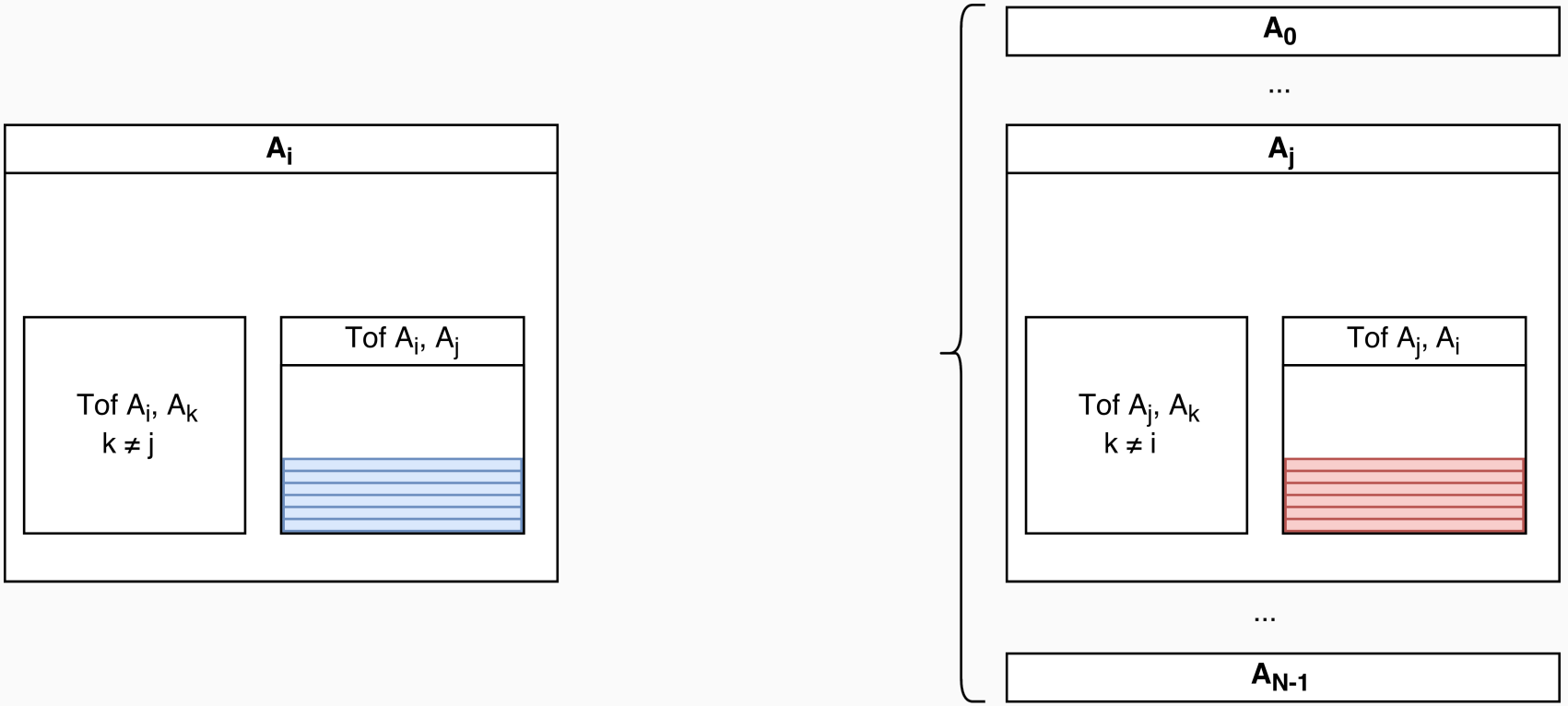
Simmetria - esempio

Sia $i < j$, l'ancora A_i esegue $M/2$ istanze di ranging



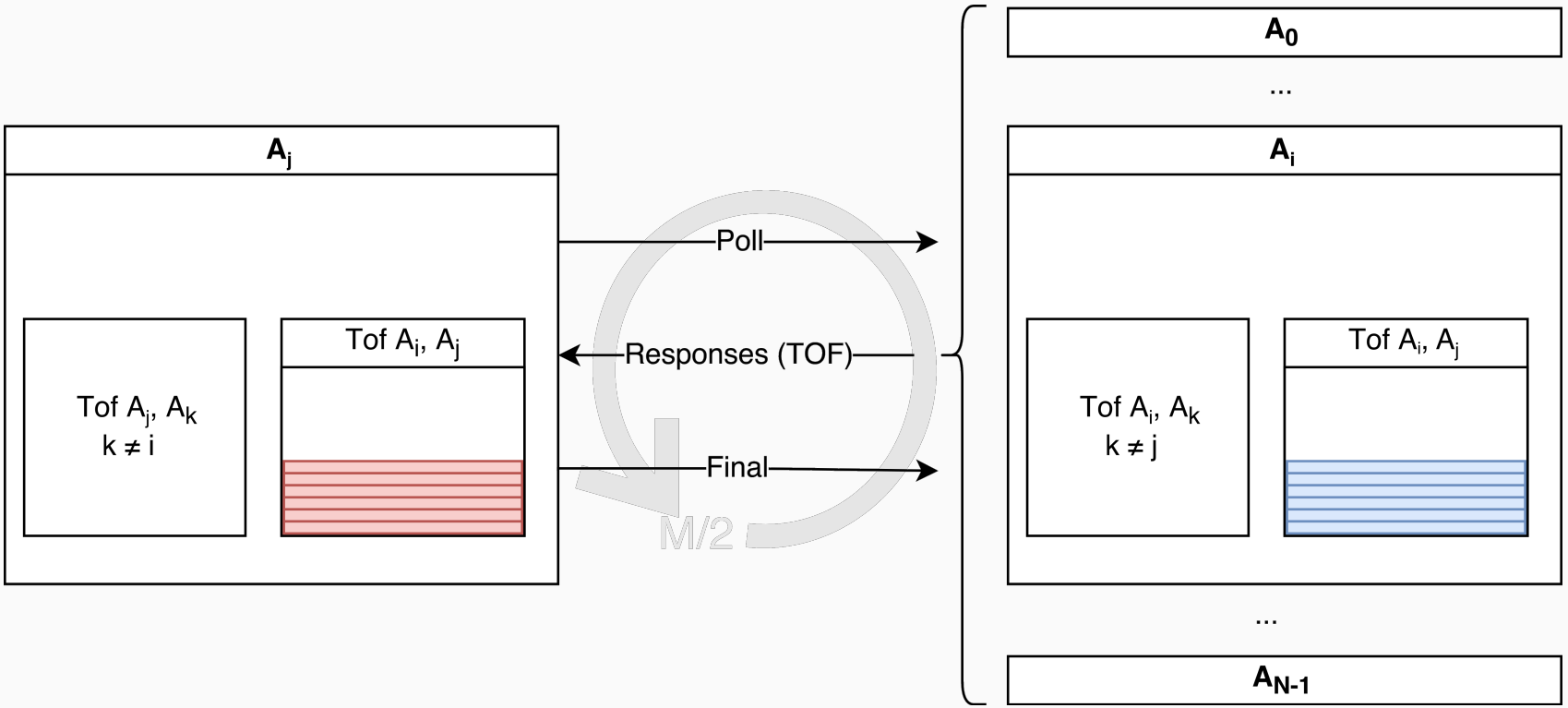
Simmetria - esempio

L'ancora A_i ha memorizzato $M/2$ misure di ranging



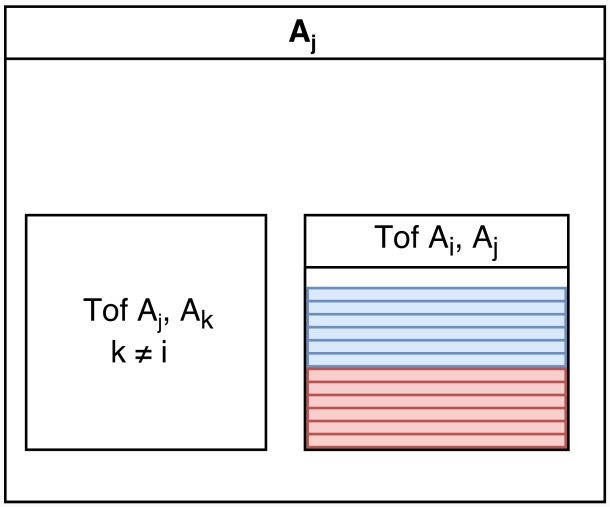
Simmetria - esempio

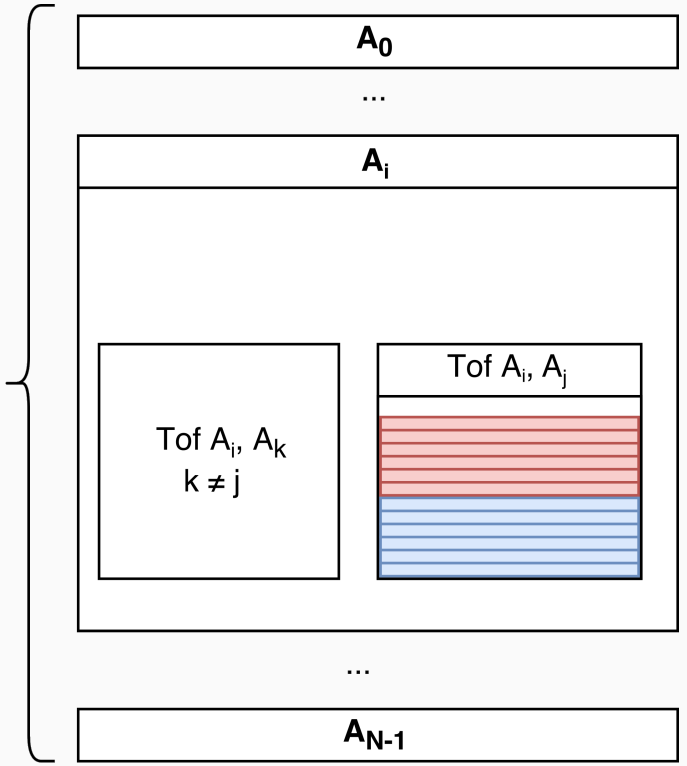
Successivamente l'ancora A_j esegue $M/2$ istanze di ranging



