POLITECNICO DI MILANO

FACOLTÀ DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea in Ingegneria Informatica

REACTIVE PROGRAMMING PER SISTEMI A RISORSE LIMITATE

Relatore: Prof. LUCA MOTTOLA

Tesi di laurea di:
NICOLA CASAMASSIMA
Matr. 819658

Anno Accademico 2014 - 2015

Abstract

L'affermazione di applicazioni sempre più legate ad aggiornamenti in tempo reale e a feedback istantanei agli input provenienti dall'utente, e la diffusione di sistemi anche per uso quotidiano che necessitano di comunicare con l'ambiente e svolgere operazioni con reattività, hanno introdotto per i programmatori, delle necessità che in passato non sono mai state di significativo interesse.
Infatti la capacità di reagire immediatamente al verificarsi di un evento non sempre è dal punto di vista della programmazione, di facile implementazione, o almeno utilizzando le tecniche di stesura del codice più comuni.
Per questo è necessaria la nascita di nuovi schemi che meglio si adattano a queste situazioni.

É qui che interviene il paradigma di programmazione Reactive Programming. Questo paradigma, si adatta alla perfezione a quei sistemi o applicazioni con caratteristiche ben precise, che sono Event-driven e Reattivi.
Di sistemi con le caratteristiche sopra citate ve ne sono di ogni tipologia, ma alcuni in particolare possono distinguersi per la loro peculiarità di avere forti limitazioni hardware, non sempre supportati adeguatamente a fronte di una loro imponente diffusione. Si tratta di alcune famiglie di sistemi embedded con risorse limitate.
Infatti la reattività è una caratteristica ricercata in sistemi mainstream, come ad esempio le GUI che devono rispondere alle interazioni con gli utenti, ma è anche una caratteristica che può essere richiesta nel mondo dei sistemi embedded. Tra questi troviamo ad esempio attuatori o sensori, che devono reagire agli input provenienti dall'ambiente. Nel caso di un sensore di temperatura ad esempio, a seguito di una variazione di temperatura dell'ambiente, questo deve aggiornare immediatamente il valore di temperatura letto precedentemente e generare una variazione di segnale visibile da altri componenti. Sistemi di questo tipo si prestano all'utilizzo di reactive programming per implementare il software che li supporta.

Il presente lavoro di tesi si propone di creare una libreria reactive programming che, offra un supporto ai programmatori di sistemi embedded a risorse limitate. Visto che il target scelto per la libreria ha forti limitazioni hardware, tra cui la più

importante riguarda la esigua quantità di memoria disponibile, le caratteristiche della libreria sono improntate alla risoluzione del problema di salvaguardare il più possibile il consumo di memoria.
Dopo aver analizzato lo stato dell'arte, la tesi si articola nell'analisi degli elementi necessari per supportare reactive programming e delle caratteristiche di un sistema embedded a risorse limitate, per poter quindi procedere alla definizione dei criteri di progetto. Una volta effettuata l'analisi, si passa alla progettazione, sviluppo e testing della nuova libreria reactive programming. Dopo aver fatto le valutazioni sulla nuova libreria e sui miglioramenti nel consumo delle risorse rispetto alle librerie già esistenti, vi è infine l'applicazione della stessa ad un caso reale.

Elenco delle figure

Elenco dei listati

Indice

Capitolo 1

Introduzione

L'affermazione di applicazioni sempre più legate ad aggiornamenti in tempo reale e a feedback istantanei agli input provenienti dall'utente, e la diffusione di sistemi anche per uso quotidiano che necessitano di comunicare con l'ambiente e svolgere operazioni con reattività, hanno introdotto per i programmatori, delle necessità che in passato non sono mai state di significativo interesse.

Infatti uno dei comuni obbiettivi che si cerca di raggiungere oggi quando si implementa un'applicazione, è la massima reattività raggiungibile. Con solo questa si migliora notevolmente il valore del lavoro svolto, dato che per l'utente finale è importante l'usabilità del prodotto, caratteristica questa strettamente legata alla reattività.

La capacità di reagire immediatamente al verificarsi di un evento non sempre è dal punto di vista della programmazione, di facile implementazione, o almeno se si utilizzano le tecniche di stesura del codice più comuni.
Ci si è abituati infatti nel tempo ad avere a che fare con una pratica implementativa che rispecchiasse il modello sequenziale del paradigma imperativo. Ma non tutti i sistemi rientrano facilmente in questo modello implementativo. Infatti nei casi sopra descritti, la varietà e la discontinuità dei dati in input, costringono all'utilizzo di nuovi schemi che meglio si adattano a queste situazioni.

É qui che interviene il paradigma di programmazione reactive programming. Questo paradigma, di più recente introduzione rispetto ai classici, si adatta alla perfezione a quei sistemi o applicazioni con caratteristiche ben precise, che sono Event-driven e Reattivi.
Di sistemi con le caratteristiche sopra citate ve ne sono di ogni tipologia, ma alcuni in particolare possono distinguersi per la loro peculiarità di avere forti limitazioni hardware, non sempre supportati adeguatamente a fronte di una loro imponente diffusione. Si tratta di alcune famiglie di sistemi embedded con risorse limitate.

Infatti la reattività è una caratteristica ricercata in sistemi mainstream, come ad esempio le GUI che devono rispondere alle interazioni con gli utenti, ma è anche una caratteristica che può essere richiesta nel mondo dei sistemi embedded. Tra questi troviamo ad esempio attuatori o sensori, che devono reagire agli input provenienti dall'ambiente. Nel caso di un sensore di temperatura ad esempio, a seguito di una variazione di temperatura dell'ambiente, questo deve aggiornare immediatamente il valore di temperatura letto precedentemente e generare una variazione di segnale visibile da altri componenti. Sistemi di questo tipo si prestano all'utilizzo di reactive programming per implementare il software che li supporta.
Reactive programming applicata ai sistemi embedded a risorse limitate, risulta tutt'oggi argomento di rara trattazione, a causa anche della mancanza di librerie che supportino il paradigma reattivo applicato a questi sistemi.

1.1 Contributo

Il presente lavoro di tesi si propone di creare una libreria reactive programming che, partendo da un'analisi dettagliata delle caratteristiche che questa deve avere, offra un supporto ai programmatori di sistemi embedded a risorse limitate per l'utilizzo di questo paradigma.
Visto che il target scelto per la libreria ha forti limitazioni hardware, tra cui la più importante riguarda la esigua quantità di memoria disponibile, le caratteristiche della libreria sono improntate alla risoluzione del problema di salvaguardare il più possibile il consumo di memoria.
I requisiti del mio lavoro quindi, devono tenere conto di queste limitazioni, caratterizzando di conseguenza la libreria che dovrà:

- Occupare meno memoria possibile.
- Mantenere un tempo di esecuzione delle azioni implementate dalla libreria, che sia il più basso possibile.
- Essere compatibile con le architetture dei microcontrollori presenti nei sistemi embedded.
- In ottica di compatibilità, una buona importanza la assume la scelta del linguaggio, scegliendone quindi uno di ampia diffusione e supportato dalla maggior parte dei sistemi.

Il progetto si articola nei seguenti punti:

- Analisi degli elementi che compongono una libreria di supporto a reactive programming, prendendo come esempio alcune librerie già realizzate per altri sistemi senza limitazioni di memoria.

- Analisi delle caratteristiche sia hardware che software di un sistema embedded a risorse limitate, per poter procedere alla definizione di criteri di progetto che caratterizzeranno la nuova libreria.
- Progettazione, sviluppo e testing della nuova libreria reactive programming.
- Applicazione della libreria creando con essa del codice reactive programming, applicato poi ad un caso reale.

1.2 Organizzazione

Il presente elaborato di tesi, saltando la ridondante descrizione di questo capitolo, è così organizzato:

- Il capitolo 2 riguarda l'analisi dello stato dell'arte, descrivendo esplicitamente il paradigma reactive programming e indicandone casi d'uso e librerie già esistenti che lo supportano.
- Il capitolo 3 è dedicato alla libreria realizzata, con descritti i criteri di progetto, le sue funzionalità.
- Il capitolo 4 tratta le scelte implementative fatte per affrontare problemi e criticità legate all'implementazione di un comportamento reattivo event-based su un sistema embedded a risorse limitate, e alla descrizione delle API fornite al programmatore.
- Nel capitolo 5 si documentano le valutazioni sulle prestazioni effettuate, descrivendo come sono state espletate, che risultati hanno restituito e che significato si attribuisce ai risultati ottenuti. Le valutazioni effettuate sono scelte in maniera tale da rendere evidente la possibilità di applicare o meno il progetto ad un sistema embedded con risorse limitate. Inoltre, sono stati eseguiti confronti con ciò che già esisteva per constatare la bontà del lavoro effettuato, in caso di presenza di effettivi miglioramenti prestazionali.
- Il capitolo 6 presenta un'applicazione del lavoro svolto ad un caso reale, mostrando poi il risultato di alcuni test effettuati sul sistema con e senza reactive programming a confronto. L'applicazione della libreria è stata fatta su un nano-drone con una quantità di memoria molto limitata.
- Infine nel capitolo 7 sono riassunti i risultati raggiunti attraverso questo lavoro e definiti possibili sviluppi futuri.

I risultati presentati in questo elaborato riguardano le valutazioni effettuate nel capitolo 5 e l'applicazione ad un caso reale del capitolo 6.
Per quanto riguarda le valutazioni tra i risultati ottenuti sulla nuova libreria è

evidente, confrontandoli con risultati ottenuti da valutazioni su un'altra libreria presa come esempio, un notevole risparmio nei consumi di memoria accompagnato da un miglioramento nei tempi di risposta. Quest'ultima non è un risultato scontato visto che spesso, la riduzione di memoria occupata, comporta un peggioramento nei tempi di risposta.
Per quanto riguarda l'applicazione ad un caso reale, i risultati evidenziano dopo l'applicazione della nuova libreria, un leggero miglioramento della precisione del drone, nel mantenere una posizione stazionaria in volo.

Capitolo 2

Stato dell'arte

Per meglio presentare questo progetto di tesi, si vuole esporre un'analisi preliminare effettuata prima di cominciare il lavoro. In questa vengono presentate le tecnologie esistenti ad oggi, riguardanti reactive programming e suoi sviluppi, comprese anche le librerie di supporto a questo tipo di programmazione.

2.1 Concetti

Per rispondere all'esigenza di avere applicazioni sempre più reattive, che reagiscono agli eventi provenienti dall'esterno o dall'applicazione stessa, è stata introdotta una tecnica di programmazione moderna, denominata reactive programming. Questa viene proposta come soluzione atta alla semplificazione della costruzione di applicazioni event-driven che hanno continue interazioni con il loro ambiente, e che processano eventi eseguendo i task corrispondenti come ad esempio aggiornare il proprio stato, cambiare il valore a delle variabili, invocare funzioni e altro.

Utilizzando i paradigmi tradizionali, costruire un'applicazione reattiva voleva dire utilizzare soluzioni non sempre di facile gestione, come ad esempio le callback asincrone. Questo causava la creazione di programmi di difficile gestione, pieno di frammenti isolati di codice che accedono alle stesse variabili e il cui ordine di esecuzione non è spesso predicibile. Con reactive programming si vuole ovviare a queste difficoltà, proponendo soluzioni che meglio si adattano a questa tipologia di applicativi. In particolare le applicazioni reattive sono:

- Event-driven: reagiscono agli eventi.
- Mantenibili: perché il codice è di facile costruzione e lettura.
- Deterministiche: a seguito di un input hanno una reazione prevedibile, che sarà sempre la stessa a fronte dello stesso input.

Reactive programming, è un paradigma di programmazione orientato verso i flussi asincroni di dati e di propagazione automatica dei loro cambiamenti nel tempo.

Con propagazione si intende la diffusione di cambiamenti avvenuti ad un dato, verso oggetti ad esso collegati attraverso relazioni logico-matematiche, che si aggiorneranno con il nuovo valore ricevuto e propagheranno a loro volta fino a che incontreranno altri oggetti a cui propagare il valore. Per esempio se ho un assegnamento "a = b + c", mentre in altri paradigmi al variare di 'b' o 'c' avvenuto dopo l'esecuzione del dato assegnamento, 'a' non vede questo cambiamento, nella reactive programming a viene riaggiornata automaticamente.

Se con il paradigma imperativo ho ad esempio

```
a = 5, b = 2;
c = a + b;
a = 2;
//con c che in questo caso è 7
```

In presenza di reactive programming 'c' sarebbe aggiornato in automatico al variare di 'a', avrei quindi:

```
a = 5, b = 2;
c = a + b;
a = 2;
//con c che in questo caso è 4
```

Altro esempio può essere la presenza di un ciclo con variabili lette ad ogni iterazione da una qualsiasi sorgente esterna, se queste variabili vengono utilizzate per degli assegnamenti, questi ultimi vengono eseguiti anche se il valore delle variabili non è cambiato rispetto all'iterazione precedente del ciclo. Con reactive programming questo non si verificherebbe con i vantaggi del caso perché vengono propagati solamente i cambiamenti. Per capirne i vantaggi, basta pensare di dover eseguire operazioni matematiche onerose al posto di normali assegnamenti per cogliere la possibilità di guadagno prestazionale utilizzando il paradigma in questione.

Reactive programming presenta quindi una struttura del codice che la rende di più facile stesura e lettura, grazie alla possibilità di creare facilmente propagazioni di dati. Con ciò sono facilmente realizzabili flussi automatici di aggiornamento delle variabili, che partono a seguito di un trigger, che di norma è il verificarsi di un evento, come ad esempio la variazione di un valore di una data variabile.

Vi sono svariati tipi di paradigmi di programmazione, si può fare quindi un confronto tra reactive programming ed alcuni di essi:

- Nel **paradigma imperativo** un programma viene inteso come un insieme di istruzioni, ciascuna delle quali può essere pensata come un "ordine" che viene impartito alla macchina. Molto simile è il paradigma procedurale, che a differenza del precedente prevede il raggruppamento di porzioni di programma in sezioni ben delimitate, richiamabili in altre parti del codice,

dette procedure. Nei paradigmi imperativo e procedurale ogni qual volta si deve aggiornare il valore di una variabile, bisogna esplicitamente codificare un assegnamento. Questo implica, come detto in precedenza, l'utilizzo di variabili in modo esplicito. Al variare di un valore quindi, è d'obbligo aggiornare nel caso mi servano successivamente (richiamando un assegnamento), eventuali variabili dipendenti da questo valore.