Simmetria - esempio

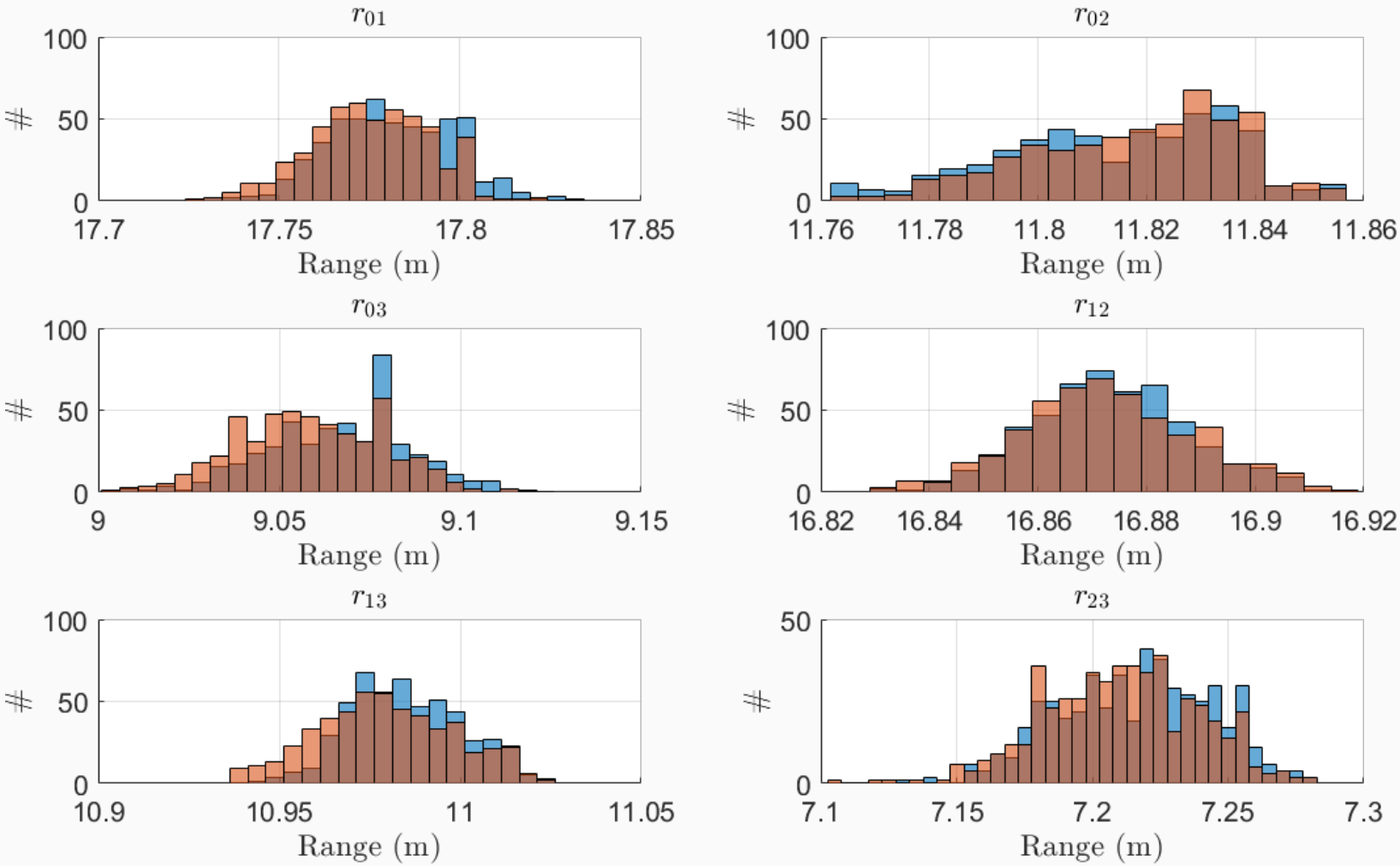
L'ancora A_i colleziona ulteriori $M/2$ misure





Simmetria - esempio

Le distribuzioni delle misure raccolte sfruttando la simmetria sono simili

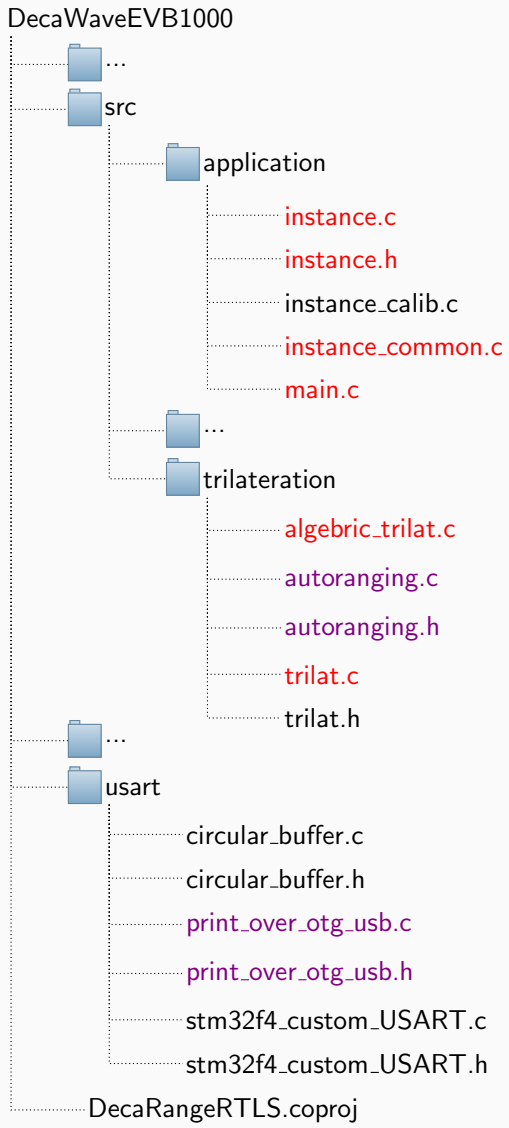


Guida al codice sviluppato

File modifica e file nuovi

Per implementare la nuova procedura di autoranging è stato necessario intervenire sul codice.

In rosso sono indicati i file modificati e in viola i nuovi file prodotti.



Header instance.h - Costanti

Sono state aggiunte le seguenti costanti nel file instance.h

```
#define NUM_AUTORNG_RNG ...
```

Definisce il numero M di misure raccolte durante l'autoranging da ogni ancora master per ogni range di interesse.
esempio: A0 raccoglie M misure di r_{01}, M misure di r_{02} ed M misure di r_{03}.

```
#define AUTORANGING_MAX_TIMEOUT_TAG ...
```

Tempo in ms dopo il quale il tag inizia la normale procedura di ranging se non riceve più messaggi di POLL inviati da un'ancora master.

```
#define ANCH_FINAL_MSG_LEN 34
```

Lunghezza del messaggio di tipo `RTLS_DEMO_MSG_ANCH_FINAL`.
Essa è pari a quella del messaggio di tipo `RTLS_DEMO_MSG_TAG_FINAL` più uno dato che contiene un byte in più che serve ad indicare alle ancore che l'autoranging per la corrente ancora master è terminato.

Header instance.h - Costanti

```
#define NUM_ALL_AUTORANING_RANGES
```

Numero complessivo di range che il tag si aspetta di ricevere da tutte le ancore.

Scritto in funzione del numero totale di ancore N.

esempio: per 4 ancore

$$|\{r_{01}, r_{02}, r_{03}, r_{12}, r_{13}, r_{23}\}| = 3 + 2 + 1 = 6$$

in generale:

$$\frac{(N-1)(N-1+1)}{2} = \frac{(N-1)N}{2}$$

Header instance.h - Costanti

```
#define END_AUTORANGING 33
```

Posizione, all'interno di `instance_data[0].msg_f.messageData`, del byte che indica la fine della procedura di autoranging per una data ancora master.
In questo caso messageData contiene un messaggio di tipo `RTLS_DEMO_MSG_ANCH_FINAL`.

```
#define AUTORANGING_RANGES 8
```

Posizione, all'interno di `instance_data[0].msg_f.messageData`, a partire dalla quale l'ancora inserisce le medie dei range che le competono.
In questo caso messageData contiene un messaggio di tipo `RTLS_DEMO_MSG_ANCH_RESP`.

Header instance.h - Struct `sfConfig_t`

Le seguenti `struct` sono state modificate

```
typedef struct
{
  ...
  uint16 tagPollSleepDly;
  uint16 anchPollSleepDly;
  ...
} sfConfig_t;
```

È stato aggiunto il campo `anchPollSleepDly` e il campo `PollSleepDly` è stato rinominato TagPollSleepDly per maggiore chiarezza.
Il campo `anchPollSleepDly` rappresenta il periodo in ms di trasmissione dei Poll dell'ancora master.

Header instance.h - Struct `instance_data_t`

```
typedef struct
{
   ...
} instance_data_t;
```

Sono stati aggiunti i seguenti campi

`int32 anchSleepTime_ms`

Periodo in ms di trasmissione dei Poll dell'ancora master

`unsigned long autoranging_poll_time`

Se il dispositivo si comporta come ancora in modalità `ANCHOR_RNG` ed è l'ancora master rappresenta l'istante di tempo, secondo l'orologio interno, in cui è stato spedito l'ultimo Poll. Utilizzato per capire se sono passati `anchSleepTime_ms` ms dall'invio dell'ultimo Poll.

Header instance.h - Struct `instance_data_t`

Se invece il dispositivo si comporta come tag in modalità `TAG_WAIT` rappresenta l'istante di tempo, secondo l'orologio interno, in cui il tag ha ricevuto l'ultimo Poll inviato da un'ancora master.
Utilizzato per capire se sono passati `AUTORANGING_MAX_TIMEOUT_TAG` ms dalla ricezione dell'ultimo Poll inviato da un'ancora master.

`uint8 autoranging_timeout`

Abilitazione del timer usato da un'ancora master per inviare periodicamente Poll alle altre ancore.
Abilitazione del timer usato da un tag in modalità `TAG_WAIT` per attendere la fine della procedura di autoranging.

Header instance.h - Struct `instance_data_t`

`double anchRngArray[MAX_ANCHOR_LIST_SIZE]`

Contiene la somma dei range raccolti da una data ancora.

esempio: sia l'ancora A1. `anchrRngArray[0]` contiene la somma dei range r_{01}, `anchrRngArray[2]` contiene la somma dei range r_{12} e `anchrRngArray[3]` contiene la somma dei range r_{13}.

`uint16 anchRngArrayCounter`

Contiene il numero di misure raccolte per ciascun range di interesse per una data ancora.

esempio: sia l'ancora A1. `anchrRngArrayCounter[0]` contiene il numero di misure del range r_{01}, `anchrRngArrayCounter[2]` contiene il numero di misure del range r_{12} e `anchrRngArrayCounter[3]` contiene il numero di misure del range r_{13}.

Header instance.h - Struct `instance_data_t`

`uint8 autoRngRangesRxMask`

Bitmask. Il bit i-esimo vale 1 se il tag ha ricevuto la medie dei range dall'ancora i-esima.

`double autoRngRangesArray[NUM_ALL_AUTORANGING_RANGES]`

Medie dei range ricevute dal tag.

esempio: Nel caso di 4 ancore `autoRngRangesArray[0]` $= r_{01}$, ..., `autoRngRangesArray[5]` $= r_{23}$.

`float anchorPositionMatrix[3 * MAX_ANCHOR_LIST_SIZE]`

Posizioni cartesiane delle ancore ottenute con apposito algoritmo a partire dalle medie dei range salvate in `autoRngRangesArray`.

in generale: `anchorPositionMatrix[i][j]` contiene la coordinata i-esima dell'ancora j-esima.

`uint8 tagPositionsSentToViewer`

Contatore utilizzato per inviare ad intervalli regolari la posizione delle ancore sulla porta seriale virtuale (USB OTG VCP). Tali posizioni sono utilizzate dal Viewer 3D.

`uint16 anchorRngMaster`

Contiene l'ID corrente dell'ancora master durante l'autoranging.