- Il **paradigma funzionale** è un paradigma di programmazione in cui il flusso di esecuzione del programma assume la forma di una serie di valutazioni di funzioni matematiche. Una caratteristica è quella della mancanza di side-effect delle funzioni, il che comporta una più facile verifica della correttezza del programma e la possibilità di una maggiore ottimizzazione dello stesso. Un uso particolare del paradigma, per l'ottimizzazione dei programmi, è quello di trasformare gli stessi per utilizzarli nella programmazione parallela. Nel paradigma funzionale si possono eseguire serie di funzioni matematiche e passaggi iterativi di parametri restituiti tra una funzione e un'altra. Questo comincia ad assomigliare ad un comportamento tipico di reactive programming, ma solo per ciò che riguarda la creazione di un flusso di dati e di chiamate a funzioni. Anche qui in caso di cambiamenti di valori, il processo di diffusione del cambiamento non è automatizzato e immediato, bisogna infatti pensare ad una struttura di chiamate a funzioni in grado di garantire l'utilizzo di variabili sempre aggiornate.

In reactive programming invece, in confronto ai 2 paradigmi appena descritti, si garantisce, con una struttura di codice molto più semplice, la creazione di flussi di dati attivati automaticamente al variare di un valore. Tali flussi aggiornano tutte le variabili implicate in relazioni con altre variabili i cui valori sono stati aggiornati.

Reactive programming è basato sul paradigma del dataflow sincrono di Lee and Messerschmitt, ma con vincoli sui tempi meno restrittivi. Il paradigma è caratterizzato dalla presenza di due nozioni chiave, intorno alle quali ruota tutta la logica di costruzione del codice: il behavior e l'evento [9].

2.1.1 Behavior

Questo tipo di oggetto varia in modo continuo nel tempo. Ci possono essere anche behavior costanti, come ad esempio numeri o colori, mentre il più semplice behavior variabile nel tempo, è il tempo stesso. Di norma assume valori reali, quindi rappresentati da variabili di tipo float. L'idea di base è quella che un valore tempo-variabile sia rappresentato in funzione del tempo.

```
1 object Behavior a =
2     Behavior {
```

```
3       Now.Time -> a
4     }
```

Come in questo esempio in pseudocodice, in cui il tipo behavior è caratterizzato da una variabile che assume il valore dell'istante in cui viene utilizzata. La natura continua nel tempo del behavior implica che più il clock in cui gira il programma reattivo aumenta, e più il valore che esso contiene è rappresentativo della realtà.

2.1.2 Evento

Rappresenta un avvenimento (o una sequenza di essi) che si può verificare nel tempo. Possono riferirsi a degli stream di cambiamenti di valore. Possono essere anche creati dal programmatore, chiamando ad esempio una determinata funzione.

A differenza dei behaviors, gli eventi occorrono in istanti discreti nel tempo. Un evento può essere rappresentato da un valore in un determinato istante di tempo, quindi una coppia tempo-valore, come ad esempio avviene a seguito di un input da tastiera: se si preme il tasto 'a' nell'istante t, l'evento "key pressed" avrà come valore (t, 'a'). Gli eventi si possono verificare anche a seguito di una condizione booleana risultata vera. In questo caso si parla di eventi predicato.

Behaviors ed eventi possono essere combinati con altri dello stesso tipo, per formarne di diversi e più complessi. Il cambiamento di valore di un behavior può essere strettamente legato ad un evento:

```
1   > color :: Behavior Color
2   > color = red `until` (lbp -=> blue)
```

In un programma si hanno solitamente combinazioni ricorsive di behaviors ed eventi, con continui cambiamenti di valori. Questo genera degli serie di infiniti campioni nel tempo, definiti stream, con i behaviors che sono visti come degli stream trasformer, dato che permettono i cambiamenti di valore.

2.2 Caratteristiche

Come si è visto reactive programming fornisce un modo elegante di esprimere computazioni in funzioni di eventi, applicabili quindi in domini che richiedono reattività come quelli di animazioni interattive, robotica, computer vision, interfacce utente. Vediamo ora le principali caratteristiche di questo paradigma per comprenderne a fondo il suo funzionamento, e la struttura che attribuisce al codice.

Di seguito vengono descritte 6 proprietà di reactive programming che costituiscono la tassonomia. Queste proprietà sono basic abstractions, evaluation model, lifting, multidirectionality, glitch avoidance, e support for distribution [14].

2.2.1 Basic Abstractions

Per comprendere questa proprietà si può esprimere un'analogia con la programmazione imperativa. Infatti come nella programmazione imperativa sono definiti basic abstractions gli operatori primitivi (come assegnamenti) e i valori, così in reactive programming parliamo di questa proprietà quando ci riferiamo a primitive reattive in grado di facilitare la scrittura dei programmi. Queste primitive variano in funzione della tipologia di linguaggio reattivo che si sta utilizzando, anche se molte tipologie offrono behaviors ed eventi descritti precedentemente.

2.2.2 Evaluation Model

Questa proprietà riguarda la modalità con cui i cambiamenti nei valori sono propagati tra i behavior. Dal punto di vista del programmatore le propagazioni avvengono in modo trasparente, però al livello di linguaggio viene fatta una differenziazione basata su chi dà inizio alla propagazione dei cambiamenti. Infatti le possibilità sono 2: o la sorgente fa una "push" del nuovo dato ai suoi dipendenti (consumatori) oppure i dipendenti fanno una richiesta "pull" alla sorgente (produttore). In entrambi i casi il flusso di dati va sempre dal produttore al consumatore. Esistono perciò 2 tipi di evaluation model: push-based e pull-based, approfonditi successivamente.

2.2.3 Glitch Avoidance

Glitch avoidance è una proprietà basata sul concetto di glitch [10], che è una inconsistenza di aggiornamento che si può presentare durante la propagazione. Quando ad esempio la computazione di un dato avviene prima che tutte le sue dipendenze siano state aggiornate, ci si ritrova con un dato creato da un nuovo valore (quello che ha dato il via alla computazione) combinato a valori vecchi che ancora non sono stati aggiornati. Questo può avvenire soltanto in linguaggi push-based. Per comprendere meglio si consideri questo esempio:

```
1 var1 = 1
2 var2 = var1 * 1
3 var3 = var1 + var2
```

In questo esempio il valore di `var2` è sempre uguale a quello di var1, quello di `var3` quindi è il doppio di `var2`.

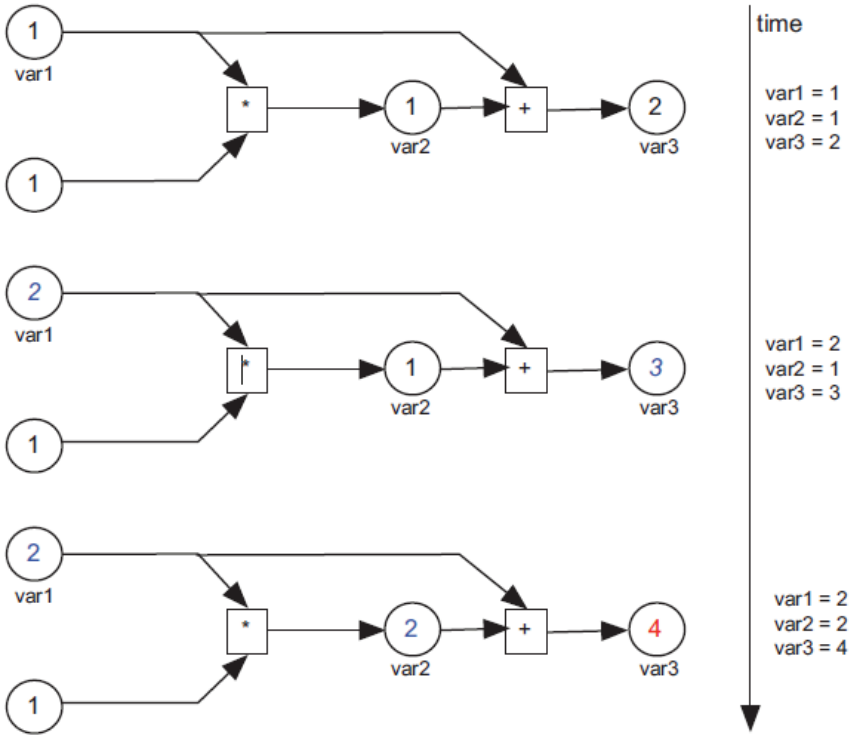

Figura 2.1: Esempio di un glitch.

In un paradigma imperativo al variare di `var1` ci si aspetta un aggiornamento di `var2` e successivamente che `var3` contenga il doppio del nuovo valore, ma in reactive programming non è così. Infatti quando `var1` viene aggiornato, sia `var2` che `var3` ricevono il nuovo valore di `var1`. In questo caso però `var3` vedrà ancora il vecchio valore di `var2` cambiando di fatto il risultato atteso. Molti linguaggi reactive programming eliminano i glithces organizzando le espressioni in un grafo topologicamente ordinato, ma rimane comunque complicato gestire queste situazioni in presenza di sistemi distribuiti, perché problemi di rete, ritardi o global clock non sincronizzato, possono causare in ogni caso ritardi negli aggiornamenti, che andrebbero così persi.

2.2.4 Lifting Operations

Quando reactive programming è supportata all'interno di un linguaggio, inclusa ad esempio in una libreria, gli operatori e le funzioni definite dall'utente di tale linguaggio devono essere convertiti per operare con i behaviors. Questa conversione è denominata Lifting. In un linguaggio tipicizzato staticamente (come java), una funzione non può essere direttamente applicata ad un behavior, perché questo viene tipicizzato dinamicamente. Il programmatore deve quindi esplicita-

mente eseguire un lift dei metodi o riscriverli, per assicurare la compatibilità con i behaviors. In linguaggi tipicizzati dinamicamente, si possono passare i behaviors come argomenti alle funzioni, senza dover eseguire lift o riscriverle. Questo è permesso ad esempio combinando reactive programming con paradigma funzionale come mostrato con le Lambda Expression nella sezione che parla di C++ del capitolo 3. I lifting sono classificati:

- Implicito: con questo approccio quando un operatore è applicato ad un behavior, questo automaticamente viene "liftato". Questo rende l'operazione di lifting trasparente al programmatore.
- Esplicito: in questo caso il linguaggio fornisce un set di combinatori utilizzabili dal programmatore per trasformare gli operatori rendendoli compatibili con i behaviors.
- Manuale: con questo approccio il linguaggio non fornisce operatori di lifting. Il programmatore deve ottenere manualmente il valore corrente dai behaviors, che potrà essere poi utilizzato con gli operatori ordinari.

2.2.5 Multidirectionality

Un'altra proprietà di reactive programming definisce se le propagazioni e i passaggi di valori avvengono in una sola direzione, cioè solo da un ipotetico behavior A ad un altro B oppure possono avvenire anche in quella opposta, da B ad A. Con la multidirezionalità i valori sono rimandati indietro aggiornati dopo essere stati propagati mentre se sono monodirezionali vengono propagati solo in un verso.

2.2.6 Support for Distribution

Questa proprietà riguarda la possibilità di supportare la scrittura da parte del linguaggio, in programmi reattivi distribuiti. Il supporto per la distribuzione permette di creare dipendenze tra dati che sono distribuiti su più nodi del sistema distribuito. Il bisogno di avere questo supporto è dovuto al fatto che le applicazioni reattive stanno diventando sempre più distribuite. D'altra parte implementando un programma destinato ad risiedere su più componenti di un sistema distribuito, risulta più difficile il controllo della consistenza dei valori di tutti i behaviors, a causa delle problematiche tipiche dei sistemi distribuiti stessi come ad esempio latenza e failure di rete.

2.3 Applicazioni

Reactive programming in generale può avere molteplici applicazioni che vanno dalle interfacce grafiche utente, ai sensori di sistemi integrati.

2.3.1 GUIs

Una delle applicazioni più immediate è, come detto precedentemente, l'utilizzo per realizzare GUI reattive [7]. Basti pensare alle caratteristiche che tendenzialmente una normale GUI ha: deve fornire rapidi accessi alle funzionalità del programma tramite oggetti grafici, deve essere responsive e deve reagire in maniera rapida ad una interazione da parte di un utente. Inoltre le GUI sono di difficile programmazione utilizzando un approccio sequenziale convenzionale, perché è impossibile prevedere che tipo di interazione esterna arriverà e quando questa si verificherà. Infatti la GUI può risultare inattiva per un tempo indefinito, in attesa che una qualsiasi azione la risvegli e la faccia computare.

Normalmente le azioni possibili in una GUI sono click con puntatore su bottoni o link, o generalizzando oggetti grafici con una forma definita, visibili all'utente.

Quando ciò avviene, si verifica un evento, ed è quindi facile pensare alla possibile applicazione del paradigma in questione. Si possono collegare a questi eventi behaviors e funzioni, permettendo quindi di eseguire computazioni in grado di restituire all'utente reazioni attese da parte dell'interfaccia.

2.3.2 Robotica

In sistemi utilizzati per la robotica, la varietà e la discontinuità dei dati di ingresso, costringe l'introduzione di reactive programming dato che ad ogni istante considerato, possono esserci nuovi dati provenienti dall'ambiente, alcuni dei quali possono influenzarne o dipendere da altri. Questo fa sì che questo tipo di sistemi ricadano tutti nella categoria dei sistemi reattivi. L'applicazione di reactive programming ai sistemi per la robotica è avvenuta quando ci si è resi conto che programmare il controllo dei robot utilizzando le informazioni provenienti dai sensori, per produrre azioni accettabili sugli attuatori, non poteva seguire gli schemi della programmazione standard normalmente utilizzata nelle applicazioni.
La presenza dei sensori stessi può rendere vantaggioso l'utilizzo di reactive programming. Basti pensare ad un sensore di temperatura, che quando rileva un cambiamento nell'input proveniente dall'ambiente, deve aggiornare in modo reattivo il proprio sistema per generare un segnale che sintetizzi un nuovo valore della temperatura rilevata.
Inoltre in un futuro prossimo reti di robot cooperanti, possono essere impiegate per elaborate missioni in ambienti sconosciuti. Questi robot possono ad esempio essere una rete distribuita di satelliti oppure una squadra di veicoli esploratori di Marte. Queste missioni richiedono un crescente livello di autonomia da parte dei robot, che devono rispondere alle incertezze dell'ambiente che andranno ad esplorare. La programmazione di queste reti di robot cooperanti sarà agevolata dal paradigma reactive programming, dato che può garantire reazioni ad eventi

e dato che è spesso necessario dedicare del tempo alla fase di ragionamento del robot che non è deterministica, ma regolata da eventi [16].