Header instance.h - Enum

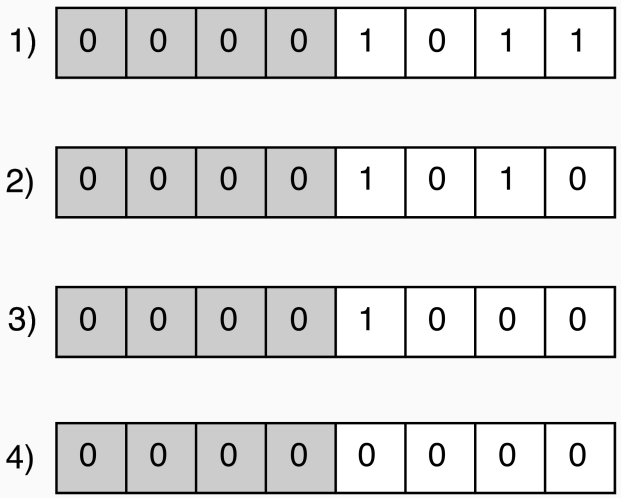
Il seguente enum è stato modificato

```
typedef enum instanceModes{LISTENER, TAG,
    TAG_WAIT, ANCHOR, ANCHOR_RNG, NUM_MODES}
    INST_MODE;
```

È stato aggiunto l'enumeratore `TAG_WAIT`. Esso corrisponde alla modalità in cui il tag attende il completamento della procedura di autoranging.

`rxResponseMaskAnc` - Lato ancora non master

L'invio delle risposte da parte delle ancore viene regolato dalla maschera `rxResponseMaskAnc`. Viene mostrato un esempio in cui l'ancora A2 si comporta da master.



1) la maschera viene inizializzata all'arrivo del Poll in tutte le ancore mettendo a 0 il bit corrispondente alla posizione dell'ancora master
2) dopo che l'ancora 0 ha risposto viene posto a 0 il bit corrispondente alla posizione dell'ancora
3) dopo che l'ancora 1 ha risposto viene posto a 0 il bit corrispondente alla posizione dell'ancora
4) dopo che l'ancora 3 ha risposto viene posto a 0 il bit corrispondente alla posizione dell'ancora

Regola generale

È il turno dell'ancora i-esima di rispondere se il bit in posizione i-esima è il primo diverso da 0 da destra.

File instance_common.c

`uint8 anch_has_responded()`

Restituisce `True` se l'istanza ha già risposto altrimenti restituisce `False`.

`uint8 anch_tx_or_wait()`

Restituisce `True` se l'istanza deve spedire la risposta al tag per poi aggiornare la maschera altrimenti restituisce `False`
. Attenzione: l'ancora i-esima deve rispondere se il bit in posizione i-esima della maschera `rxResponseMaskAnc` è il primo bit diverso da 0. da destra

`void handle_rx_mask_on_timout()`

Gestisce un evento di Timeout di ricezione risposta nella procedura di autoranging.
Trova la posizione del primo bit uguale a 1 nella maschera `rxResponseMaskAnc`. La posizione di questo bit corrisponde all'indice dell'ancora che avrebbe dovuto inviare la risposta. Successivamente setta il bit a 0 così da simulare l'avvenuta ricezione del messaggio.

File instance_common.c

`int eval_range(double* range, uint32 tof)`

Converte i tof in distanze espressi in m. Restituisce 0 se il risultato è negativo o maggiore di 20 km, 1 altrimenti.

`void instanceclearcounts()`

Rispetto alla versione originale di questa funzione viene aggiunta l'inizializzazione di `anchRngArray[i]` e `tofAnc` dove *i* varia da 0 a `MAX_ANCHOR_LIST_SIZE - 1`

```
instance_data[0].anchRngArray[i] = INVALID_TOF;
instance_data[0].tofAnc = INVALID_TOF;
```

File instance_common.c

`void instanceclearcounts()`

Rispetto alla versione originale di questa funzione viene anche inizializzato il contatore del numero di invii della posizione del Tag al Viewer.

```
instance_data[0].tagPositionSentToViewer = 0;
```

Attenzione: ogni `SEND_ANCHOR_POSITION_EVERY_CYCLES` volte che il tag invia la sua posizione al Viewer tramite seriale viene inviata la posizione delle ancore.

`void instance_set_anch_sleep_delay(int sleepdelay)`

Copia in `anchSleepTime_ms` il periodo di trasmissione di ogni Poll da parte dell'ancora master.

File instance_common.c

```
void enable_tag_polling(instance_data_t *inst)
```

Inizializza il Tag per la normale procedura di ranging. Viene chiamata dal tag quando si rende conto che la procedura di autoranging è terminata.

File instance_common.c

```
void instance_backtoanchor(instance_data_t *inst)
```

Configura un'ancora per funzionare nella modalità standard di ranging.

Reimposta i Timeout

```
dwt_setrxtimeout(0);
dwt_setpreambledetecttimeout(0);
dwt_setrxaftertxdelay(0);
```

e si occupa di informare l'utente, attraverso lo schermo LCD, che la procedura di autoranging è terminata

```
sprintf((char*)&buff[0], "A:%d AutoRng End",
    inst->instanceAddress16 & 0x3);
writetoLCD(16, 1, buff);
```

File instance_common.c

```
void eval_means(instance_data_t *inst)
```

Calcola le medie dei range accumulati durante la fase di autoranging

```
for(i = 0; i < MAX_ANCHOR_LIST_SIZE; i++)
{
  if(i == (inst->instanceAddress16 & 0x3))
    continue;
  inst->anchRngArray[i] /=
      inst->anchRngArrayCounter[i];
}
```

Durante il calcolo della media non viene considerato il range di un'ancora con se stessa.

File instance_common.c

```
void inst_processrxtimeout(instance_data_t *inst)
```

Gestisce il comportamento dell'ancora master in caso di Timeout.

Qualora lo stato precedente fosse `TA_TXPOLL_WAIT_SEND` prepara l'ancora all'invio di un nuovo Poll

```
inst->instToSleep = TRUE;
inst->testAppState = TA_TXE_WAIT;
inst->nextState = TA_TXPOLL_WAIT_SEND;
```

Qualora lo stato precedente fosse `TA_TXFINAL_WAIT_SEND` prepara l'ancora all'invio di un nuovo Poll

```
dwt_forcetrxoff();
inst->instToSleep = TRUE;
inst->testAppState = TA_TXE_WAIT;
inst->nextState = TA_TXPOLL_WAIT_SEND;
```

Infine gestisce il comportamento dell'ancora non master in caso di Timeout riabilitando immediatamente la ricezione

```
inst->testAppState = TA_RXE_WAIT;
dwt_setrxtimeout(0);
```

File instance_common.c

`uint8 anc_rx_reenable()`

Gestisce il comportamento dell'ancora master ad ogni ricezione di un messaggio di risposta.
La maschera `mask`

```
uint8 mask = (~(0x1 << instance_address)) & 0xF;
```

viene utilizzata per capire se tutte le ancore hanno risposto ed in tal caso l'istanza si prepara ad inviare il messaggio di Final.
Nel caso in cui solo alcune risposte sono arrivate e `instance_data[0].responseTO > 0` viene riconfigurato il Timeout e riabilitata la ricezione

```
dwt_setrxtimeout(fwtoTime_sy * responseTO);
dwt_rxenable(DWT_START_RX_IMMEDIATE);
```

File instance_common.c

`uint8 anc_rx_reenable()` - continuazione

Infine se, secondo la maschera, non tutti hanno risposto ma `instance_data[0].responseTO <= 0` viene rilevata una condizione di errore e viene comunque inviato il Final così da permettere alle ancore che hanno risposto di calcolare il TOF.

`uint8 anctxorrxreenable(uint16 sourceAddress)`

Eliminate le parti realtive alla procedura di autoranging già implementata nel firmware originario della DecaWave.

File instance_common.c

`uint8 anc_tx_or_rxreenable_autoranging()`

Gestisce i turni di invio delle risposte ad un Poll ricevuto da un'ancora master.
Qualora tutte le ancore avessero risposto (i.e. `instance_data[0].rxResponseMaskAnc == 0`) l'istanza torna nuovamente in ricezione, disabilitando tutti i Timeout, in attesa del messaggio di Final da parte dell'ancora master

```
dwt_setrxtimeout(0);
dwt_setpreambledetecttimeout(0);
```

Se, invece è il suo turno di rispondere (i.e. `anch_tx_or_wait()== True`) effettua un invio ritardato

```
dwt_setdelayedtrxtime(instance_data[0].delayedReplyTime);
dwt_starttx(DWT_START_TX_DELAYED | DWT_RESPONSE_EXPECTED);
```

Qualora non fosse ancora il momento di rispondere (i.e. `anch_tx_or_wait()== False`) riabilita immediatamente la ricezione

```
ancenablerx();
```

Attenzione: dopo aver chiamato `anch_tx_or_wait()` la `rxResponseMaskAnch` potrebbe essere cambiata ed essere diventata nulla ad indicare che tutte le ancore hanno risposto. In tal caso vengono disabilitati i timeout e l'istanza torna nuovamente in ricezione per attendere l'arrivo del messaggio di Final.

File instance_common.c

`void handle_error_unknownframe(event_data_t dw_event)`

Gestisce l'arrivo di un frame non identificato.
Nel caso in cui il tag sia in modalità `TAG_WAIT` gestisce l'arrivo di un messaggio con modalità di indirizzamento errata oppure il fatto che `rxd->event` sia diverso da `DWT_SIG_RX_OKAY` e da `DWT_SIG_RX_TIMEOUT` La chiamata a questa funzione comporta l'inserimento in coda di un evento di tipo `TIMEOUT`

```
dwt_setrxtimeout(0);
dwt_setpreambledetecttimeout(0);
...
instance_putevent(dw_event, DWT_SIG_RX_TIMEOUT);
```

File instance_common.c

```
void ancprepareresponse(uint16 sourceAddress, uint8 srcAddr_index, uint8
fcode_index, uint8 *frame, uint32 uTimeStamp)
```

È stata rimossa la parte che gestiva la procedura di autoranging già implementata nel firmware della DecaWave. Vengono inoltre copiate nel messaggio di risposta che viene inviato al tag le medie dei range calcolate durante la procedura di autoranging

```
for(i = anc_addr + 1, j = 0; i < MAX_ANCHOR_LIST_SIZE; i++,
    j++)
{
  uint8 msg_index = AUTORANGING_RANGES + j * sizeof(double);
  memcpy(&(instance_data[0].msg_f.messageData[msg_index]),
      &instance_data[0].anchRngArray[i], sizeof(double));
}
```

- l'ancora A0 invia i range $r_{0,1}$, $r_{0,2}$ e $r_{0,3}$
- l'ancora A1 invia i range $r_{1,2}$ e $r_{1,3}$
- l'ancora A2 invia il range $r_{2,3}$

File instance_common.c

```
void anc_prepare_response_autoranging(uint8 srcAddr_index,
uint8 fcode_index, uint8 *frame)
```

Gestisce la preparazione del messaggio di risposta di un'ancora ad un Poll ricevuto da un'ancora master.