2.3.3 Mobile wireless networks

Le applicazioni per reti di device wireless non possono essere strutturate come programmi monolitici che accettano input fissi, dato che queste reti subiscono rapide variazioni con nuove connessioni e perdite di altre. Per permettere di rispondere in modo corretto a questi cambiamenti, vengono adottate architetture event-driven, che devono garantire ai produttori/consumatori di eventi, di poter connettersi e lasciare liberamente la rete. Per permettere ciò l'application layer deve essere in grado di reagire agli eventi rilevati dal publish/subscribe layer. In questo contesto torna quindi utile l'utilizzo di reactive programming per realizzare l'application layer di questa architettura [13].

2.3.4 Sistemi Embedded e Time Sensitive

Un'altra possibile applicazione della reactive programming può essere nell'ambito dei sistemi embedded e time sensitive.

Sistemi Embedded

I sistemi embedded sono tutti quei sistemi elettronici con microprocessore progettati solitamente per una determinata applicazione, spesso con una piattaforma hardware ad hoc, integrati nel sistema che controllano ed in grado di gestirne le funzionalità.

In questa area si collocano sistemi di svariate tipologie e dimensioni, in base al tipo di microprocessore, al sistema operativo, ed alla complessità del software che può variare da poche centinaia di byte a parecchi megabyte di codice. Appartengono a questa categoria i microcontrollori. Un sistema di questo tipo, ha delle finalità già note in fase di sviluppo. Grazie a ciò l'hardware può essere ridotto ai minimi termini limitando così anche i consumi, i tempi di elaborazione (maggiore efficienza) ed il costo di fabbricazione.

Spesso, gran parte dell'hardware di un sistema embedded deve sottostare a requisiti di prestazioni prestabiliti senza richiesta di scalabilità (essendo costruito su misura per scopi noti), a differenza di altri sistemi costruiti senza una finalità definita. Questo permette all'architettura del sistema embedded di essere semplificata rispetto a quella di altri sistemi, usando ad esempio una CPU più economica, minor quantità di memoria RAM, e uno spazio di archiviazione sufficiente appena per installare il proprio sistema operativo. Alcune tipologie di sistemi embedded sono utilizzati per scopi che li rendono time sensitive, come ad

esempio quei sistemi il cui funzionamento è regolato da un sistema operativo real time.

Sistemi Time Sensitive

I sistemi time sensitive sono sistemi in cui la correttezza non dipende solamente dai risultati ma anche dal tempo in cui questi sono proposti. Spesso questi sistemi sono dotati di sistemi operativi real time. Il termine real time infatti significa che il sistema funziona in modo sincronizzato con il tempo dell'ambiente in cui agisce.

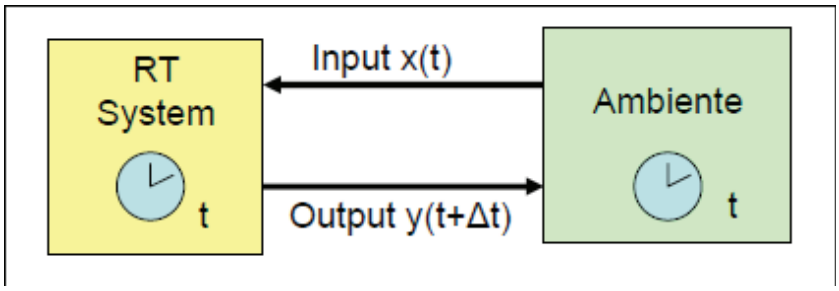

Figura 2.2: Scambio di informazioni tra sistema e ambiente.

Come si vede in figura 2.2 i sistemi real time si sincronizzano e comunicano con l'ambiente in cui sono inseriti. Gli approcci di tali sistemi sono conosciuti in campi come la robotica, sistemi di monitoraggio, sistemi con controllori e altri sistemi embedded in generale. Questi sistemi per sincronizzarsi con il tempo reale dell'ambiente, fanno uso di timer dedicati, che sono periferiche dipendenti dai cicli di clock.

Molta importanza la assumono i concetti di task e lo scheduling con il quale questi vengono eseguiti.

Come in altri sistemi, vi sono diverse tipologie di scheduling, in base alle quali varia l'ordine di esecuzione dei task. Quello che conta maggiormente però, è eseguire tutte le operazioni in un determinato intervallo di tempo, dato che dovendo il sistema rispondere in tempo reale agli input provenienti dall'ambiente, non si può protrarre a lungo la computazione. In caso contrario si avrebbe la non sincronia con il tempo dell'ambiente circostante.

Un esempio di un sistema real time può essere un agente che si deve muovere in un ambiente, che a seguito di valori in input dall'ambiente letti all'interno di un loop di controllo, comanda gli attuatori per concretizzare degli spostamenti. Qui reactive programming conosce una sua naturale integrazione dovuta alla presenza del loop di controllo. Gli oggetti reattivi si predispongono per leggere aggiornamenti degli input provenienti dall'ambiente, e questi valori saranno poi

propagati per aggiornare le variabili inerenti ai movimenti da effettuare, da inoltrare poi ad un controllore PID, che agirà sui motori finalizzando lo spostamento desiderato.

```
1
2 while(true){
3    readSensors(&gx, &gy, &gz);
4
5    //x, y, z dipendono da gx, gy e gz
6    x = 0.5f * (-q1 * gx - q2 * gx - q3 * gx);
7    y = 0.5f * (q0 * gy + q2 * gy - q3 * gy);
8    z = 0.5f * (q0 * gz - q1 * gz + q3 * gz);
9
10   //a dipende da x, y e z
11   a = 0.5f * (q0 * x + q1 * y - q2 * z);
12 }
```

Listing 2.1: Esempio di loop di controllo

Nel codice 2.1, si nota come le prime tre variabili `x, y, z`, dipendano rispettivamente da `gx, gy e gz`. La quarta dipende dalle 3 precedenti. All'interno del loop vengono lette le 3 variabili g dai sensori, e poi eseguite operazioni matematiche float che impiegheranno la risorsa CPU. Se invece applicassi a 2.1 reactive programming, avrei:

```
1 reactObjects gX, gY, gZ;
2
3 while(true){
4    readSensors(&gx, &gy, &gz);
5
6    gX.update(gx);
7    gY.update(gy);
8    gZ.update(gz);
9 }
```

Listing 2.2: Loop di controllo gestito con oggetti reattivi

Nel codice 2.2 gli oggetti reattivi eseguono le operazioni indicate nel primo esempio propagando i valori letti dai sensori. Però nel caso in cui si abbiano valori di gx, gy o gz che non cambiano rispetto all'iterazione precedente del ciclo, non vengono propagati i valori e di conseguenza non si eseguono calcoli inutili che restituirebbero lo stesso risultato che ho già memorizzato. Questo permette un miglioramento in reattività da parte dell'applicazione, dato che risparmierà computazioni evitabili.

2.3.5 Limitazioni

Questo tipo di paradigma ovviamente non è adatto a tutte le casistiche di programmazione, ma è improntato su tipologie di programmi con determinate ca-

ratteristiche che giovano dell'utilizzo di reactive programming. Infatti scrivere codice con questo paradigma non sempre può risultare un'operazione semplice, trattandosi sostanzialmente di una programmazione ad eventi. Quindi spesso si potrebbe dover rinunciare a questo, a vantaggio di altri paradigmi.

Un altro occhio di riguardo va dato alla gestione della memoria. Infatti non è remota la possibilità di avere una crescita smisurata dello stack, soprattutto in presenza di numerose chiamate a funzioni ricorsive o annidate che propagano i valori da una variabile ad un'altra.

Si può trarre giovamento dall'utilizzo di reactive programming quando si è in presenza di operazioni ripetute, oppure quando si vuole facilmente creare una reazione ad un evento.

Nell'esempio dell'applicazione di reactive programming ad un loop di controllo, in cui ad ogni iterazione si leggono delle variabili e se ne aggiornano altre dipendenti dalle prime, si riscontrerebbero vantaggi nell'uso del paradigma. In questo caso infatti risulterebbe inutile effettuare aggiornamenti se i valori delle variabili lette non sono cambiati. Aggiornando quindi solamente quando un valore cambia, si otterrebbe un miglioramento di prestazioni. In questo caso si avrebbe come evento "la variabile X cambia il suo valore".

2.4 Librerie

In linguaggi in cui non sono presenti implementazioni native di eventi (come ad esempio il C o C++) è necessario implementare i comportamenti che devono avere i vari oggetti al verificarsi di un evento, e come attivare gli effetti dell'evento stesso.

Per sgravare il programmatore dal dover realizzare quanto detto prima, ogni qual volta desiderasse utilizzare degli oggetti reattivi, sono state implementate librerie di supporto alla reactive programming e alla FRP. La maggior parte delle librerie in circolazione supportano la FRP, aggiungendo maggiori possibilità al programmatore grazie all'utilizzo di chiamate annidate a funzioni che possono trasformare il dato propagato.

Non è difficile pensare al modello di esecuzione di reactive programming come un grafo, in cui i nodi sono singole unità in grado di contenere un dato (behaviors) mentre gli archi orientati sono le relazioni che legano i nodi tra di loro e che saranno percorsi dai dati durante le propagazioni attivate (con un evento) a partire da nodi padre.

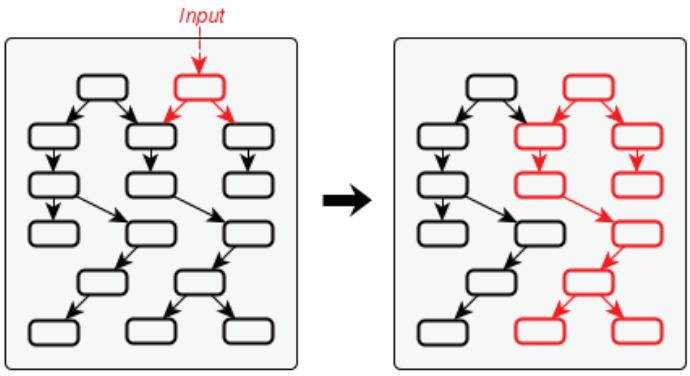

Figura 2.3: Modellizzazione con grafo.

Durante l'esecuzione di un programma reactive programming, il grafo delle dipendenze può trovarsi in 3 stati:

1. Attesa di un evento
2. Elaborazione di un input
3. Propagazione del dato arrivato come input

```
1  begin
2     while(true)
3        //Stato di attesa
4        waitInIdle(Graph)
5        //Arrivo di un input
6        x = readInput()
7        //Propagazione del dato
8        propagate(x)
9     end While
10 end
11
12 //Rimane in stato di attesa fino ad un evento
13 begin waitInIdle(Graph)
14    do
15       Graph <- Idle State
16    till inputArrived()
17 end
18
19 //Propaga il dato lungo il grafo
20 begin propagate(x)
21    Node n <- Graph.inputNode
22    n.value <- x
23    do
24       if n.value is changed
25          n.next.value = n.value
```

```
26        n <- n.next
27    till n.next is Graph.end
28 end
```

Come rappresentato in figura 2.3, la propagazione di dati avviene sempre tra nodi direttamente collegati, e segue la direzione improntata dall'arco orientato. Essa parte dai nodi che ricevono un dato in input dall'esterno, e iterativamente copre tutto il grafo, fino a che questo non è finito. Su questa idea di modellizzazione sono nate nel tempo alcune librerie reactive programming, scritte in diversi linguaggi e di complessità molto varia. Queste si differenziano solitamente per la costruzione dei flussi di dati, per la quantità di oggetti definiti e utilizzati, quindi per la memoria impiegata nella costruzione del grafo, e per le strategie di propagazione dei cambiamenti.

Riguardo queste ultime, si possono distinguere le librerie in due grandi famiglie: librerie push-based e librerie pull-based [8].

- **Librerie Push-based**: dette anche data o input-driven, sono librerie in cui la propagazione dei cambiamenti nei dati avviene quando un nuovo input è disponibile. Questo alleggerisce il sistema dalle operazioni di verifica sulle variabili, quando nessun valore è cambiato. Infatti solamente i componenti pronti per essere propagati saranno considerati. Di norma l'evento push che fa da trigger per l'inizio delle propagazioni, è scaturito dal programmatore invocando nel codice una funzione che attivi il tutto. Questo comportamento si può considerare come erede dalla programmazione imperativa e trova la sua applicazione in situazioni in cui si hanno cambiamenti discreti dei behaviors, cioè quando le variabili cambiano solo a seguito di eventi. Una libreria push-based è meglio utilizzarla in sistemi reattivi come ad esempio una GUI, oppure in sistemi in cui il programmatore vuole il controllo di quando propagare un valore, oppure in sistemi in cui vi sono input letti programmaticamente in modo ciclico nel tempo. Meglio non utilizzarla in sistemi in cui i behavior vengono aggiornati in modo continuo nel tempo, perché se programmati per processare su un singolo thread, dopo aver eseguito una push di un dato, il sistema chiamante va in attesa che la propagazione sia terminata. Questo genera inefficienze e perdite di aggiornamenti di dati nei behavior, dato che il sistema è impegnato ad eseguire ancora la push precedente.

- **Librerie Pull-based**: dette anche demand-driven, sono librerie che azionano il meccanismo di propagazione dei dati quando l'utilizzatore dell'output è pronto per ricevere i dati propagati. Questo implica continue valutazioni dello stato delle variabili, generando conseguente overhead. Però semplifica molto l'integrazione di reactive programming nel caso in cui si stia utilizzando un paradigma puramente funzionale. Si utilizzano in casi in cui i

behaviors vengono aggiornati in modo continuo nel tempo, cioè quando l'unico evento rilevabile può essere la richiesta di dati da parte dell'utilizzatore. Con variazioni di dati discrete nel tempo si potrebbero avere inefficienze nel momento in cui l'utilizzatore chiede più volte lo stesso dato che nel frattempo non è cambiato.