File instance_common.c

La funzione

```
void instance_rxcallback(const dwt_callback_data_t *rxd) ...
```

File instance_common.c - funzione `instance_rxcallback`

fcode `RTLS_DEMO_MSG_TAG_POLL`

Nel caso in cui un tag sia in modalità `TAG_WAIT` inizia la normale procedura di ranging senza aspettare il timeout di 10 s

```
enable_tag_polling(&instance_data[0]);
instance_data[0].autoranging_timeout = FALSE;
```

File instance_common.c - funzione `instance_rxcallback`

fcode `RTLS_DEMO_MSG_ANCH_POLL`

Nel caso in cui un tag sia in modalità `TAG_WAIT` sono settati a zero i timeout di ricezione

```
dwt_setrxtimeout(0);
dwt_setpreambledetecttimeout(0);
```

Nel caso in cui il messaggio sia ricevuto da un'ancora in modalità `ANCHOR_RNG` viene settata la maschera `rxResponseMaskAnc`

```
instance_data[0].rxResponseMaskAnc = (~(0x1 << master_anchor)) & 0xF;
```

viene preparata la risposta con l'ultimo TOF calcolato

```
anc_prepare_response_autoranging(srcAddr_index, fcode_index,
    &dw_event.msgu.frame[0]);
```

configurato il timeout

```
dwt_setrxtimeout((uint16)instance_data[0].fwtoTimeAnc_sy);
```

e gestito l'invio del messaggio

```
anc_tx_or_rxreenable_autoranging(instance_data[0].instanceAddress16);
```

File instance_common.c - funzione `instance_rxcallback`

fcode `RTLS_DEMO_MSG_ANCH_RESP2`

Nel caso in cui un tag sia in modalità `TAG_WAIT` sono settati a zero i timeout di ricezione

```
dwt_setrxtimeout(0);
dwt_setpreambledetecttimeout(0);
```

Nel caso in cui il messaggio sia ricevuto da un'ancora master in modalità `ANCHOR_RNG` viene aggiornata la maschera e decrementato il contatore

```
instance_data[0].responseTO--;
instance_data[0].rxResponseMaskAnc |= (0x1 << (sourceAddress & 0x3));
```

Viene inoltre gestito il comportamento dell'ancora master (i.e. continua ad aspettare ed invia il messaggio di Final)

```
anc_rx_reenable();
```

Nel caso in cui il messaggio sia ricevuto da un'ancora, non master in modalità `ANCHOR_RNG`, la cui maschera sia diversa da 0 (i.e. non sono state ricevute tutte le risposte dalle altre ancore) `rxResponseMaskAnc` viene aggiornata e l'invio del messaggio viene gestito

```
instance_data[0].rxResponseMaskAnc &= ~(0x1 << (sourceAddress & 0x3));
anc_tx_or_rxreenable_autoranging();
```

Il caso in cui la maschera fosse completamente nulla l'istanza potrebbe non aver ricevuto un messaggio di Poll oppure tutte le risposte sono state già ricevute, in questo caso viene resettato il Timeout e riabilitata immediatamente la ricezione

```
dwt_setrxtimeout(0);
dwt_rxenable(DWT_START_RX_IMMEDIATE);
```

File instance_common.c - funzione `instance_rxcallback`

condizione `rxd->event == DWT_SIG_RX_TIMEOUT`

Nel caso in cui un'ancora non master in modalità `ANCHOR_RNG` non abbia ricevuto la risposta attesa e il Timeout sia scattato viene aggiornata la `rxResponseMaskAnc` e, nel caso sia arrivato il turno per l'istanza di inviare la risposta, risponde all'ancora master così da preservare il funzionamento della procedura di autoranging

```
handle_rx_mask_on_timout();
anc_tx_or_rxreenable_autoranging();
```

Viene inoltre generato un evento di tipo `DWT_SIG_RX_TIMEOUT`

```
instance_putevent(dw_event, DWT_SIG_RX_TIMEOUT);
```

File instance_common.c

```
int instance_run()
```

Nel caso in cui l'uscita della macchina a stati sia `INST_DONE_WAIT_FOR_NEXT_EVENT_TO` viene reimpostato il Timeout per l'invio di un nuovo Poll da parte dell'ancora master

```
nextPeriod = instance_data[0].anchSleepTime_ms;
instance_data[0].nextSleepPeriod = (uint32)
    nextPeriod;
```

Nel caso `TAG_WAIT` viene gestito il Timeout che consente di iniziare la normale procedura di ranging quando non viene più rilevata attività di autoranging

```
enable_tag_polling(&instance_data[0]);
instance_data[instance].autoranging_timeout =
    FALSE;
```

File instance.c - `testapprun`

La funzione

```
int testapprun(instance_data_t *inst, int message)
```

contiene una macchina a stati che implementa la logica di funzionamento delle ancore e dei tag. Essa è stata modificata al fine di rimuovere la vecchia modalità di autoranging ed inserire la nuova.
Nel seguito sono descritti i principali cambiamenti apportati.

Stato TA_INIT

Un tag in modalità TAG_WAIT abilita la ricezione di eventuali messaggi provenienti dalle ancore che eseguono l'autoranging.
Il codice

```
inst->autoranging_timeout = TRUE;
inst->autoranging_poll_time =
    portGetTickCount();
```

abilita il timeout software utilizzato per attendere la fine della procedura di autoranging e memorizza l'istante di tempo corrente nel caso in cui la procedura di autoranging fosse terminata prima dell'accensione del tag.

Stato TA_INIT - continuazione

Di solito (vedi ??), infatti, autoranging_poll_time viene assegnato alla ricezione da parte del tag di un Poll inviato da un'ancora master.

Il codice

```
inst->autoRngRangesRxMask = 0;
```

inizializza la maschera che indica da quali ancore il tag ha ricevuto i range medi.

Lo stato successivo è TA_RXE_WAIT.

Stato TA_INIT - continuazione

Un'ancora in modalità ANCHOR_RNG si comporta diversamente a seconda che sia l'ancora master o meno.
L'ancora A0 viene configurata come ancora master e preparata per eseguire il primo sleep prima dell'invio del primo Poll. Il codice

```
for(i=0; i<MAX_ANCHOR_LIST_SIZE; i++)
    inst->anchRngArrayCounter[i] = 0;

uint16 instance_address =
    inst->instanceAddress16 & 0x3;
inst->anchRngArrayCounter[instance_address] =
    0xFFFF - 0x1;
```

si occupa di inizializzare il contatore delle misure raccolte per ogni range di interesse.

Stato TA_INIT - continuazione

Per ogni ancora master, sia l'i-esima, $i \geq 0$, la procedura di autoranging si considera terminata quando l'elemento più piccolo dell'array anchRngArrayCounter è maggiore o uguale a NUM_AUTORNG_RGN. Dal momento che l'elemento i-esimo dell'array, che si riferisce al range r_{ii}, non viene mai aggiornato esso viene posto al massimo numero rappresentabile su 16 bit.

Lo stato successivo è TA_TXE_WAIT.

Stato TA_INIT - continuazione

Le altre ancore, con indirizzo diverso da A0, abilitano la ricezione e rimangono in attesa di messaggi di Poll inviati dall'ancora master.

Lo stato successivo è TA_RXE_WAIT.

File instance.c - testapprun

Stato TA_SLEEP_DONE

In questo stato i tag, nel loro funzionamento normale, e le ancore master, in modalità ANCHOR_RNG, attendono che sia il momento di inviare un nuovo Poll.
Per le ancore è stato deciso di non utilizzare la modalità DEEP_SLEEP per cui, pur entrando nel ramo DEEP_SLEEP == 1, viene replicata l'istruzione Sleep(3) prevista nel ramo #else.

```
#if (DEEP_SLEEP == 1)
    if (inst->mode == TAG)
    {
      ...
    }
    else if (inst->mode == ANCHOR_RNG)
        Sleep(3);
#else
    Sleep(3);
#endif
```

Stato TA_TXE_WAIT

In questo stato i tag, nel loro funzionamento normale, e le ancore master, in modalità ANCHOR_RNG, eseguono alcune operazioni in preparazione alla spedizione del successivo Poll.
In particolare con le istruzioni

```
if (inst->mode == ANCHOR_RNG)
    inst->rangeNumAnc++;
...
else if (inst->mode == ANCHOR_RNG)
    inst->rxResponseMaskAnc = 0;
```

l'ancora master, replicando il comportamento di un tag, incrementa il rangeNumber, utilizzato nel protocollo di trasmissione per eseguire un integrity check, e riporta a zero la maschera che indica quali ancore hanno risposto al Poll inviato.

Stato TA_TX_POLL_WAIT_SEND

In questo stato i tag, nel loro funzionamento normale, e le ancore master, in modalità ANCHOR_RNG, spediscono effettivamente il Poll.
In particolare con le istruzioni

```
inst->responseTO = MAX_ANCHOR_LIST_SIZE - 1;
dwt_setrxtimeout((uint16)inst->fwtoTime_sy *
    (MAX_ANCHOR_LIST_SIZE - 1));
```

l'ancora master, replicando il comportamento di un tag, configura un timeout di ricezione di durata proporzionale al numero di risposte attese MAX_ANCHOR_LIST_SIZE - 1. Tale numero è inferiore di uno rispetto al numero di risposte attese normalmente da un tag dato che una delle ancore si comporta come master.

Stato TA_TX_FINAL_WAIT_SEND

In questo stato i tag, nel loro funzionamento normale, e le ancore master, in modalità ANCHOR_RNG, spediscono effettivamente il Final. Dato che l'ancora master invia un byte in più per specificare se la propria procedura di autoranging è terminata è necessario modificare la lunghezza del messaggio con l'istruzione

```
inst->psduLength = (ANCH_FINAL_MSG_LEN +
    FRAME_CRTL_AND_ADDRESS_S + FRAME_CRC);
```

dove ANCH_FINAL_MSG_LEN vale 34 invece che 33 come nel caso del tag.

Stato TA_TX_FINAL_WAIT_SEND - continuazione

Con le istruzioni

```
inst->msg_f.messageData[END_AUTORANGING] =
    FALSE;
...
if(inst->mode == ANCHOR_RNG &&
array_min(inst->anchRngArrayCounter,
    MAX_ANCHOR_LIST_SIZE) >= NUM_AUTORNG_RNG)
{
  inst->anchorRngMaster++;
  inst->msg_f.messageData[END_AUTORANGING] =
      TRUE;
}
```

viene modificato il flag che indica se la procedura di autoranging è terminata e viene incrementata la variabile contenente l'ID dell'ancora master corrente.

File instance.c - testapprun

Stato TA_TX_FINAL_WAIT_CONF

In questo stato i tag, nel loro funzionamento normale, e le ancore master, in modalità ANCHOR_RNG, verificano l'avvenuta spedizione del Poll e del Final.
Per quanto riguarda l'ancora master, nel caso in cui stia verificando l'avvenuta spedizione di un Poll, si comporta come un tag.
Se invece l'ancora master sta verificando la spedizione di un Final e viene soddisfatto il criterio di termine della procedura di autoranging allora si porta nello stato di ricezione TA_RXE_WAIT in cui attenderà la ricezione di nuovi Poll provenienti dalla nuova ancora master.

```
if(inst->previousState == TA_TXFINAL_WAIT_SEND)
{
  if(inst->mode == ANCHOR_RNG &&
  array_min(inst->anchRngArrayCounter, MAX_ANCHOR_LIST_SIZE) >=
      NUM_AUTORNG_RNG)
  {
    ...
    inst->testAppState = TA_RXE_WAIT;
    ...
```

File instance.c - testapprun

Stato TA_TX_FINAL_WAIT_CONF - continuazione

Inoltre se, dopo aver incrementato il contatore anchorRngMaster, l'indirizzo della nuova ancora master è pari al numero totale di ancore allora la procedura complessiva di autoranging è terminata ed è possibile ripristinare la modalità normale di funzionamento ANCHOR attraverso la chiamata alla funzione instance_backtoanchor().

```
if((inst->anchorRngMaster & 0x7) == MAX_ANCHOR_LIST_SIZE))
  instance_backtoanchor(inst);
...
}
```

Solo l'ultima ancora chiama la funzione instance_backtoanchor() in questa parte della macchina a stati. Le altre ancore rilevano il termine della procedura di autoranging quando ricevono l'ultimo final inviato dall'ultima ancora nello stato TA_RX_WAIT_DATA.

File instance.c - `testapprun` - `TA_RX_WAIT_DATA`

Lo stato `TA_RX_WAIT_DATA`, assieme alla callback di ricezione `instance_rxcallback()` contenuta nel file `instance_common.c`, si occupa di processare i messaggi ricevuti dal dispositivo, ancora o tag, e ne modifica il comportamento di consenguenza.
Nel caso in cui il dispositivo abbia ricevuto un messaggio valido, lo stato `TA_RX_WAIT_DATA` prevede comportamenti diversi in base all'`fcode` contenuto nel messaggio.
Di seguito sono descritti i principali cambiamenti apportati a questo stato suddivisi per `fcode`.

File instance.c - testapprun - TA_RX_WAIT_DATA

Il firmware originale gestiva gli fcode RTLS_DEMO_MSG_TAG_POLL e RTLS_DEMO_MSG_ANCH_POLL assieme. Nella nuova release sono gestiti separatamente.

fcode RTLS_DEMO_MSG_ANCH_POLL

Le ancore che ricevono un Poll da parte di un'ancora master si comportano come le ancore che ricevono un Poll da parte di un tag nella normale procedura di ranging.
Un tag nella modalità TAG_WAIT reimposta il tempo di ricezione del Poll in modo da garantire il funzionamento del sistema di timeout con cui il tag si rende conto del termine della procedura di autoranging. In questa situazione il tag torna ad ascoltare nello stato TA_RXE_WAIT.

```
...
if (inst->mode == TAG_WAIT)
{
  inst->autoranging_poll_time = portGetTickCount();
  inst->testAppState = TA_RXE_WAIT;

  break;
}
...
```

File instance.c - testapprun - TA_RX_WAIT_DATA

Il firmware originale gestiva gli fcode RTLS_DEMO_MSG_ANCH_RESP e _RESP2 assieme. Nella nuova release sono gestiti separatamente.

fcode RTLS_DEMO_MSG_ANCH_RESP

Il tag che riceve una risposta da parte di un'ancora si occupa anche di gestire i range medi elaborati durante la fase di autoranging e che ogni ancora si occupa di inviare al tag in risposta ad un Poll (oltre al Tof come previsto nel funzionamento normale della procedura di ranging).
In particolare se il tag non ha ancora ricevuto i range medi dall'ancora avente indirizzo srcAddr[0]

```
if ((inst->mode == TAG) &&
    (inst->autoRngRangesRxMask & (0x1 <<
    (srcAddr[0]&0x3))) == 0)
{
```

fcode RTLS_DEMO_MSG_ANCH_RESP - continuazione

valuta il numero di range che si aspetta di ricevere da tale ancora

```
uint8 number_of_ranges = MAX_ANCHOR_LIST_SIZE
    - 1 - (srcAddr[0]&0x3);
```

e copia i dati dal messaggio appena ricevuto nella struttura dati autoRngRangesArray a partire dal byte offset-esimo.

```
uint8 offset = 0;
for(i = 0; i < (srcAddr[0]&0x3); i++)
  offset += (MAX_ANCHOR_LIST_SIZE - 1 - i);

memcpy(&(inst->autoRngRangesArray[offset]),
    &(messageData[AUTORANGING_RANGES]),
sizeof(double) * number_of_ranges);
```

fcode RTLS_DEMO_MSG_ANCH_RESP - **continuazione**

Inoltre aggiorna la maschera autoRngRangesRxMask in modo da non ripetere la procedura di salvataggio dei range medi alla ricezione di altre risposte dalla medesima ancora.

```
inst->autoRngRangesRxMask |= (0x1 << (srcAddr[0]&0x3));
```

Se il tag ha ricevuto tutti i range medi da tutte le ancora valuta le posizioni cartesiane delle ancore attraverso la chiamata alla funzione rangeToPos che salva il risultato nella matrice anchorPositionMatrix.

```
    if (inst->autoRngRangesRxMask  == (0x1 <<
        MAX_ANCHOR_LIST_SIZE) - 1)
    {
      rangesToPos(inst->autoRngRangesArray,\
      inst->anchorPositionMatrix);
    }
  }
  ...
```

File instance.c - testapprun - TA_RX_WAIT_DATA

Il firmware originale gestiva gli fcode RTLS_DEMO_MSG_ANCH_RESP e RTLS_DEMO_MSG_ANCH_RESP2 assieme. Nella nuova release sono gestiti separatamente.

fcode RTLS_DEMO_MSG_ANCH_RESP2

Un tag in TAG_WAIT ignora il messaggio e riabilita la ricezione.
Le ancore, master o meno, si comportano come in una normale procedura di ranging eccetto per il fatto che l'ancora master salva il tof ricevuto in un accumulatore anchRngArray e incrementa il contatore anchRngArrayCounter.

```
...
if (inst->anchRngArrayCounter[src_addr] == 0)
  inst->anchRngArray[src_addr] = range;
else
  inst->anchRngArray[src_addr] += range;
inst->anchRngArrayCounter[src_addr]++;
...
```

fcode RTLS_DEMO_MSG_TAG_FINAL **e** RTLS_DEMO_MSG_ANCH_FINAL

Un tag in TAG_WAIT e riabilita la ricezione.

Un'ancora non master che riceve un ANCH_FINAL calcola il tof, come farebbe nel caso in cui ha ricevuto un TAG_FINAL, lo salva in un accumulatore anchRngArray ed incrementa il contatore anchRngArrayCounter

```
if (inst->anchRngArrayCounter[src_addr] == 0)
  inst->anchRngArray[src_addr] = range;
else
  inst->anchRngArray[src_addr] += range;
inst->anchRngArrayCounter[src_addr]++;
```

fcode RTLS_DEMO_MSG_ANCH_FINAL - **continuazione**

Inoltre l'ancora verifica se il messaggio di Final ricevuto contiene il byte che indica la fine della procedura di autoranging da parte dell'ancora master corrente. Nel caso in cui la procedura sia terminata viene incrementato l'indirizzo dell'ancora master corrente. Inoltre viene invalidato il tof dal momento che non potrebbe essere utilizzato nella risposta ad un Poll proveniente dalla successiva ancora master dato che è riferito all'ancora master precedente.

```
if(messageData[END_AUTORANGING] == TRUE)
{
  inst->anchorRngMaster++;
  inst->autoranging_timeout = FALSE;
  inst->tofAnc = INVALID_TOF;
}
```

File instance.c - testapprun - TA_RX_WAIT_DATA

fcode RTLS_DEMO_MSG_ANCH_FINAL - **continuazione**

Qualora l'ancora che ha ricevuto il final fosse la nuova ancora master si prepara per spedire il suo primo Poll (le inizializzazioni sono le medesime eseguite dall'ancora 0 nello stato TA_INIT).

```
if(inst->mode == ANCHOR_RNG && inst->instanceAddress16 ==
    inst->anchorRngMaster)
{
  ...
  inst->autoranging_timeout = FALSE;
  inst->nextState = TA_TXPOLL_WAIT_SEND;
  inst->testAppState = TA_TXE_WAIT;

  inst->rangeNumAnc = 0;

  for(i=0; i<MAX_ANCHOR_LIST_SIZE; i++)
      inst->anchRngArrayCounter[i] = 0;

  uint16 instance_address = inst->instanceAddress16 & 0x3;
  inst->anchRngArrayCounter[instance_address] = 0xFFFF - 0x1;
  ...
}
```

fcode RTLS_DEMO_MSG_ANCH_FINAL - **continuazione**

Se al contrario l'ancora rileva che l'intera procedura di autoranging è completata, poichè l'indirizzo della nuova ancora master è pari al numero totale di ancore, calcola le medie dei range attraverso la funzione `eval_means()` e ripristina la modalità normale di funzionamento ANCHOR attraverso la funzione `instance_backtoanchor()`.

```
...
else if((inst->mode == ANCHOR_RNG) &&
    (inst->anchorRngMaster & 0x7) ==
    MAX_ANCHOR_LIST_SIZE)
{
  eval_means(inst);
  instance_backtoanchor(inst);
}
...
```

File instance.c - `array_min`

La seguente funzione è stata aggiunta

```
uint16 array_min (uint16* array, int len)
```

Calcola il minimo di un array.

File main.c

Nel seguito sono riportate alcune note riguardanti il file `main.c`

Configurazione Slot and Superframe

In generale è stato aggiunto il campo relativo al periodo di trasmissione dei Poll da parte di un'ancora master. È stato verificato sperimentalmente che un periodo di 10 ms è sufficiente a gestire 4 ancore di cui una master.

```
sfConfig_t sfConfig[4] =
{
...
//mode 2 - S1: 2 on, 3 off
{
  (10),   // slot period (ms)
  (1),    // number of slots
  (10*1), // superframe period (ms)
  (10*1), // poll sleep delay (ms)
  (10),   // anch sleep delay in autoranging (ms)
  (2500)  // final transmission delay
},
...
};
```

File main.c

`uint32 inittestapplication(uint8 s1switch)`

La modalità di funzionamento di un dispositivo DecaWaveEVB1000 dipende dallo switch *n*.4 presente sul PCB.
Un dispositivo di tipo tag, nella nuova release, non viene più inizializzato in modalità TAG ma TAG_WAIT. In tale modalità attende il termine della procedura di autoranging.
Un dispositivo di tipo ancora, nella nuova release, non viene più inizializzato in modalità ANCHOR ma ANCHOR_RNG. In tale modalità le ancore eseguono la procedura di autoranging.

```
if((s1switch & SWS1_ANC_MODE) == 0)
{
  instance_mode = TAG_WAIT;
}
else
{
  instance_mode = ANCHOR_RNG;
}
```

File main.c

`void setLCDline1(uint8 s1switch)`

Questa funzione gestisce il testo riportato sul display LCD a seconda della modalità del dispositivo DecaWaveEVB1000.
Nella nuova release del firmware i tag all'accensione riportano la scritta "Waiting..." ad indicare che attende il termine della procedura di autoranging. Le ancore invece riportano la scritta "Autoranging".

```
...
else if(role == TAG_WAIT)
{
  sprintf((char*)&dataseq1[0], "Tag:%d Waiting...",
      tagaddr);
  writetoLCD( 16, 1, dataseq1);
}
...
else if(role == ANCHOR_RNG)
{
  sprintf((char*)&dataseq1[0], "A:%d Autoranging",
      ancaddr);
  writetoLCD( 16, 1, dataseq1);
}
...
```

File main.c

`int main(void)`

Rispetto alla release precedente, il tag, dopo aver ricevuto i tof da tutte le ancore, invia il risulutato della trilaterazione attraverso la porta seriale (USB OTG VCP)

```
...
if(rx == TOF_REPORT_T2A)
{
  if (instance_data[0].mode == TAG)
  {
    ...
    instance_data[0].tagPositionSentToViewer++;

     float pos_x = (float) best_solution.x;
     float pos_y = (float) best_solution.y;
     float pos_z = (float) best_solution.z;
     print_over_otg_usb("tpr %d %08x %08x %08x",
                        (instance_data[0].instanceAddress16 & 0x7),
                        *(uint32_t*)&pos_x,
                        *(uint32_t*)&pos_y,
                        *(uint32_t*)&pos_z);
```

File main.c

`int main(void)` - continuazione

Inoltre ogni `SEND_ANCHOR_POSITION_EVERY_CYCLES` cicli invia sulla medesima porta la posizione cartesiana delle ancore.

```
        if(instance_data[0].tagPositionSentToViewer ==
            SEND_ANCHOR_POSITION_EVERY_CYCLES)
        {
          instance_data[0].tagPositionSentToViewer = 0;
          print_over_otg_usb("apr %d %08x %08x %08x %08x %08x %08x %08x
              %08x %08x %08x %08x %08x",
                             (instance_data[0].instanceAddress16 & 0x7),
                              *(uint32_t*)&(instance_data[0].
                              anchorPositionMatrix[0][0]),
                              ...
                              *(uint32_t*)&(instance_data[0].
                              anchorPositionMatrix[2][3]));
        }
        ...
      }
```

Files autoranging.c e autoranging.h

La funzione

```
int rangesToPos(double* ranges, float
    anch_position[][4])
```

è stata implementata sfruttando l'algoritmo algebrico di calibrazione sviluppato da Federica Fioretti.