Un'esempio di libreria pull-based è sodiumFRP Questa libreria è contenuta in un progetto più ampio che include più versioni della stessa, compatibili con altri linguaggi. Infatti ad oggi sono state implementate versioni della libreria per C sharp, C++, Java, Haskell, embedded C e Scala. La sua struttura è di media complessità, possiede oggetti per behaviors e functor, e in più rispetto alle precedenti ha anche un gestore degli eventi. Questo perché essendo una libreria pull-based, ha la necessità di rimanere in ascolto sull'evento di propagazione, lanciato dal consumer dei dati, cioè da chi ne necessita.

Nel caso in questione saranno approfondite le librerie push-based, essendo la strategia che meglio si adatta al contesto definito in questa tesi.

Le librerie elencate sono state analizzate e testate su un sistema embedded, quindi verrà fatta anche un'analisi di compatibilità con questi, e in caso di incompatibilità, saranno evidenziati gli elementi che la rendono non cross-compilabile.

Vista la buona compatibilità del linguaggio con i sistemi target per questo progetto, è stato scelto l'utilizzo del C++. Tra le varie librerie reactive programming scritte in C++, ne sono state analizzate dettagliatamente alcune della tipologia push-based.

2.4.1 Cpp.React

Questa libreria, reperibile presso il repository di Shlangster su git-hub, si presenta molto ricca di funzionalità, e molto complessa nella gestione delle propagazioni. E' composta da numerosi oggetti, alcuni dei quali opzionali, e da numerose possibilità di configurazioni delle sue componenti, che permettono di scegliere quale algoritmo di propagazione adottare, se usare o meno multithread e altro.
La libreria è articolata e ha una struttura abbastanza complessa, contenendo 13 classi e numerosi metodi, per questo il consumo di memoria sia RAM che di massa, non la rende adattabile a tutti i sistemi.

Il modello di rappresentazione di un applicativo reactive programming è, come descritto precedentemente, un grafo con nodi e archi orientati. Tale struttura è trasparente all'utente dato che i nodi sono mascherati da leggeri proxies: Signal (che corrispondono ai behavior), Event e Observer. Questi proxies sono oggetti

collegati all'effettivo nodo che rappresentano in memoria e fungono da interfaccia con esso e con tutte le funzionalità di più basso livello di gestione del grafo. Per ricevere dati e propagarli la libreria usa due tipologie di nodi speciali: nodi di input e nodi di output. Tra le funzionalità della libreria c'è quella di poter creare gruppi di valori reattivi tramite i *domini*. I domini in cpp.react sono definiti come dei tag da aggiungere al tipo, quando dichiaro nel codice un nuovo oggetto, e tramite questi posso raggruppare più oggetti reattivi. Ogni gruppo può comunicare con l'altro tramite messaggi asincroni, inviati da nodi di output specializzati alla comunicazione tra gruppi. I domini sono stati implementati con l'intento di semplificare grafi molto complessi, dividendoli in sotto grafi più semplicemente gestibili.

Ogni percorso di propagazione all'interno del grafo, è detto transazione, e ognuna di queste può essere computata in un thread separato, se l'opzione per il parallelismo è attiva. Il parallelismo è gestito tramite la libreria esterna Intel TBB, e ove è presente una parallelizzazione di task, è richiesta un'adeguata gestione della concorrenza. Questa è gestita facendo, ove possibile, dei merge tra le transazioni che si incrociano in un determinato nodo. Il merge è un'operazione di unione di due o più flussi di propagazione in un unico nodo. Se non regolato da una funzione parametrica con i flussi in input, il merge propaga indistintamente in quel nodo i dati dei precedenti flussi nell'ordine in cui questi arrivano.

La propagazione dei dati è gestita dal propagation engine. Questo dà la possibilità di scegliere fra tre diversi algoritmi di propagazione:

- Toposort: è sequenziale e single threaded
- Pulsecount: parallelizza tutti gli update
- Subtree: un misto tra sequenziale e parallelo

Quello che si adatta meglio al target sistema embedded è il **toposort**, non essendo multithread e non avendo quindi problemi con la gestione della concorrenza.

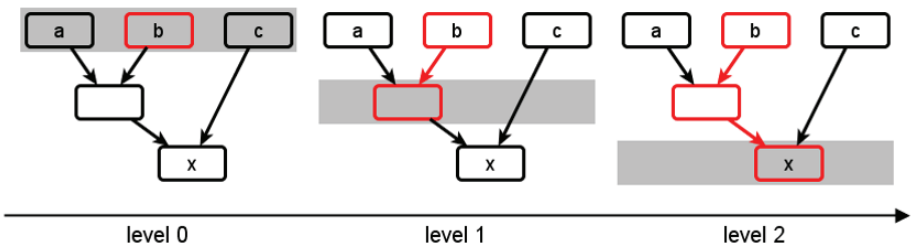

Figura 2.4: Funzionamento algoritmo toposort.

L'algoritmo di propagazione, come rappresentato in figura 1.4, supporta una coda di nodi ordinati per livello (da 0 a n). Ad ogni step viene incrementato il livello, mano a mano che il valore si propaga in profondità nel grafo. In coda vengono messi solo i nodi successivi al nodo input che ha subito una modifica.

Tra gli algoritmi elencati precedentemente, il toposort sequenziale offre minori overhead rispetto agli altri multithread. Soprattutto se si hanno grafi con nodi dipendenti da più thread (può capitare che un nodo vada in attesa della propagazione proveniente da uno o piò thread da cui dipende) Gli altri che sfruttano il parallelismo e pulsecount in particolare, permettono di avere migliori prestazioni se si hanno grafi con un numero molto elevato di nodi e più percorsi di dipendenze che partono da uno stesso nodo input.

La libreria si presenta ricca di funzionalità che per essere gestite richiedono l'utilizzo di più oggetti e strutture dati complesse. Com'è intuibile questo causa sicuramente un'ingente occupazione di memoria, e ciò la rende poco adatta a quei sistemi con limitazioni nelle risorse, come possono essere alcuni sistemi embedded.
Inoltre per la compilazione di cpp.react sono necessarie alcune dipendenze quali Intel TBB che è una libreria di Intel che serve a gestire il multithreading tra i processi, google test framework e boost 1.55, necessaria se si vogliono utilizzare alcune funzioni di cpp.react. A fronte di quanto appena detto, CPP.React risulta non compatibile con le comuni piattaforme embedded, dato che necessita della presenza nel sistema compilante di Intel TBB che non si presta per essere cross-compilato per altre architetture, o se in una versione compilabile, risulta deficitario di alcuni elementi necessari al funzionamento della libreria.

2.4.2 Spira FRP

Questa libreria si presenta estremamente più semplice rispetto alla precedente per quanto riguarda le funzionalità e le classi impiegate. Implementa in maniera molto più diretta rispetto alla precedente, il modello a grafo di dipendenze. Più precisamente ciò che questa libreria permette di costruire con le sue funzionalità, sono grafi con merge e propagazioni con inserimento di funzioni per modificare i dati durante i passaggi da nodo a nodo. Infatti avere una tale semplicità implementativa porta sicuramente a dei benefici prestazionali, visti i consumi ridotti di memoria che la rendono candidabile per essere applicata ad un sistema embedded, però altrettanti svantaggi come perdite dal punto di vista delle funzionalità rispetto a cpp.react. Sono infatti ridotte le possibilità di varianti costruzione del grafo, e non vi è la possibilità di scegliere tra più algoritmi di propagazione.

Tutte le operazioni che si possono fare con questa libreria, sono basate sull'utilizzo di costruttori, che in base a come vengono chiamati e combinati, generano

un determinato grafo di dipendenze. A differenza della precedente non vi è una netta distinzione tra nodi di input e nodi di output, ma potenzialmente qualsiasi nodo creato può essere un nodo di input. Difatti la propagazione è avviata tramite l'invocazione del metodo `Push` contenuto nella classe. Basterà chiamare la `Push` di un determinato nodo, per farlo diventare un nodo di input.

Richiede lo standard C++11 dato che fa uso di functor, che sono costrutti presenti solamente da quella versione di standard. I functor vengono usati per la dichiarazione inline di funzioni, passate direttamente come parametro di altre funzioni, tipico di un paradigma funzionale. Questa modalità di scrittura delle funzioni, permette di organizzare strutture dati di soli puntatori a funzioni salvate in memoria, richiamabili in qualsiasi istante dinamicamente.

A parte il fatto di dover utilizzare lo standard C++11, questa libreria non ha altri requisiti o dipendenze particolari. Questo la rende perfettamente compatibile con un sistema embedded o in generale con la cross compilazione per un microprocessore. Quindi è stato possibile cross-compilarla per un processore Cortex e testarla come vedremo in seguito.

Capitolo 3

Embed Reactive

In questo capitolo sono spiegate le fasi e i criteri per la realizzazione di Embed Reactive, una libreria di supporto a reactive programming con target finale un sistema embedded a risorse limitate. Inoltre sono descritte le caratteristiche e le funzionalità di Embed Reactive, con una breve spiegazione sul suo utilizzo, attraverso una dettagliata documentazione delle sue API.

Nella realizzazione di una libreria reactive programming ci si imbatte in due macroproblemi principali:

- **Il problema della costruzione del grafo delle dipendenze**, che consiste nel trovare un modo per poter collegare gli oggetti reattivi tra di loro in un modello sintetizzato da un grafo, come visto in sezione 2.4, tramite delle regole e delle dipendenze. Questi collegamenti saranno poi le vie tramite le quali i dati si propagano. Durante la costruzione del grafo, i protagonisti sono i behaviors, dato che il grafo si costruisce mano a mano che i behaviors vengono dichiarati e inizializzati nel codice. Il grafo e le dipendenze tra behaviors saranno create in base alle sorgenti di dati e alle relazioni che intercorrono tra essi. Il codice in questo caso si occuperà quindi di inizializzare gli oggetti e del salvataggio delle informazioni riguardanti le relazioni tra behaviors. In questo caso è di interesse la memoria occupata dal singolo behavior e dalle strutture dati per memorizzare le dipendenze. Il consumo di memoria dovrà infatti essere limitato il più possibile.

- **Il problema della propagazione**, che consiste nell'implemantere un metodo per sfruttare i collegamenti tra oggetti costruiti, per propagare i cambiamenti dei dati. La propagazione dovrà iniziare quando un nuovo valore è disponibile, e il codice si occupa di effettuare prima dei controlli sul valore da propagare, in modo tale da non propagare se il valore aggiornato è uguale al precedente, e successivamente di compiere un'esplorazione delle strutture dati create durante la costruzione del grafo. In questo caso è di interesse il tempo che le informazioni impiegano ad essere propagate lungo

tutto il grafo di dipendenze. La propagazione dovrà terminare nel più breve tempo possibile.

Nelle sezioni che compongono il capitolo si parlerà quindi anche delle soluzioni adottate per risolvere i due macroproblemi appena citati, fornendo un'analisi critica sulla bontà delle soluzioni adottate. Prima di iniziare con l'implementazione e dopo aver condotto un'analisi preliminare, esplorando le tecnologie già esistenti descritte nel precedente capitolo, sono stati identificati dei criteri di progetto atti a stabilire le caratteristiche che la libreria deve avere.

3.1 Criteri di Progetto

Embed Reactive si differenzia dalle altre librerie esistenti, proprio per il fatto che è concepita con il fine di supportare rective programming su un sistema embedded a risorse limitate, mentre quelle realizzate sino ad ora non hanno come finalità il funzionamento su questo tipo di sistemi in particolare.

Da ciò si evince che la novità introdotta in questo progetto, è quella di dover rispettare dei vincoli prestazionali legati al tipo di sistema su cui dovrà essere utilizzata la libreria. Questi vincoli saranno legati alla limitatezza delle risorse come la memoria, oltre che a una differente architettura dei microcontrollori rispetto ai sistemi tradizionali.
Nel caso non si rispettino i vincoli prestazionali ci sarebbe l'impossibilità di utilizzare la libreria su un sistema embedded a risorse limitate, a causa dell'esaurimento delle risorse o a varie incompatibilità del codice con l'architettura del sistema preso in esame.

I criteri sono:

- Riduzione del consumo statico sia di memoria RAM che di memoria dedicata allo storage del codice del programma. Quindi la quantità di codice utilizzato per l'implementazione delle funzionalità non deve essere eccessiva, ma ridotta al minimo indispensabile.

- Limitazione alla crescita dinamica dello stack. Prediligere quindi l'utilizzo di variabili dichiarate staticamente rispetto a quello di allocazioni in memoria dinamica. Questo per evitare un eccessivo overhead in memoria. Per rispettare questi primi due vincoli, si è anche disposti a sacrificare prestazioni nel tempo di esecuzione della costruzione delle strutture dati che supportano il grafo delle dipendenze.

- Il tempo di esecuzione della propagazione pur non avendo un upper bound critico, è bene che non sia troppo elevato. Bisogna quindi fare in modo che

la propagazione dei dati e l'esplorazione del grafo delle dipendenze, avvenga più rapidamente possibile. La parte temporale che si può trascurare invece, è quella che riguarda il tempo impiegato per costruire gli oggetti reattivi e le loro associazioni. Infatti questo è un effort che si paga una sola volta all'avvio del programma. E' quindi necessario concentrarsi su altri vincoli prestazionali anche a costo di perdere prestazioni in quest'ottica.

- In ottica di compatibilità bisogna evitare l'utilizzo di librerie esterne e di parallelismo. In particolare la dipendenza da librerie esterne può causare problemi di compatibilità con alcune architetture specifiche. Per fare un esempio pratico, la libreria *TBB* di Intel per la gestione della concorrenza in caso di uso di multithreading non è compatibile e neanche importabile in architetture *ARM*. Quindi il multithreading pur essendo implementabile in programmi per processori *ARM*, andrebbe gestito utilizzando librerie compatibili o evitato, annullando così del tutto i rischi di incompatibilità.

Con uno sguardo ai criteri di progetto, si nota che scegliendo dei vincoli molto restrittivi almeno per ciò che riguarda il consumo delle risorse, si assicura un funzionamento della libreria su un numero più elevato di sistemi. Quindi la quantità di risorse utilizzate è inversamente proporzionale al numero di piattaforme ai quali è possibile applicare la libreria.