File trilat.c

La funzione

```
int computePosition_mm(float d1,float d2,float
    d3,float d4, vec3d *best_solution2, float
    anch_position[][4])
```

utilizza la posizione delle ancore in m calcolate mediante la funzione `rangesToPos`

```
// Anchor i
pi.x = anch_position[0][i];
pi.y = anch_position[1][i];
pi.z = anch_position[2][i];
```

File algebric_trilat.c

La funzione

```
int computePosition_mm(float d1,float d2,float
    d3,float d4, vec3d *best_solution2, float
    anch_position[][4])
```

utilizza la posizione delle ancore in mm calcolate mediante la funzione `rangesToPos`

```
for(index = 0; index<3; index++)
{
  A_mm[index] = anch_position[index][0] * 1000;
  B_mm[index] = anch_position[index][1] * 1000;
  C_mm[index] = anch_position[index][2] * 1000;
  D_mm[index] = anch_position[index][3] * 1000;
}
```

File print_over_otg_usb.c

La funzione

```
void print_over_otg_usb(const char * format, ...)
```

permette di inviare una stringa, utilizzando il formato della funzione `printf()`, tramite la porta USB OTG.

Viewer 3D

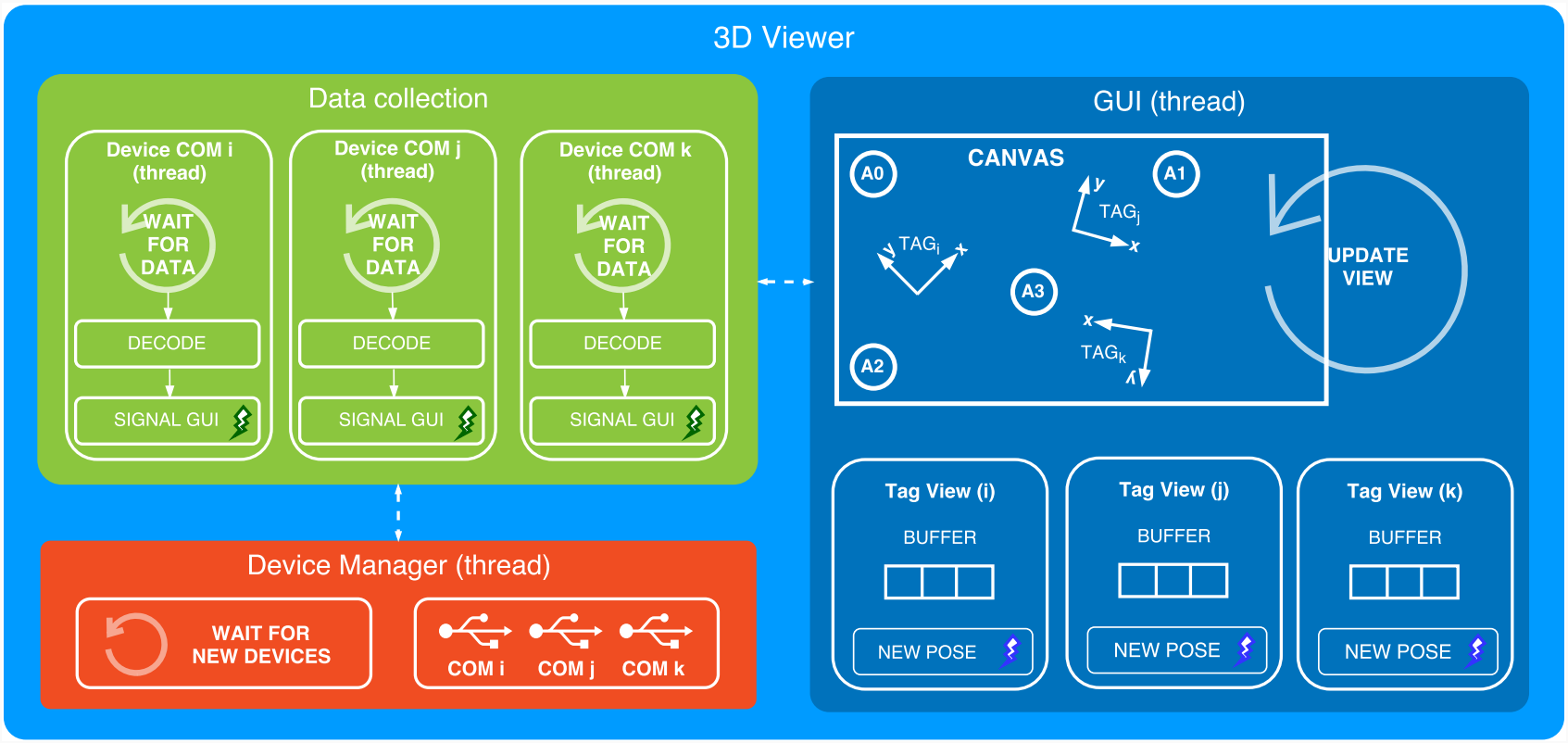
Viewer 3D

Al fine di valutare qualitativamente la validità dell'algoritmo di trilaterazione durante l'esecuzione di un esperimento è stato sviluppato un viewer 3D.
Tale viewer consente di visualizzare la posizione cartesiana di uno o più tag mediante uno scatter plot.
Inoltre il software è in grado di ricevere da uno dei tag la posizione delle ancore ottenuta mediante la procedura di calibrazione (procedura F. Fioretti) e di visualizzarne la posizione.
Nel seguito è descritta brevemente l'architettura del software ed i metodi più importanti delle classi Python che ne costituiscono l'implementazione.

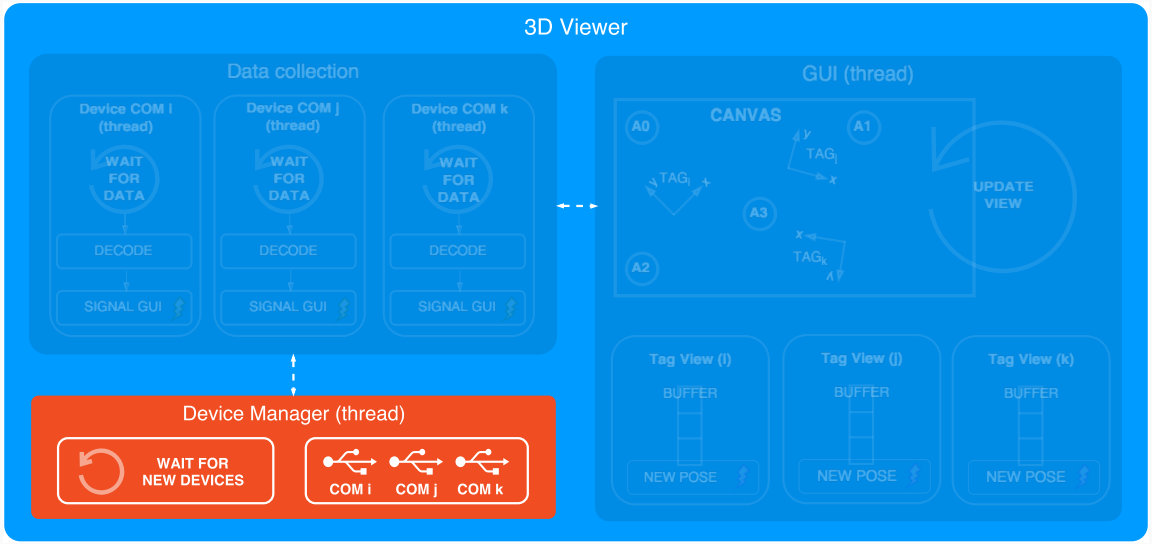
Architettura

L'architettura prevede tre livelli:

- gestione dei dispositivi (Device Manager)
- raccolta e decodifica dei dati (Device *i*-esimo)
- visualizzazione dei dati (GUI, Canvas)

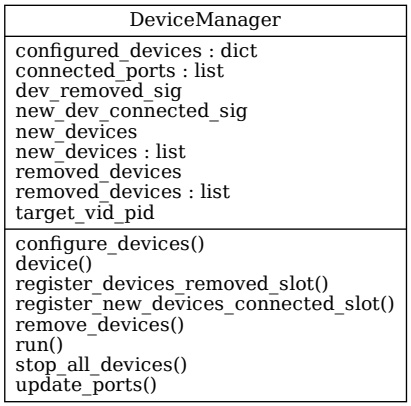


Architettura - Device Manager



Il Device Manager si occupa di rilevare periodicamente nuovi dispositivi connessi al computer tramite porta seriale (reale o virtuale).
Per ogni nuovo dispositivo connesso vengono allocate apposite strutture dati e creato un thread che si occupa della gestione dei dati in arrivo.
Quando un dispositivo viene disconnesso il thread ad esso associato viene terminato.

Implementazione - classe `DeviceManager` - metodo `run()`



Il metodo `run` è il metodo principale associato al thread del Device Manager.

```
    def run(self):
        while True:
            new_ports, removed_ports = self.update_ports()
```

Il metodo `update_ports()` rileva le nuove porte seriali connesse e le porte che sono state scollegate di recente.

Per ogni nuova porta la funzione `configure_devices()` si occupa di generare un'istanza della classe `Device`.

```
            if new_ports:
                self.new_devices = self.configure_devices(new_ports)
```

Implementazione - classe `DeviceManager` - metodo `run()`

Inoltre viene emesso un segnale per notificare l'evento alla GUI. Nel seguito verrà spiegato in che modo la GUI si registra agli eventi emessi dal Device Manager.

```
self.new_dev_connected_sig.emit()
```

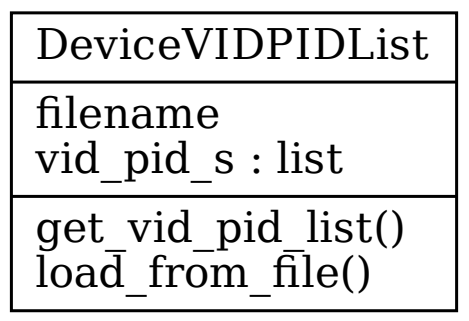
Per ogni porta scollegata la funzione `remove_devices()` si occupa di terminare il thread associato a tale porta. Inoltre la GUI viene notificata dell'evento.

```
if removed_ports:
    self.removed_devices =
        self.remove_devices(removed_ports)
    self.dev_removed_sig.emit()
```

Il thread del Device Manager esegue periodicamente con frequenza di 1 Hz.

```
sleep(1)
```

Implementazione - classe `DeviceVIDPIDList`



Per poter rilevare i nuovi dispositivi connessi di interesse il Device Manager necessita di conoscere il `VID (Vendor ID)` e `PID (Product ID)` associati alla porta USB della porta seriale utilizzata. Una classe `DeviceVIDPIDList` si occupa di caricare un elenco di ID desiderati da un file di configurazione. Un'istanza della classe `DeviceVIDPIDList` viene passata all'istanza della classe `DeviceManager` al momento della sua costruzione.