3.2 Caratteristiche

Vengono presentate ora le caratteristiche generali e le proprietà della libreria.
Per quanto riguarda il linguaggio di programmazione è stato scelto lo standard C++11 perché già compatibile con quasi la totalità dei sistemi embedded e compilatori, e perché offre la possibilità di utilizzare Espressioni Lambda, la cui utilità vedremo in seguito.

L'evaluation model, proprietà descritta in sezione 2.2.2, è stato scelto di tipo push-based, per dare la possibilità alla libreria di essere applicata in sistemi in cui è la sorgente dei dati (il produttore) a iniziare la propagazione dei cambiamenti verso il consumatore, mentre non è possibile stabilire quando il consumatore avrà bisogno dei dati per richiederli, caratteristica questa dell'evaluation model pull-based. La libreria inoltre permette la creazione di oggetti denominati `Signal`, che sono i corrispondenti dei behaviors in reactive programming. Il nome Signal è scelto prendendo spunto dalle altre librerie reactive programming esistenti.

Un Signal può essere specializzato al momento della dichiarazione con qualsiasi tipo supportato dallo standard C++11, che sarà anche il tipo del valore che rappresenterà.

Ricordando quanto detto nel capitolo 2, in reactive programming i valori sono soggetti ad una propagazione a seguito del verificarsi di un evento. La propagazione avviene sempre da un nodo padre verso i nodi da esso dipendenti. Si può quindi fornire una modellizzazione logica di questo comportamento, pensando ad un grafo di dipendenze in cui i nodi sono i signal mentre le dipendenze tra nodi sono archi direzionati.

Ciò che in sintesi la libreria permette di fare è creare un grafo generico di dipendenze, senza l'obbligo di utilizzare dei nodi specializzati come nodi di input che avviano la propagazione dei valori, come invece avveniva ad esempio in `cpp.react`, e con la possibilità di scrivere qualunque funzione matematica con la quale si vuole aggiornare i nodi durante la propagazione dei valori tra le varie dipendenze. Il grafo delle dipendenze viene realizzato per mezzo dei costruttori. Infatti al momento della dichiarazione, un nodo viene appeso al grafo in una posizione e con una modalità che varia al variare del costruttore utilizzato e dei parametri ad esso passati.

Per meglio comprendere come sono state implementate le funzionalità, è utile adesso fare un breve approfondimento sul C++11 e sugli strumenti che offre, che hanno reso possibile una maggiore flessibilità implementativa.

3.3 C++11

Il C++11, è uno standard recente per il C++ che sostituisce la revisione del 2003. Questo comprende numerose novità per il core del linguaggio ed estenderà la libreria STD incorporando molte nuove funzionalità. Inizialmente poco diffuso, ora può vantare di un'ottima compatibilità con i compilatori più diffusi.

Una delle ragioni che spingono ad un processo evolutivo un linguaggio di programmazione come il C++ è la necessità di poter programmare più velocemente, elegantemente e, soprattutto, ottenendo un codice meno complesso la cui manutenzione sia agevole.

```
1 int main()
2 {
3     //definizione di un'espressione lambda
4     auto func = [] () { cout << "Hello world"; };
5     func(); // chiamata alla funzione
6 }
```

Listing 3.1: Esempio espressione lambda

Tra le novità introdotte dal C++ di maggiore interesse per questo progetto, sono senza dubbio le espressioni lambda, che sono risultate essere essenziali per poter

semplificare notevolmente l'utilizzo di Embed Reactive da parte del programmatore. Degni di nota sono anche i puntatori alle funzioni, in realtà già inseriti nella versione del 2003, ma in questa perfezionati con anche la creazione di un oggetto ad essi dedicato: l'oggetto *function*. Le espressioni Lambda sono delle funzioni senza nome che definiscono delle operazioni tipiche di una normale funzione, e che sono invocabili tramite un reference restituito dall'espressione lambda stessa al momento della dichiarazione. Il vantaggio tratto dal loro utilizzo, oltre all'eleganza e leggibilità del codice, è anche di avere la possibilità di creare funzioni in modo dinamico, senza doverle precedentemente dichiarare nel codice e invocarle poi quando servono. Dopo la sua creazione, la funzione realizzata tramite espressione, è identificabile tramite il suo indirizzo.

Con riferimento al listato 3.1, che rappresenta la definizione con relativa chiamata di un'espressione Lambda che stampa a video una stringa, si precisa che tra le parentesi quadre in riga 3, si possono all'occorrenza inserire le variabili che si vogliono passare al momento della scrittura della funzione, per poterle utilizzare al suo interno. Tra le tonde invece, le variabili che si vogliono passare come parametri della funzione. In questo caso il funzionamento è come quello di una normale funzione, cioè passo i parametri al momento dell'invocazione.

Un'espressione lambda permette di identificare la funzione che implementa tramite un reference. Perciò il tipo restituito dall'espressione, è un oggetto introdotto sempre a partire dallo standard c++11, chiamato `std::function`. Questo è adibito alla gestione di un puntatore ad una funzione e sostituisce il precedente (*nomeFunct)(). Un'espressione lambda può essere anche dichiarata come parametro ad un'altra funzione, con evidenti potenzialità soprattutto in presenza di un paradigma funzionale.

```
1  Void foo(double *a, double b, std::function func)
2  {
3      *a = func(b);
4  }
5
6  Int main(){
7        double x,y;
8  //foo contiene un'espressione lambda
9  foo(&x, y, [](double d) {
10
11              if (d < 0) {
12                  return 0;
13              } else {
14                  return d;
15              }
16          });
17 }
```

Listing 3.2: Espressione lambda in chiamata a funzione

Nell'esempio 3.2 la funzione foo riceve in input 2 interi e un puntatore ad una funzione. Questa funzione quando sarà invocata (all'interno dell'implementazione della funzione *foo*) riceverà in input un double e restituirà un valore. E' importante notare che la funzione foo non riceve in input il risultato restituito dalla funzione generata con l'espressione lambda, bensì riceve una funzione invocabile al suo interno. Questo sposta la gestione della chiamata della funzione creata, all'interno della funzione *foo*, ma si comprenderà in seguito il vantaggio di questa caratteristica.

L'utilizzo di espressioni lambda oltre a mantenere il codice più compatto (soprattutto se le funzioni in questione sono brevi) aggiunge una flessibilità notevole. Questo, menzionando il mio progetto, permette di risolvere il problema di creare dei metodi personalizzabili dal programmatore, il che è importante dato che la libreria si deve adattare ad un numero più grande possibile di situazioni in cui un programmatore che utilizzerà la libreria, può trovarsi a dover gestire.

Con le espressioni lambda è inoltre possibile usare le variabili dichiarate esternamente alla funzione, e questo è un grosso vantaggio perché dà la possibilità di utilizzare qualsiasi variabile al suo interno come se fosse una variabile globale già in fase di dichiarazione dell'espressione, altrimenti si potrebbero utilizzare solo parametri passati al momento della chiamata alla funzione generata con l'espressione lambda. Tutto ciò permette una maggiore flessibilità di invocazioni, potendo creare facilmente serie di chiamate a funzioni annidate che possono accedere al contesto del chiamante. Questa proprietà sarà poi utile per realizzare le propagazioni automatiche nella libreria reactive programming.

3.4 Funzionalità

In questa sezione sono descritte le funzionalità, documentate con la descrizione di alcune API significative, le loro funzionalità e i problemi che risolvono. Nella sezione successiva poi, si entrerà nel dettaglio su come sono state implementate. Per l'organizzazione delle funzionalità in questa sezione, vengono presi come riferimento i due macro problemi enunciati a inizio capitolo, iniziando con il problema del costruire un grafo di dipendenze tra nodi.

3.4.1 Costruzione del Grafo di Dipendenze

Il problema della costruzione di un grafo di dipendenze tra nodi, viene risolto dai costruttori della classe `Signal`, che come detto è chiamata in tal modo prendendo spunto dalla nomenclatura già esistente in altre librerie. Un oggetto di

tipo `Signal`, rappresenta il behavior della libreria. Esso contiene un valore e, contestualizzandolo alla modellizzazione a grafo vista nel capitolo 2, può essere visto come un nodo del grafo.

Ogni costruttore, tra quelli forniti dalla libreria, permette un tipo diverso di collegamento tra due o più nodi e di impostare la modalità di aggiornamento del valore del nodo creato.
Di costruttori, ne sono stati creati 5, in numero tale da garantire la massima flessibilità nel creare le dipendenze, permettendo molteplici combinazioni. Le modalità con le quali si collegano i nodi tra di loro, vanno ad influenzare poi l'effetto dei passaggi dei dati da un nodo all'altro all'interno del grafo, cioè la propagazione di essi.

Propagazione Lineare e Split

Come caso più semplice che si può incontrare nella costruzione di un grafo, si ha il caso in cui si voglia collegare tra di loro solamente due nodi in una relazione padre-figlio.

Figura 3.1: Propagazione lineare.

```
1 Signal(Signal sA);
```

Questo costruttore permette di dichiarare un signal e collegarlo ad un altro signal già esistente, da cui poi riceverà aggiornamenti del valore che sarà uguale al propagato.
La relazione tra i valori è di assegnamento, quindi si può dire che questo sia il

costruttore che permette il più elementare collegamento padre-figlio tra due nodi. Inoltre permette di fare lo split su più nodi di un flusso di dati proveniente da un unico nodo. Questo è possibile dichiarando più oggetti con questo costruttore e passandogli in input sempre lo stesso nodo, che sarà il padre di tutti i nodi creati.

Merge Tra Due Flussi di Dati

Quando si costruisce un grafo ci si potrebbe trovare in una situazione in cui si vuole creare un nuovo nodo, e collegarlo a due nodi padre preesistenti. In questo caso si parla di merge tra flussi di dati, dato che durante una propagazione di valori all'interno del grafo, questi seguono i collegamenti tra nodi andando a creare dei flussi di dati propagati.

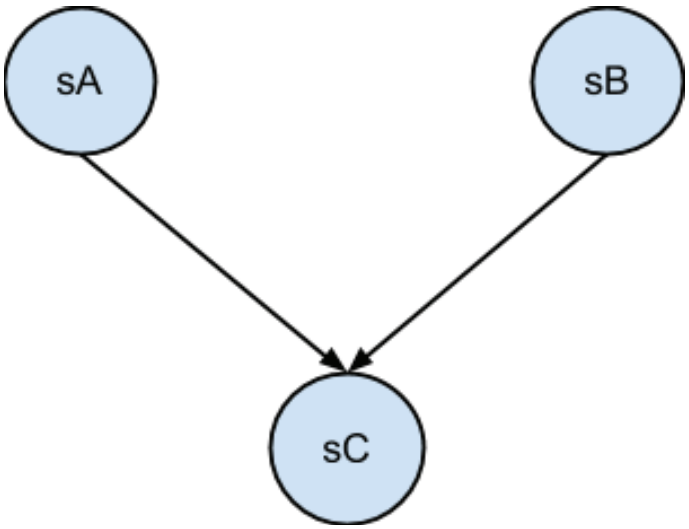

Figura 3.2: Merge tra due flussi.

Immaginiamo infatti di avere la struttura in figura 3.2. In questo caso i due nodi padre riceveranno degli aggiornamenti del proprio valore, o dall'esterno o da nodi precedenti. Dato che il nodo figlio è dipendente dai due precedenti, il suo valore sarà aggiornato in funzione dei primi. Ciò significa che al variare di uno e/o dell'altro, il nodo figlio aggiornerà il proprio valore di conseguenza, ricevendo il dato di uno o entrambi i nodi padre.
Per poter realizzare questo tipo di dipendenza interviene il seguente costruttore:

```
1 Signal(Signal sA, Signal sB);
```

Questo costruttore di merge permette di dichiarare un oggetto signal dipendente da altri due oggetti, `sA` ed `sB`. Il valore del nuovo oggetto creato, sarà aggiornato tramite un assegnamento del valore di `sA` oppure di `sB`. Questo vuol

dire che, dopo la propagazione, il nuovo valore dell'oggetto inizializzato con il costruttore, sarà uguale o a quello di `sA` o a quello di `sB`. E' facilmente intuibile che nel caso entrambi i nodi precedenti propaghino i valori, il nuovo oggetto assumerà prima il valore di uno poi dell'altro, in ordine cronologico di propagazione.

Inserimento di una Funzione tra i Merge

Anche se nella precedente situazione si è utilizzata un'operazione di assegnamento per propagare i valori da padre in figlio, potrebbero esserci casi in cui oltre al merge tra due o più flussi, si voglia coinvolgere tutti questi flussi in una operazione matematica, che restituisca un unico valore da propagare al nodo figlio in cui questi flussi convogliano.

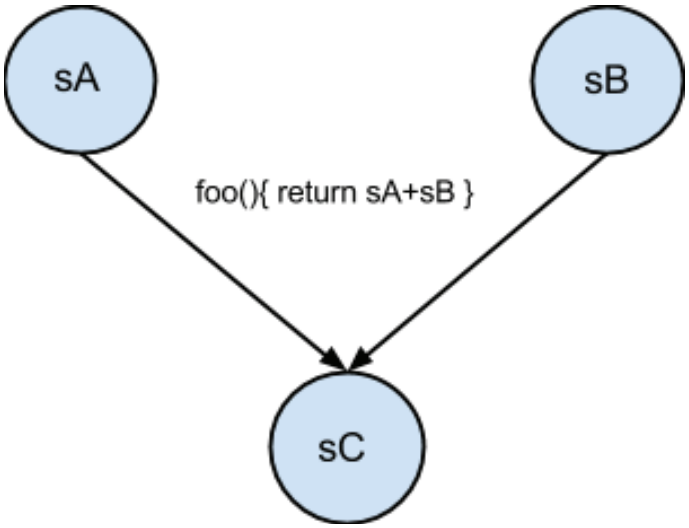

Figura 3.3: Il merge tra due flussi regolato da una funzione.

Per poter realizzare un collegamento come quello in figura 3.3, bisogna inserire e invocare al momento della propagazione, una funzione matematica che, per garantire la massima flessibilità, dovrà essere definita dal programmatore. A questo proposito è stato creato il seguente costruttore.

```
1 Signal(foo(), nodi[n]);
```

Listing 3.3: Costruttore con funzione

Il costruttore nel listato 3.3 riceve in input gli n nodi che si vogliono unire con un merge nel nuovo nodo che si sta creando, e la funzione che a partire dai valori propagati da tutti gli n nodi, genererà il valore del nodo figlio di tutti. Quindi oltre a unire i flussi di dati, che nell'esempio in figura 3.3 sono 2 ma possono essere di arbitrario, in questo caso viene generato un valore che è il risultato di una funzione matematica che coinvolge come parametri tutti i dati dei flussi che

si stanno unendo. La funzione sarà valutata nel momento esatto in cui il dato viene propagato da uno dei nodi padre al figlio.

La funzione da inserire in questo costruttore deve essere generata tramite espressione lambda e quindi passata con un parametro di tipo `std::function`, come si vedrà nella descrizione dell'implementazione. Ma se si volesse generare un merge con una funzione dichiarata da un'altra parte del codice, rendendo quindi il programmatore indipendente dal contesto in cui dichiara l'oggetto, è necessario optare per un'altra tipologia di costruttore che risolva il problema del collegare a dei nodi una funzione definita esternamente.

Passaggio di una Funzione per Indirizzo

Premettendo che lo schema finale del grafo e il funzionamento della propagazione sono identici al caso precedente, in questo caso si ha la libertà di poter passare come parametro al costruttore una funzione già dichiarata. Questo però causa una perdita in flessibilità, dato che il numero di parametri da passare alla funzione deve essere noto a priori. Questo numero nel costruttore creato è standardizzato a 2, il programmatore però può modificare tale numero in base alla funzione che necessita di passare al costruttore. Questa scelta è stata fatta per fornire al programmatore la possibilità di creare funzioni occupandosi solo dell'implementazione delle operazioni matematiche di aggiornamento dei valori, non obbligandolo a dover implementare del codice atto a gestire un numero variabile di argomenti. Infatti mentre nel caso precedente la gestione degli input della funzione avveniva internamente alla libreria, in questo caso se ne deve occupare il programmatore.

```
1 signal(Signal s1, Signal s2, T indirizzoFunzione(T,T));
```

Listing 3.4: Costruttore con reference a funzione

Il costruttore indicato nel listato 3.4 riceve in input il reference della funzione di aggiornamento scrivibile dal programmatore che in questo caso accetta 2 parametri, e poi i nodi padre da unire nel merge in questo nodo figlio creato con il costruttore stesso. Come si nota dal codice, i valori che vengono passati come parametri alla funzione in input al costruttore, sono i valori dei due nodi i cui flussi di dati si stanno unendo.

Bind

Oltre ai costruttori tra i metodi pubblici utilizzabili dal programmatore troviamo il metodo `bind`, il metodo `push` e il metodo `get value` che è una normalissima funzione di interfaccia che restituisce il valore del signal definito come privato, e che non necessita di ulteriori documentazioni.

Viste le possibilità che offrono i costruttori, è sufficiente il loro utilizzo per permettere al programmatore di utilizzare tutte le features della libreria. Però per dare maggiore flessibilità e permettere di agganciare oggetti tra di loro in un istante successivo alla loro dichiarazione, è presente il metodo `bind` che permette di creare dipendenze in qualsiasi momento, con l'unico vincolo che impone che l'oggetto dal quale si invoca la bind sia già stato dichiarato tramite uno dei costruttori. Quindi è possibile andare a collegare ad un altra parte del grafo, un oggetto già esistente. Questo è lo si fa aggiungendo una dipendenza al grafo utilizzando il metodo bind.

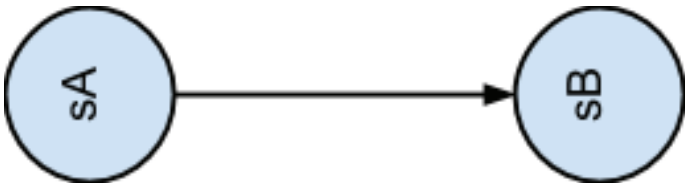

Figura 3.4: sA dichiarato come padre di sB.

```
bind(numeroNodi, funzioneInserita(), nodi);
```

Listing 3.5: Prototipo della funzione bind

```
//dichiaro due oggetti
signal sA;
signal sB;

sB.bind(1, std::function<int()>([&sA](){
            return sA.getValue();
       }), sA);
```

Listing 3.6: Esempio utilizzo bind

La bind mostrata nei listati 3.6 e 3.5, riceve in input il numero di elementi da cui deve dipendere il valore di sB (in questo caso solo sA quindi 1), la funzione di aggiornamento scritta con una Lambda Expression passando per indirizzo la variabile esterna da rendere visibile all'interno di essa senza passarla come parametro (reference di sA) e il nodo padre a cui collegare con un arco di dipendenza sB. Così facendo sono stati creati due oggetti `Signal` e collegati entrambi direttamente come mostrato di seguito.

In questo caso, come mostrato in figura 3.4, la funzione di aggiornamento è una pura propagazione del valore di sA in sB. Questa funzione si deve scrivere come Lambda Expression grazie alla quale inserisco in modo dinamico una funzione di assegnamento inline all'interno della chiamata stessa della bind, come mostrato nel codice precedente.

Il funzionamento della bind è speculare con quello del costruttore che permette il passaggio di una funzione per indirizzo, descritto nella sezione 3.3. A differenza di questo però dopo la sua invocazione non viene allocato un nuovo elemento, ma vengono generate nuove dipendenze.

Con tutte le API presentate sino ad ora si è fatto un quadro generale su tutti gli elementi utilizzati per risolvere il problema della costruzione del grafo delle dipendenze.

3.4.2 Propagazione degli Aggiornamenti

Una volta che si ha una struttura di dipendenze definita, si presenta la problematica del come poter utilizzare i collegamenti costruiti tra gli oggetti, per propagare i valori. A questo scopo viene utilizzato il metodo `push`, unico responsabile della propagazione.

```
sA.push(2); //inizia la propagazione di 2 da sA
```

Listing 3.7: Esempio utilizzo push

```
void push(valore)
begin

   se vecchioValore == nuovoValore
      return

   vecchioValore <- nuovoValore

   foreach(n in nodiFigli){
      n.push(nuovoValore) //propaga valore a nodi figli
   }
end
```

Listing 3.8: Funzionamento della push

Il metodo `push` accetta in input un valore che deve essere dello stesso tipo del valore dell'oggetto `Signal`, come mostrato nel listato 3.7 in cui si vuole propagare il valore intero 2. Il metodo `push` aggiorna il valore dell'oggetto che lo invoca, con quello ricevuto in input e, appurato che il nuovo valore non è uguale al precedente, lo propaga lungo i nodi successivi, che sono gli oggetti i cui valori dipendono dal chiamante. Se non vi sono nodi successivi aggiorna solamente il valore con quello in input, come mostrato con lo pseudocodice del listato 3.8.

Push con Condizione

Per comprendere meglio l'esempio presente in questa sottosezione si genera, con il costruttore mostrato con codice C++ nel listato 3.9, un grafo contenente 4 nodi A,B,C e D. Si ha D che dipende dagli altri 3 e il suo valore è dato dalla somma dei 3 valori di A,B e C. Il grafo è quello rappresentato nella figura 3.5

```
signal D = new signal(3, 0, [](){return A.getValue() +
                        B.getValue() + C.getValue();});
```

Listing 3.9: Dichiarazione di un nodo figlio di altri 3

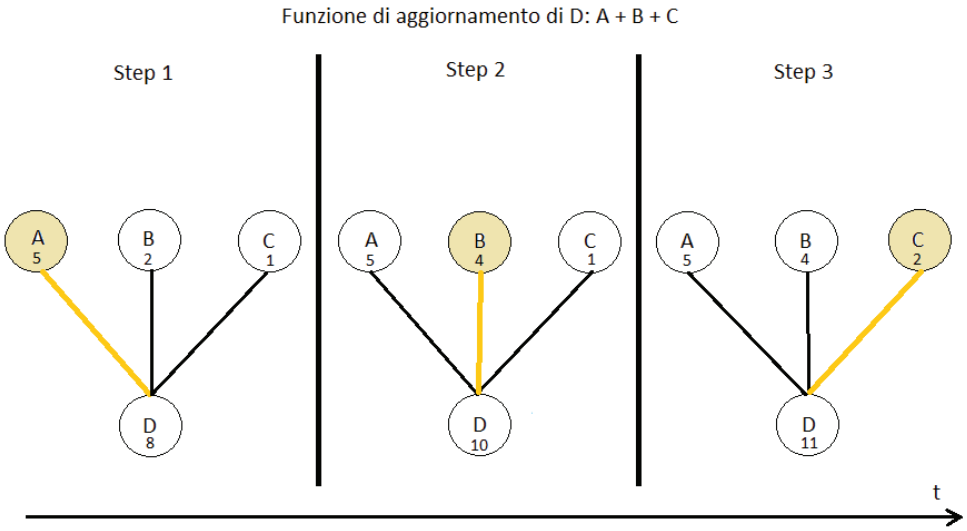

Figura 3.5: 3 push consecutive verso lo stesso nodo.

E' possibile trovarsi in una situazione in cui vi sono più nodi collegati tutti ad un medesimo nodo figlio tramite un merge regolato da una funzione. I nodi padre aggiornano il proprio valore in modo ripetitivo nel tempo ma in istanti non predicibili. Non si conosce inoltre la frequenza con cui i valori dei nodi padre cambiano nel tempo.
Prendendo come riferimento l'esempio in figura 3.5, ho 3 nodi che aggiornano tramite la somma dei loro valori il nodo D. Se propagassi da subito i valori di A e B negli step 1 e 2 della figura, effettuerei inutili calcoli, dato che il valore propagato in D verrebbe subito scartato nello step successivo. Questo perché nei primi due step effettuo la somma (A+B+C) con valori vecchi di B e C, mentre il risultato finale considera tutti e 3 i valori aggiornati all'ultima versione.
Può essere quindi necessario in questi casi, eseguire comunque le tre push aggiornando solamente il valore dei primi due nodi, e invece che avviare la propagazione verso D, avviarla solamente dopo l'esecuzione dell'ultima push eseguita dal nodo

C. Questo per ottimizzare l'esecuzione visto che le due push precedenti all'ultima, risulterebbero inutili considerando il fatto che il risultato finale è dato da un'espressione che include in ogni caso i valori dei tre nodi padre aggiornati all'ultima lettura.
Nell'esempio 3.5, se la frequenza di aggiornamento dei valori delle sorgenti dati dai quali A, B e C aggiornano il proprio valore, fosse un valore noto e definito costante nel tempo, si potrebbe gestire la situazione tramite una corretta temporizzazione delle push. Nella situazione sopra esposta invece, c'è la necessità di aggiornare in ogni caso i valori di A e B, ma di propagare solo dopo che anche il valore di C sarà stato aggiornato.
Prendendo spunto sempre dall'esempio in figura 3.5, si può ipotizzare una situazione simile in cui però il nodo C non cambi il proprio valore e quindi teoricamente non necessiterebbe di una propagazione. In questo caso bisogna fare in modo che la propagazione avvenga comunque perché altrimenti si perderebbero le variazioni avvenute nei nodi A e B. Quindi C deve valutare se nei nodi A e B è avvenuto un cambiamento di valore, in caso affermativo propaga, altrimenti non propaga.

Per risolvere questa problematica di ottimizzazione esposta con gli esempi appena sopra, è stata pensata una funzione push modificata, che permette di scegliere se propagare il valore o solamente aggiornare quello del nodo corrente. Questo tipo di push è conveniente utilizzarla solo se si verificano le 2 condizioni:

- La frequenza di aggiornamento dei valori dei nodi padre non è un valore noto o costante nel tempo.
- Tutti gli oggetti che fanno uso di questa push devono essere tutti quanti legati tra loro tramite una relazione matematica con la quale viene propagato un valore allo stesso figlio in comune.

Questa tipologia di push verifica che tra tutti i nodi padre non vi siano state modifiche nel loro valore. Nel caso non vi siano state modifiche, la push non propaga nulla e non esegue la funzione di aggiornamento del figlio, risparmiando computazioni per eseguirla che non sarebbero quindi necessarie. Ma se anche solo uno tra i nodi padre ha aggiornato il proprio valore, è necessaria una propagazione che avverrà solo dopo aver valutato per tutti i nodi padre se ci sono stati cambiamenti di valore. Da adesso in avanti questa funzione sarà chiamata `push` condizionata per distinguerla dal metodo push descritto precedentemente.

Rate di Propagazione

Vi può essere l'esigenza di ottimizzare la situazione in cui ho, un oggetto il cui valore dipende dal risultato di una funzione matematica calcolata da due o più oggetti che si aggiornano ad una diversa frequenza. Per capire meglio la situazione è fornito il seguente esempio:

C'è un nodo **C** dipendente da due nodi **A** e **B**, legati ad esso tramite una funzione matematica:

```
C.valore = A.valore * B.valore
```

Ipotizzando i valori di A e B legati alle letture effettuate da fonti di dati che lavorano ad una frequenza differente, con ad esempio A che si aggiorna ogni secondo e B ogni 2, e ipotizzando che il valore C serva sempre aggiornato alle ultime letture che aggiornano A e B, risulterebbe quindi inutile propagare il valore dell'espressione ogni secondo (periodo della fonte dati più veloce), se B ha ancora un valore risalente alla lettura precedente. Con la funzionalità di impostazione del rate di aggiornamento, si risolve questo problema, impostando il rate di A con un valore doppio rispetto a quello di B. L'impostazione del rate avviene tramite i costruttori al momento dell'inizializzazione di un nuovo nodo oppure tramite un metodo specifico chiamato `set rate`. Nei costruttori è infatti previsto, oltre ai parametri descritti precedentemente, un parametro dedicato alla definizione del rate di aggiornamento.

Capitolo 4

Implementazione

Volendo caratterizzare il tipo di propagazione implementata, si specifica che essa avviene in modo lineare, single thread, con una politica di aggiornamento del grafo logico delle dipendenze di tipo depth-first. L'ordine all'interno del grafo con cui vengono effettuate le propagazioni, prende in considerazione l'ordine con cui i nodi sono stati creati, sempre nel rispetto della politica depth-first.
Questo significa che se un oggetto inizia la sua propagazione, questa parte sullo stesso thread del processo dal quale viene invocata la push, che quindi rimane in attesa della fine delle propagazioni prima di continuare con la sua normale esecuzione.
Queste chiamate a `push` annidate causano una crescita dello stack proporzionale al numero di propagazioni, che raggiunge il suo apice quando si arriva agli ultimi elementi del grafo (quelli che non hanno figli) e che poi torna delle dimensioni di partenza che aveva prima dell'inizio delle propagazioni.