Implementazione - file di configurazione

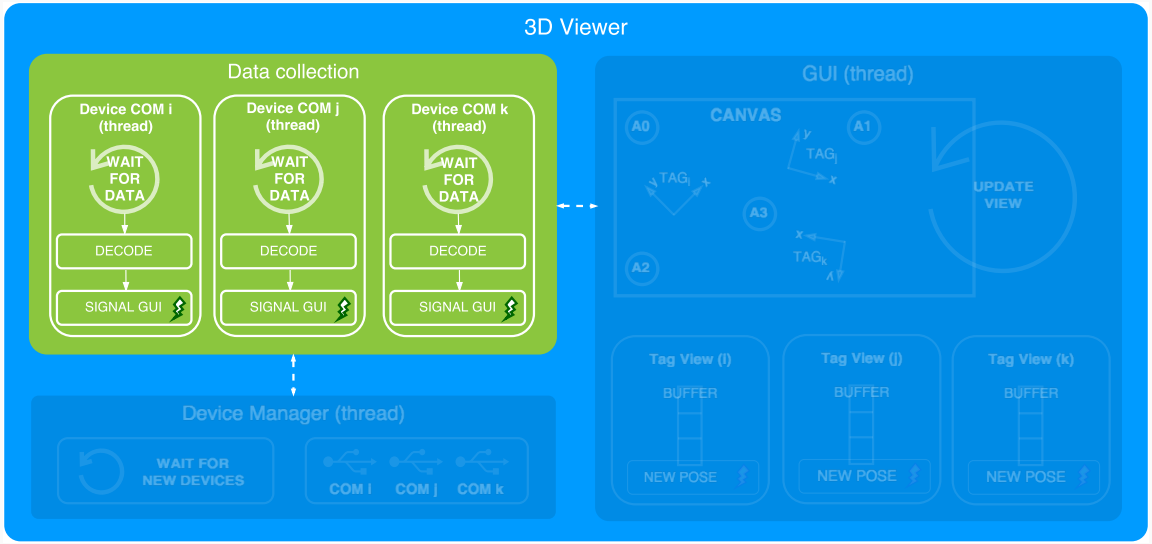
La struttura del file di configurazione è molto semplice ed è la seguente

```
CONFIG_VID_PID
VID PID
<vid 1> <pid 1>
...
<vid N> <pid N>
```

dove `<vid i>` e `<pid i>` sono dati ciascuno da una sequenza di 4 cifre. Ad esempio

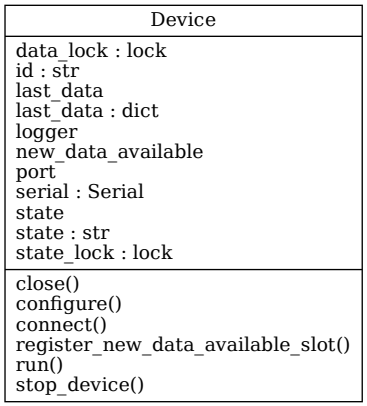
```
CONFIG_VID_PID
VID PID
0483 5740
```

Architettura - Device *i*-esimo



Un thread viene associato ad ogni dispositivo connesso. Esso si occupa di interrogare continuamente la porta seriale associata al dispositivo e di decodificare ogni nuova linea, terminata da `\r\n`, ricevuta.
Se la decodifica avviene con successo il thread, mediante un meccanismo di signal-slot, notifica alla GUI la presenza di nuovi dati.

Implementazione - classe `Device` - `run()`



Il metodo `run` è il metodo principale associato al thread di ciascun Device.
Dopo aver verificato l'avvenuta connessione con la porta seriale

```
def run(self):
    if not self.connect():
        return
```

il metodo entra nel loop principale dal quale esce solo nel caso in cui lo stato diventi `stopped` (i.e. quando il Device Manager termina il thread)

```
        while self.state == 'running':
```

Implementazione - classe `Device` - `run()`

Il metodo tenta quindi di leggere una linea terminata da `\r\n` e di decodificarla attraverso la classe `DataFromEVB1000`

```
try:
    line = self.serial.readline()
    if len(line) > 0:
        try:
            evb1000_data = DataFromEVB1000(line)
        except InvalidDataFromEVB1000:
            continue
```

Nel caso di decodifica avvenuta con successo i nuovi dati sono memorizzati nel campo `last_data` dell'istanza del `Device`, la GUI viene notificata della presenza di nuovi dati e il logger CSV memorizza su file i dati

```
if evb1000_data.msg_type_decoded:
    self.last_data = evb1000_data.decoded
    self.new_data_available.emit(self.id)
    self.logger.log_data(evb1000_data)
```

Implementazione - classe `Device` - `run()`

Nel caso di errore nella lettura da seriale i nuovi dati vengono scartati

```
except SerialException:
    pass
```

Qualora lo stato del `Device` diventi `stopped` il logger CSV viene chiuso e la porta seriale chiusa

```
if self.state == 'stopped':
    self.logger.close()
    self.close()
```

Implementazione - classe `DataFromEVB1000` - `decode_msg_type()`

DataFromEVB1000
decoded line msg_fields msg_fields : list msg_structure : list msg_type : str msg_type_decoded : bool
decode() decode_msg_type()

Il metodo fondamentale della classe `DataFromEVB1000` è `decode_msg_type` che definisce la struttura interna del messaggio ricevuto.

Il tipo del messaggio è ricavato dai primi tre caratteri della linea `line`

```
def decode_msg_type(self):
  if len(self.line) < 3:
      return False
  msg_type = self.line[0:3]
```

Implementazione - classe `DataFromEVB1000` - `decode_msg_type()`

Nel caso di un messaggio di tipo Tag Position Report il tipo è `tpr`,

```
if msg_type == 'tpr':
```

i campi del messaggio sono nominati `msg_type`, `tag_id` (i.e. l'id del tag come da switch sul PCB), `x`, `y` e `z`

```
self.msg_fields = ['msg_type', 'tag_id', 'x', 'y', 'z']
```

e i loro tipi sono indicati come `s` (stringa), `u` (unsigned int) e tre `f` (float)

```
self.msg_structure = ['s'] + ['u'] + ['f'] * 3
```

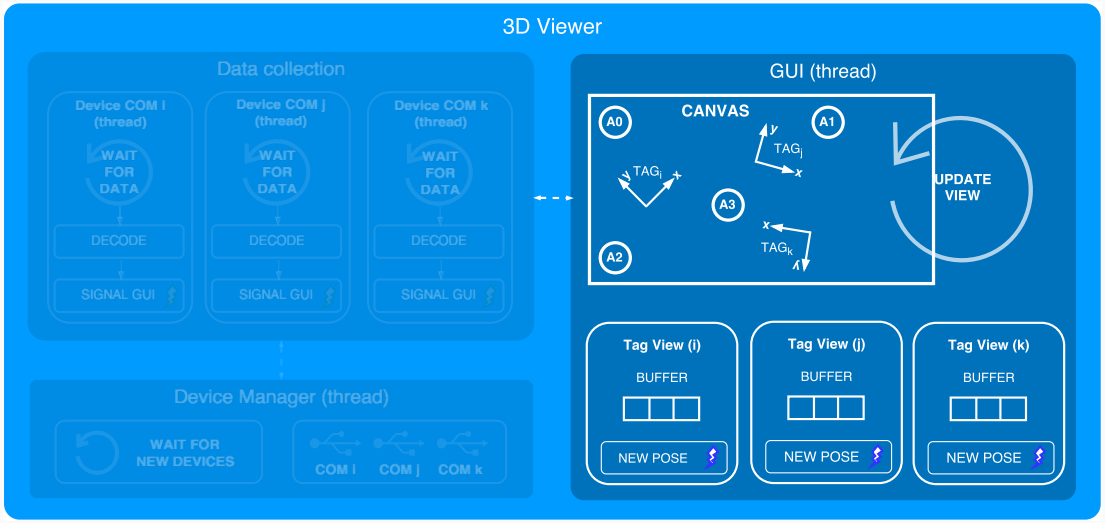
Altri tipi di messaggio sono indicati negli altri rami

```
elseif ...
```

mentre decodifica fallisce qualora il tipo di messaggio non sia tra quelli riconosciuti

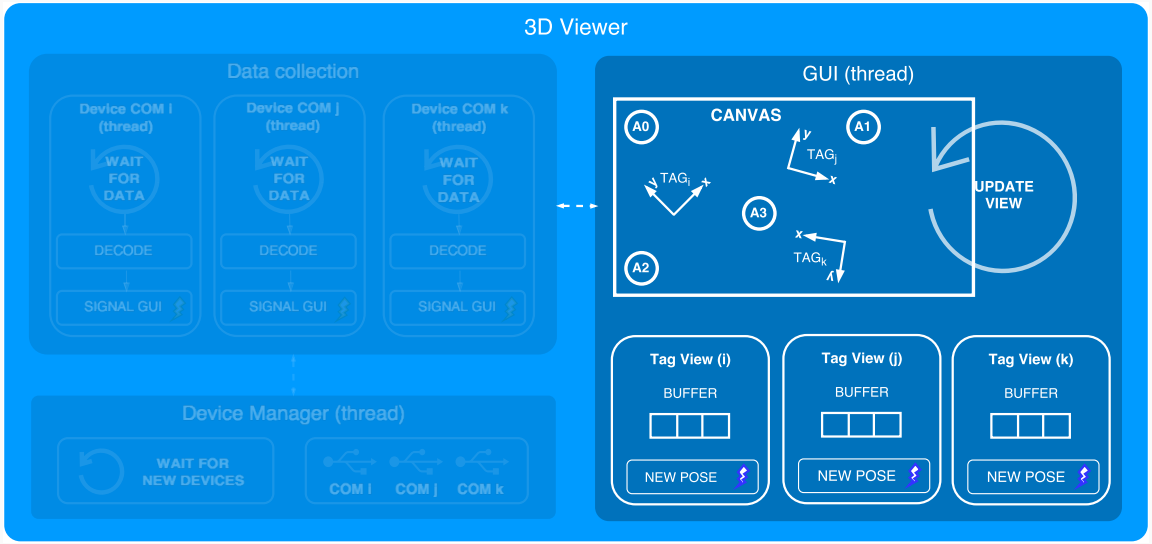
```
else:
    return False

return True
```

Architettura - GUI



Per la GUI è previsto un thread a parte che, mediante callback dette slot, riceve le notifiche provenienti da ciascun Device thread e dal Device Manager.
Quando una nuova posizione del tag è disponibile viene memorizzata all'interno di una struttura dati con buffer, detta Tag View. Una variante di Tag View, detta Tag Position Attitude View, è prevista per ospitare la stima di posizione e assetto.

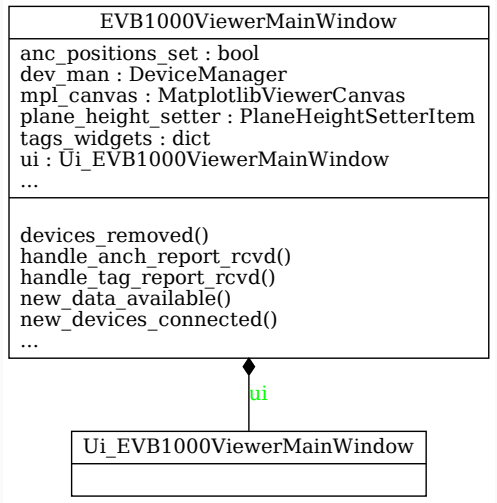
Architettura - GUI



Periodicamente la GUI aggiorna il Canvas principale, ospitato nella finestra, in cui vengono rappresentate la posizione del tag con uno scatter 3D e la stima con una terna.