Dopo aver presentato le funzionalità, si vuole descrivere più approfonditamente in questa sezione, le tecniche utilizzate per implementarle, mostrando e motivando le scelte di programmazione effettuate. In questa sezione si scende volutamente ad un livello di dettaglio più basso, per approfondire alcuni dei concetti espressi precedentemente.

Il codice della libreria è implementato su 3 file C++: uno con la definizione del namespace, della classe e degli header delle funzioni, un altro con l'implementazione dei metodi e degli attributi della classe `Signal` e un terzo file che contiene la classe `Connector`, di supporto alla creazione delle dipendenze tra i vari nodi del grafo delle dipendenze, e che gestisce le strutture dati necessarie.

Le funzionalità della libreria, implementate nella classe `Signal` sono supportate dalla presenza di attributi in grado di memorizzare le informazioni essenziali per la gestione degli oggetti reattivi e la loro interazione.

A inizio capitolo si è parlato di 2 macroproblemi, questi a loro volta sono stati scomposti in altri problemi più piccoli risolti con le feature presentate nella sezione precedente.
Il problema della costruzione del grafo si può vedere come il problema del creare legami tra due o più nodi.
I legami devono permettere un passaggio di valori da un nodo all'altro, quindi devono essere sintetizzati come un tipo di informazione in grado di permettere ad un nodo di comunicare con altri.

4.1 Collegamento tra Oggetti

Come detto nella sezione precedente, i costruttori sono responsabili della creazione di tutti i collegamenti di dipendenze tra nodi. La creazione delle dipendenze tra nodi tramite i costruttori, avviene sfruttando le espressioni Lambda. Infatti una dipendenza padre-figlio tra due nodi è realizzata tramite un riferimento, dato dall'indirizzo generato dall'espressione Lambda, alla funzione `Push` del nodo figlio. Per poter salvare il reference alle funzioni push, serve una struttura dati accessibile dai nodi.

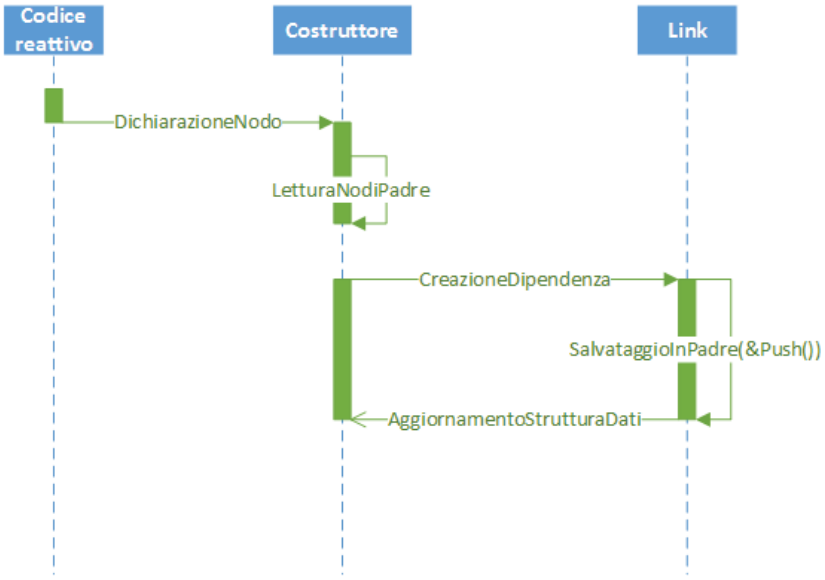

Figura 4.1: Dichiarazione e collegamento del nodo nel grafo.

Il costruttore, per generare le dipendenze tra nodi, si serve di un metodo chiamato *link* che si interfaccia con le strutture dati finalizzate a contenere i reference delle push dei nodi che dipendono da altri. Questo per rendere al programmatore totalmente trasparente il modo con cui vengono gestite le informazioni delle dipendenze tra nodi. Riferendosi alla figura 4.1, durante la scrittura del codice e la

dichiarazione di un nuovo nodo, basterà invocare il costruttore specificando quali devono essere i nodi padre, per salvare in maniera automatica all'interno dei nodi padre i riferimenti alla funzione *Push* del nuovo nodo che si sta dichiarando.

4.2 Propagazione dei Valori

Avendo visto che il problema di collegare i nodi, si risolve facendo chiamare al nodo padre le push dei nodi figli, si approfondisce l'implementazione della propagazione dei valori. Dato che per poter implementare il collegamento tra nodi si sfruttano i reference delle funzioni push, per poter poi chiamare da un nodo padre la push appartenente ad un altro nodo, è necessario leggere il puntatore a tale funzione dalla struttura dati popolata durante la costruzione del grafo.

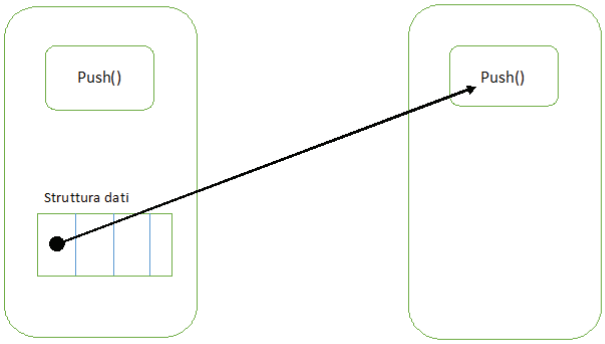

Figura 4.2: Collegamento a push di altri oggetti.

L'idea che il collegamento sia realmente fatto tra due nodi padre-figlio fa parte sempre di quella modellizzazione logica che vede la struttura reattiva come un grafo. Ma a livello implementativo, quando si parla di "collegamento tra nodi", si verifica un salvataggio di puntatori a metodi push appartenenti ai nodi figli all'interno delle strutture dati accessibili dal nodo padre. Quindi il collegamento logico rappresentato in figura 3.4, diventa in realtà, implementandolo, quello mostrato nella figura 4.2.

Come si è visto però nel capitolo 3, all'interno della funzione push è necessario passare un valore che può essere frutto di una propagazione proveniente da un unico nodo padre, oppure un risultato di una funzione matematica nella quale sono implicati valori di più nodi padre. Per passare il risultato di una funzione, è necessario creare un'espressione Lambda che sarà poi valutata al momento dell'invocazione della push. Il metodo push, ha la caratteristica di aggiornare il grafo seguendo le dipendenze tra i nodi.

La funzione push non avvia una propagazione in modo incondizionato, bensì come prima operazione aggiorna il valore dell'oggetto dal quale viene invocata, poi verifica alcune condizioni che, se vere, permettono l'inizio della propagazione. Tali condizioni sono la verifica sul rate di aggiornamento per quel nodo, e su un parametro booleano nel caso si stia usando la variante della funzione di push descritta precedentemente come `push` condizionata.

Per poter propagare, il metodo `push`, ha accesso alle strutture dati popolate durante la costruzione del grafo delle dipendenze. Ricordando che nelle strutture dati sono salvati i puntatori alle funzioni `push` dei vari nodi del grafo, per ogni puntatore letto dalle strutture dati, viene invocato il metodo push sfruttando il puntatore stesso e passandogli in input il valore aggiornato da propagare. Il metodo push invocato, a sua volta potrà procedere alla lettura di altri puntatori ad altre push e così via.

Si è creata così una catena di invocazioni ricorsive che terminerà quando la struttura dati accessibile dal nodo che sta propagando, è stata letta completamente. Una volta terminata la propagazione, il programma principale nel quale si sta utilizzando la libreria, continua con la sua esecuzione.

Così si è risolto il problema della propagazione automatica dei dati. Infatti sarà sufficiente al programmatore invocare nel codice la prima push, che logicamente deve essere la push appartenente al nodo di input del grafo, cioè quello dal quale partono le propagazioni. Dopo di che il tutto avviene in maniera automatica, fino ad aver esplorato tutto il grafo.
Il modello ricorsivo di esplorazione del grafo è basato su una strategia depth-first. Infatti la funzione di propagazione presente nella push, esplora nodi tra essi fratelli all'interno del grafo, però per ogni nodo, prima di passare al fratello, esplora tutti i suoi figli ricorsivamente.

Se si volesse infine fare uso della `push` condizionata, essa è utilizzabile solo dopo aver attivato la funzionalità tramite la definizione di una costante nel codice. Questo perché non essendo una funzionalità essenziale e il cui utilizzo si può rivelare utile frequentemente, è stato scelto di non creare in modo permanente variabili aggiuntive, che occuperebbero quindi più memoria, se non strettamente necessarie e attivabili a discrezione del programmatore. La parte attivata nei costruttori con la definizione della costante, è una differente implementazione del collegamento della push ai nodi padre. Infatti nel caso in cui la `push` condizionata sia attivata, la funzione di push viene inserita nelle strutture dati solo se una condizione è verificata. Questa condizione verifica se nelle `push` che per ordine cronologico di invocazione precedono quella considerata, abbiano o meno causato degli aggiornamenti alle variabili degli altri nodi implicati nella propagazione.

Il ruolo dell'elemento booleano è quello di identificare se tra tutti i nodi padre implicati nelle propagazioni verso lo stesso figlio, vi sono stati cambiamenti nei valori. In caso affermativo viene eseguita una propagazione verso il nodo attuale. In caso contrario non succede nulla. Si tratta di fatto del caso esposto nell'esempio in figura fig:cond-push nel capitolo precedente.

Fino ad ora si è parlato di strutture dati volutamente in modo generale. Questo perché si possono scegliere diverse soluzioni compatibili con il funzionamento della libreria. In base alla scelta effettuata però si avranno caratteristiche prestazionali molto differenti, essendo la struttura dati che implementa i collegamenti tra nodi, l'elemento principale della libreria ed anche quello più dispendioso in termini di consumo RAM e di tempo di esplorazione.
Modificando quindi la struttura dati e di conseguenza adattandone la sua gestione, è possibile cambiare profondamente le caratteristiche non funzionali della libreria. Proprio per questo è stato naturale ritrovarsi al termine dei lavori, con due versioni di libreria: una prima che è stata implementata sulla base di Spira incentrandosi soprattutto sul rispetto delle funzionalità, e una seconda che è stata implementata successivamente per rispondere alla necessità di risparmio di memoria RAM.

4.3 Alternative per Strutture Dati

Durante l'implementazione e l'evoluzione della libreria sono state utilizzate due alternative di strutture dati. Le differenze e i miglioramenti prestazionali avuti cambiando la tipologia di strutture dati saranno esposti di seguito, facendo un'analisi utile a comprendere i risultati dei test presentati nel capitolo delle valutazioni. Ai fini di comprendere i ragionamenti che sono stati fatti per migliorare la libreria implementando la struttura dati finale, è utile fare un accenno agli STL container e ad un confronto tra le loro caratteristiche. Questo perchè come detto, la scelta della tipologia di struttura dati, si rivela di primaria importanza per quelle che saranno le prestazioni della libreria.

4.3.1 Gli STL container

La Standard Template Library (STL) costituisce la parte del linguaggio ormai divenuta fondamentale per i programmatori C++. Essa fornisce un set precostituito di classi comuni, come diverse implementazioni di container, la classe string, iteratori, e oltre 70 algoritmi per lavorare su tali strutture. Inoltre, sono disponibili i (multi)set e le (multi)map che sono array associativi. Un STL container è una classe di oggetti che è preposta al contenimento di altri oggetti. Questi oggetti usualmente possono essere di qualsiasi tipo, e possono anche essere a loro volta dei container. I principali container che hanno riguardato la libreria per la

creazione delle strutture dati adibite al collegamento logico tra i `Signal`, sono il `List`, il `Vector` e l'`Array`. La differenza tra questi, trasparente al programmatore, è però abbastanza marcata per ciò che riguarda la gestione delle allocazioni degli oggetti in memoria.
Di seguito vengono presentate le caratteristiche prestazionali dei 3 STL Container utilizzati dalla libreria Spira e durante l'evoluzione di Embed Reactive. In questa analisi viene presa in considerazione anche Spira perchè costituisce il modello sul quale è ispirata la libreria realizzata. Questo è anche il motivo per il quale, come si vedrà nel capitolo successivo, nelle valutazioni sulle prestazioni è stato effettuato un confronto tra Spira ed Embed Reactive.

- L'oggetto List, utilizzato come struttura dati principale nella libreria Spira, gestisce la creazione e la gestione di una lista. Come si sa, per accedere ad un nodo all'interno della lista, bisogna scorrerla tutta in modo sequenziale fino ad arrivare al nodo cercato. Non esiste quindi possibilità di accedere in modo random ad un elemento. Questo però non è visibile al programmatore, che semplicemente invoca il metodo `at(index)` e ottiene l'elemento in quella data posizione. E' facile intuire che in una simile struttura dati i nodi vengono allocati dinamicamente in memoria in modo sparso, senza seguire una sequenza e andando a occupare le prime locazioni di memoria disponibili. Con questo container quindi si ha una memoria frammentata, allocata dinamicamente con degli overhead causati durante le operazioni di allocazione di memoria. Non è performante quindi per ciò che riguarda l'accesso randomico ad un elemento, mentre è ottima se si vuole dinamicità nell'aggiungere un numero non preventivamente conosciuto di elementi.

- L'oggetto Vector, è il container che più somiglia ad un normale array per ciò che riguarda la struttura fisica. Infatti esso non ha nodi collegati l'un l'altro come la lista ma ha elementi fisicamente disposti in maniera consecutiva in memoria. Anche questo oggetto come la lista, permette di allocare un numero non precisato di elementi in modo dinamico, e come nel precedente caso l'allocatore causa un certo overhead di memoria per salvare alcune informazioni di servizio. A differenza dell'oggetto list però necessita di avere un certo blocco di spazio consecutivo nella RAM, il che rende complicata la gestione di inserimento di nuovi elementi, nel momento in cui si esauriscono le locazioni adiacenti tra loro. Per ovviare a ciò, l'allocatore gestisce la memoria in maniera incrementale e preventiva. Al momento della dichiarazione dell'oggetto vector infatti, l'allocatore destina un certo numero di locazioni di memoria al vettore, anche se questo è ancora vuoto. Arrivati ad una certa soglia di riempimento l'allocatore entra di nuovo in azione, il vettore nel caso non vi siano più aree di memoria consecutive a sufficienza, viene spostato interamente in una nuova porzione in grado di contenere se stesso e i nuovi elementi che saranno aggiunti. E' evidente quindi di come a

volte l'operazione di creazione di un nuovo elemento del vettore, sia onerosa in termini di effort temporale. Ha comunque il vantaggio di poter accedere immediatamente ad un elemento con accesso randomico risparmiando la ricerca sequenziale. Perciò, un suo possibile utilizzo nella libreria, ne potrebbe migliorare i tempi di esecuzione delle propagazioni, visto che nelle letture effettuate da nodi differenti su una struttura condivisa, può essere necessario eseguire accessi random al vettore (come si vedrà appena più avanti).