Il Canvas prevede inoltre la rappresentazione delle ancore mediante dei cilindri. La posizione delle ancore viene trasmessa dal tag attraverso la porta seriale.

Implementazione - classe `EVB1000ViewerMainWindow`



La classe fondamentale che implementa la GUI è `EVB1000ViewerMainWindow`. Essa contiene un riferimento alla classe `Ui_EVB1000ViewerMainWindow` che contiene l'implementazione della finestra principale generata dal software Qt Creator.

Al fine di capire in che modo il Device Manager e il Device notificano alla GUI rispettivamente la presenza di nuovi dispositivi connessi e la presenza di nuovi dati è opportuno analizzare il costruttore della classe `EVB1000ViewerMainWindow` ed i metodi `new_devices_connected()` e `new_data_available()`.

Implementazione - classe `EVB1000ViewerMainWindow` — costruttore

Al costruttore della classe `EVB1000ViewerMainWindow` viene passata un'istanza `device_manager` del Device Manager

```
    def __init__(self, device_manager = None):
        ...
```

viene inizializzata/configurata la classe della finestra principale

```
        self.ui = Ui_EVB1000ViewerMainWindow()
        self.ui.setupUi(self)
        ...
```

viene creata un'istanza del Canvas che ospiterà la rappresentazione delle ancore e dei tag

```
        frame_rate = 25.0
        tag_positions_buffer_size = 10
        self.mpl_canvas =
            MatplotlibViewerCanvas(self.ui.matPlotGroupBox,\
                                                    frame_rate,\
                                                    tag_positions_buffer_size)
        ...
```

Implementazione - classe `EVB1000ViewerMainWindow` — costruttore

e viene salvato il riferimento al Device Manager

```
self.dev_man = device_manager
```

Molto importante è la seguente parte in cui la GUI registra due metodi della classe come `slot` associati a segnali emessi dal Device Manager.
In pratica il metodo `new_devices_connected` verrà richiamato quando il Device Manager rileva nuovi dispositivi connessi ed il metodo `devices_removed` sarà richiamato quando alcuni dispositivi sono stati scollegati.

```
if self.dev_man != None:
    self.dev_man.register_new_devices_connected_slot(\
                 self.new_devices_connected)
    self.dev_man.register_devices_removed_slot(\
                 self.devices_removed)
...
```

Impl.ne - classe `EVB1000ViewerMainWindow` — `new_devices_connected`

Lo slot `new_devices_connected` si occupa principalmente di richiedere al Device Manager la lista dei nuovi dispositivi connessi

```
@pyqtSlot()
def new_devices_connected(self):
    devs = self.dev_man.new_devices
```

e di registrare lo slot `new_data_avilable` in modo tale che sia richiamato quando uno qualsiasi dei Device notifica la presenza di nuovi dati.

```
    for dev in devs:
        dev.register_new_data_available_slot(\
            self.new_data_available)
    ...
```

Impl.ne - classe `EVB1000ViewerMainWindow` — `new_data_available`

Lo slot `new_data_available` viene chiamato con un argomento `device_id`

```
@pyqtSlot(str)
def new_data_available(self, device_id):
```

che consente di recuperare il dispositivo che ha emesso il segnale attraverso il metodo `device()` del Device Manager.

```
    try:
        device = self.dev_man.device(device_id)
```

Nel caso di un dispositivo appena disconnesso può accadere che l'ultimo messaggio ricevuto dalla porta seriale venga comunque elaborato dal thread associato al dispositivo poichè lo stato del thread (vedi slide ..) non è ancora cambiato da `running` a `stopped` quando il messaggio è arrivato.
In tal caso la notifica viene comunque inviata dal Device e la callback `new_data_available` viene chiamata ma ormai il `device_id` non identifica più un dispositivo valido. L'eccezione `KeyError` gestisce questa situazione.

```
    except KeyError:
        return
```

Impl.ne - classe `EVB1000ViewerMainWindow` — `new_data_available`

Una volta recuperato il Device da cui la notifica è originata è possibile accedere all'ultimo messaggio ricevuto contenuto in `device.last_data`.
L'accesso a questo campo è protetto con un semaforo di mutua esclusione.

```
data = device.last_data
```

In base al tipo di messaggio la GUI gestisce la situazione diversamente.

```
if data['msg_type'] == 'tpr' or data['msg_type'] == 'kmf':
    self.handle_tag_report_rcvd(device_id, data)
elif data['msg_type'] == 'apr':
    self.handle_anch_report_rcvd(data)
```

Impl.ne - classe `EVB1000ViewerMainWindow` — `handle_tag_report_rcvd`

Come esempio si consideri la funzione `handle_tag_report_rcvd` che gestisce l'arrivo di nuovi dati di posizione e sia `data['msg_type'] == 'tpr'`.
La funzione prende in ingresso i dati `data`.
Per prima cosa viene recuperato l'ID associato al tag (i.e. l'id del tag come impostato nello switch del PCB, tale id è diverso dal `device_id` che è invece associato alla specifica porta seriale a cui il tag è stato collegato).

```
def handle_tag_report_rcvd(self, ..., data):
    tag_id = data['tag_id']
```

Nel caso in cui il tag abbia già spedito al Viewer la posizione delle ancore (i.e. `anc_positions_set == True`) la funzione si interfaccia con la classe `MatplotlibViewerCanvas` di cui `mpl_canvas` è istanza.

```
    if self.anc_positions_set:
```

Come già detto è presente una struttura con buffer TagView utilizzata per memorizzare le ultime N posizioni ricevute. Nel caso in cui non sia presente un Tag View per il `tag_id` di interesse allora viene configurato per la prima volta il tag con la chiamata alla funzione `set_new_tag`

```
        if not self.mpl_canvas.is_tag_view(tag_id):
            self.mpl_canvas.set_new_tag(...)
        ...
```

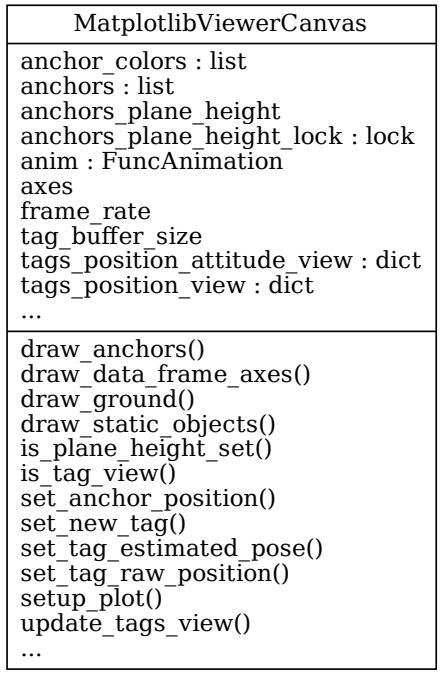
Impl.ne - classe `EVB1000ViewerMainWindow` — `handle_tag_report_rcvd`

In ogni caso i dati di posizione vengono estratti e passati alla classe `mpl_canvas`.

Ulteriori dettagli sulla classe `mpl_canvas` seguono nelle slide successive.

```
x = data['x']
y = data['y']
z = data['z']

if data['msg_type'] == 'tpr':
    self.mpl_canvas.set_tag_raw_position(tag_id,
        x, y, z)
elif ...

...
```

Impl.ne - classe `MatplotlibViewerCanvas`



La classe `MatplotlibViewerCanvas` si occupa di gestire il Canvas, inserito nella finestra principale, che ospita la rappresentazione del tag e delle ancore.

I metodi fondamentali che saranno analizzati sono `set_new_tag`, `set_tag_raw_position` e `update_tags_view`.

Impl.ne - classe `MatplotlibViewerCanvas - set_new_tag`

Il metodo `set_new_tag` prende come argomenti l'ID del tag `tag_ID` ed un colore `tag_color`.
Queste informazioni sono utilizzate per istanziare due oggetti.
Il TagPositionView utilizzato per conservare in un buffer di grandezza `tag_buffer_size` le posizioni risultato della trilaterazione eseguita internamente dal tag. Il buffer viene utilizzato per realizzare uno scatter plot 3D con una "scia".
Il TagPositionAttitudeView è invece pensato per conservare la stima di posizione e assetto correnti.

```
    def set_new_tag(self, tag_ID, tag_color):
        self.tags_position_view[tag_ID] =
             TagPositionView(self.axes,\
                             self.tag_buffer_size,\
                             tag_color)

        self.tags_position_attitude_view[tag_ID] =
             TagPositionAttitudeView(self.axes,\
                                     tag_color)
```

Impl.ne - classe `MatplotlibViewerCanvas - set_tag_raw_position`

Il metodo `set_tag_raw_position` prende come argomenti l'ID del tag `tag_ID` e le coordinate cartesiane `x`, `y` e `z`.

```
def set_tag_raw_position(self, tag_ID, x, y, z):
```

La posizione viene trasformata mediante trasformazione omogenea (`vector_hom_transformation`)per tenere conto del fatto che i dati sono espressi rispetto ad una terna posizionata all'altezza del piano comune alle ancore A0, A1 ed A2. Inoltre, a seconda di come le ancore sono state posizionate, l'asse z del sistema di riferimento, nel quale sono espresse le posizioni, potrebbe essere rivolto verso il basso. In tal caso la parte di rotazione della trasformazione omogenea sarà diversa dalla matrice identità.

```
position_data_frame = np.array([[x,y,z]]).T
position_mpl_frame =
    self.vector_hom_transformation(position_data_frame)
```

Impl.ne - classe `MatplotlibViewerCanvas - set_tag_raw_position`

Dopo essere stata trasformata la posizione viene effettivamente memorizzata nel Tag View relativo all'ID `tag_ID` attraverso la chiamata alla funzione `new_position()`

```
x = position_mpl_frame.item(0)
y = position_mpl_frame.item(1)
z = position_mpl_frame.item(2)
self.tags_position_view[tag_ID].new_position(x, y,
    z)
```

Le funzioni descritte fino a questo punto spiegano, anche se in maniera sintetica, il flusso dei dati a partire dalla porta seriale per arrivare alle strutture dati che contengono i dati trasformati. Diversi aspetti sono stati trascurati tra cui, ad esempio, la gestione vera e propria degli elementi grafici attraverso la libreria Matplotlib. Si rimanda al codice per maggiori informazioni.
Per concludere il percorso è necessario descrivere la funzione che si occupa di aggiornare periodicamente il Canvas utilizzando i dati disponibili.

Impl.ne - classe `MatplotlibViewerCanvas - update_tags_view`

La funzione `update_tags_view` viene richiamata periodicamente ad una frequenza `frame_rate` che viene configurata nel metodo `setup_plot`

```
def setup_plot(self, figure):
    ...
    time_step = 1.0 / self.frame_rate * 1000
    self.anim = animation.FuncAnimation(figure,
        self.update_tags_view, interval = time_step)
    ...
```

La funzione `update_tags_view` si occupa di richiamare il metodo `update_view` per ogni TagView e per ogni TagPositionAttitudeView configurato. L'aggiornamento delle View nel tempo di fatto realizza l'animazione così come visibile all'utente.

```
def update_tags_view(self, frame_number):
    for view_name in self.tags_position_view:
        self.tags_position_view[view_name].update_view()
    for view_name in self.tags_position_attitude_view:
        self.tags_position_attitude_view[view_name].update_view()
```