- L'Array è una semplice classe che gestisce un normale array allocato in modo statico nella maniera più standard. E' infatti richiesto sapere preventivamente la dimensione della quale si vuole dichiarare l'array, che una volta fissata, tramite una costante, non può più essere aumentata dinamicamente. La classe oltre a contenere la struttura dati, offre dei metodi di interfaccia per accedere in modo semplice agli elementi dell'array e alcuni attributi utili come ad esempio la lunghezza dell'array. In questo STL container la memoria viene allocata in modo statico al termine della compilazione, questo garantisce di avere un'area di memoria non frammentata e senza overhead di vario tipo. I punti di forza sono i medesimi del Vector, con il vantaggio di non avere tempi di inserimento allungati dalla riallocazione dell'intero vettore, e di avere all'aumentare degli elementi un risparmio di memoria rispetto al Vector e a List, visto che l'allocazione avviene in modo statico senza overhead e visto il contenuto consumo di base di memoria da parte della classe Array rispetto alle altre due che devono gestire più informazioni per la gestione dell'allocazione dinamica in memoria.

Le dimensioni di memoria occupata dai vari container differiscono in base alla libreria utilizzata per lo standard C++11. La libreria GNU di default utilizzata per C++11 è la libstdc++ però non mancano alternative valide come ad esempio libc++ implementata sotto licenzia MIT. Di seguito le prestazioni a confronto di list e vector per ciò che il guarda il tempo di esecuzione di una ricerca di un elemento e di un inserimento random secondo un esperimento effettuato sugli STL[18]. Il test è stato effettuato su una macchina dotata di un Intel Core i7, e realizzato con codice utilizzante lo standard C++11.

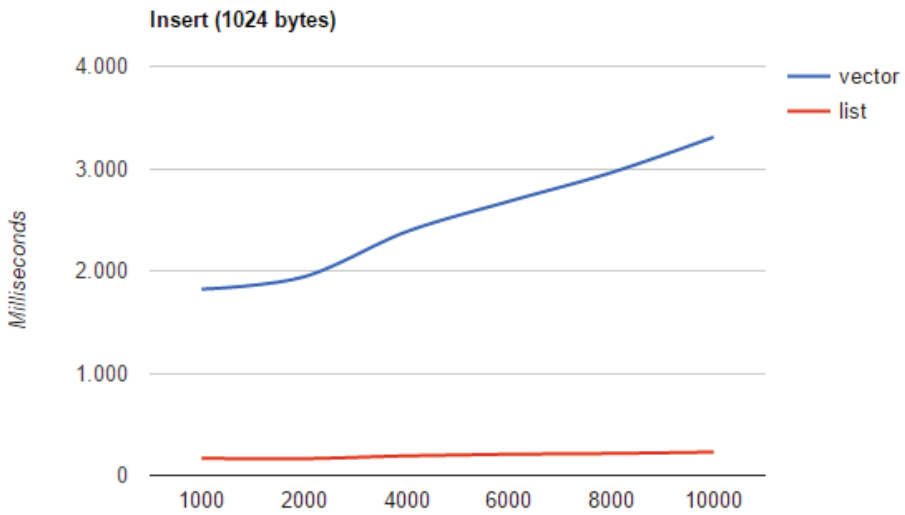

Figura 4.3: Prestazioni a confronto per inserimento random.

Nel grafico in figura 4.3 è evidente come il Vector sia più veloce rispetto a List nell'inserimento di un elemento in una data posizione all'interno della struttura dati. Questo perché come detto, il Vector è più performante rispetto a List nell'accesso casuale alla struttura dati, mentre con il tipo List, avendo elementi non contigui in memoria, si necessita una scansione sequenziale di tutta al lista fino alla posizione specificata nell'inserimento.

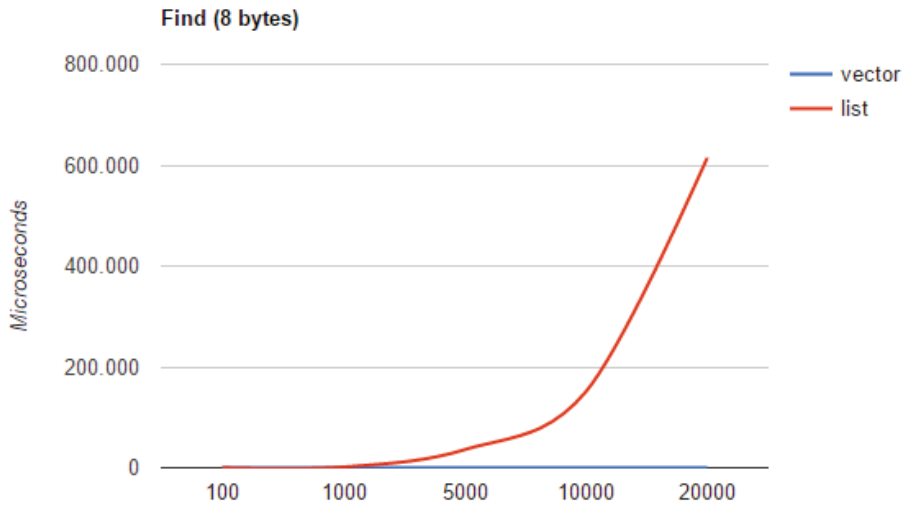

Figura 4.4: Prestazioni a confronto per la ricerca.

Quanto appena detto è confermato anche nel grafico in figura 4.4 dove è presente un confronto tra Vector e List per quanto riguarda il tempo di esecuzione della ricerca di un elemento all'interno della struttura dati. Nel Vector infatti è sempre tendente allo 0 dato che l'accesso avviene tramite indice in maniera diretta.

Questi risultati torneranno poi utili per comprendere le differenze nei tempi di esecuzione delle propagazioni tra Spira ed Embed Reactive, mostrate nel capitolo sulle valutazioni.

4.3.2 Struttura Dati con Vector

Inizialmente la libreria implementa esattamente il modello di grafo che oltre ad essere un modello logico, qui è anche un modello con cui sono organizzati gli oggetti tra di loro. Infatti ogni nodo presenta un vettore di nodi collegati tra di loro nel grafo, oltre alla variabile che contiene il value e un numero intero per il rate.

Il vettore di nodi collegati tra di loro è implementato con l'STL container di STD chiamato `Vector`. Il grafo risulta quindi di rapida costruzione in termini di tempo di esecuzione dato che, ad ogni aggiunta di nodo successivo, viene fatto un inserimento in coda al vettore di puntatori dell'oggetto a cui lo collego, senza altre particolari elaborazioni.

Con questa versione di struttura dati sono stati fatti i primi test di performance che, confrontati con i risultati ottenuti dagli stessi test effettuati sulla libreria Spira (che utilizza List), hanno evidenziato un sostanziale guadagno per ciò che riguarda i tempi di esecuzione, e un attenuato guadagno in termini di occupazione di RAM. Il guadagno nei tempi di esecuzione della propagazione è spiegato da ciò che si è detto prima sulle differenze tra List e Vector, come mostrato in figure 4.3 e 4.4.

Inizialmente Embed Reactive è stata realizzata rispettando i soli requisiti funzionali. Dopo un lavoro di ricerca e ottimizzazione, è stato messo a punto un miglioramento della struttura dati che ha permesso di rispettare i criteri di progetto riguardanti il risparmio di memoria. L'approfondimento sugli STL Container è stato fatto perché saranno infatti questi i protagonisti dell'evoluzione della struttura dati di Embed Reactive, visto che List è utilizzato in Spira, Vector è stato utilizzato inizialmente in Embed Reactive e Array è quello definitivamente utilizzato in Embed Reactive, come spiegato di seguito.

4.3.3 Evoluzione della Struttura Dati

Nel miglioramento della libreria effettuato, sono presi in considerazione gli elementi della prima versione di che portavano ad un utilizzo più marcato della memoria. Il vettore dei puntatori alle push dei nodi successivi presente in ogni nodo `Signal`, è stato perciò identificato essere l'elemento più dispendioso. Questo perché si tratta di una struttura dati di tipo function pointer dinamica, che cresce al crescere del numero degli archi uscenti dal nodo possessore della lista. Inoltre dalle caratteristiche degli STL container [4], valide se si utilizza libstdc++ come in questo caso, si sa che:

- Un oggetto di tipo `List` occupa 16 Byte per la sua inizializzazione.
- Per ogni elemento occupa 4 Byte (la dimensione del tipo di elemento puntatore) + 16 Byte (puntatore a nodi successivi e altro overhead utilizzato dalla classe list per ogni suo nodo).
- Quindi per ogni nodo si ha una lista di dimensioni in Byte = 16 + |N| * (16 + 4), dove N è l'insieme degli elementi contenuti nella lista.

Mentre se si utilizzasse invece un tipo `Vector`:

- Un oggetto di tipo Vector occupa 24 Byte per la sua inizializzazione.
- Per ogni elemento occupa 4 Byte (la dimensione del tipo di elemento puntatore).

- Quindi si ha un vettore di dimensione in Byte = 24 + |N| * 4, dove N è l'insieme degli elementi contenuti nel vettore.

Dato che l'accesso alla struttura dati è sequenziale sia in lettura che in scrittura, utilizzando il `Vector` non si va a peggiorare il tempo di esecuzione rispetto a quello che si ha con il tipo `List`. Anche perchè il tipo `Vector` utilizza la memoria in modo contiguo (non sparso come il tipo `List`), l'ideale per operazioni di lettura sequenziale. Si è allora utilizzato in prima istanza il tipo Vector per implementare le strutture dati di Embed Reactive, utilizzando quindi un'allocazione dinamica della memoria. Per semplicità questa versione della libreria sarà chiamata **Dynamic Embed Reactive**.

E' evidente che utilizzando un tipo Vector al posto del List si ha un buon risparmio di memoria soprattutto con l'aumentare degli elementi del vettore. D'altra parte rimangono 24 Byte fissi per la dichiarazione, che sono occupati quindi ogni volta che creo un nuovo oggetto signal, il che non è ottimale. Se per esempio dichiarassi 10 signal senza collegarli tra loro con archi di propagazione, occuperei 240 Byte di RAM senza aver neanche aggiunto elementi ai vettori.

Volendo quindi ottimizzare ulteriormente l'occupazione di memoria, si è pensato inizialmente di ricorrere ad algoritmi di compressione dei grafi, per riuscire magari ad occupare meno RAM per la costruzione della struttura. Un qualunque algoritmo di compressione applicabile però, prevedeva la modifica della rappresentazione del grafo, con l'accorpamento dei nodi simili. Dato che ogni nodo ha una funzione essenziale nella lettura e propagazione del segnale, il grafo che viene creato dal programmatore non si può ottimizzare accorpando nodi. Allora si è pensato di abbandonare le strutture dati che nel loro insieme andavano a formare un grafo, e di separare quindi quella che è la struttura logica del grafo da quella che è la struttura fisica: dato che un grosso peso nella occupazione della RAM lo dà la dichiarazione del tipo `Vector`, al posto che utilizzare un vettore per ogni nodo, si può risparmiare memoria, utilizzando un unico vettore globale in comune a tutti i nodi. Così facendo si avrebbe un sostanziale risparmio di memoria tanto più grande quanto più alto è il numero dei nodi.

4.3.4 Vettore Globale dei Puntatori

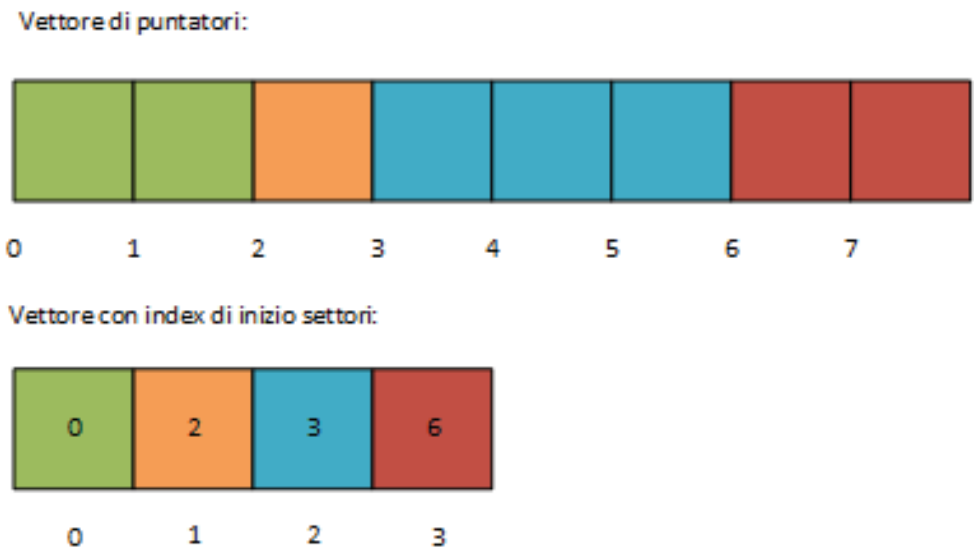

Figura 4.5: Vettori per la gestione dei puntatori alle funzioni.

Il vettore globale dei puntatori, contiene al suo interno una porzione contigua di elementi per ogni nodo creato. Tale porzione ha una dimensione specificata all'interno dell'attributo `dim` della classe `Signal`. Vi è però il problema di dover assegnare una porzione del vettore globale ad ogni oggetto Signal, così che ognuno di questi oggetti possa avere un certo numero di elementi del vettore dedicati su cui effettuare delle letture e scritture. A tal fine è necessario quindi un altro vettore, in questo caso di interi, che memorizza gli indici di inizio di ogni porzione di vettore globale di puntatori. In figura 4.5 sono mostrati i due vettori in questione. Gli elementi dei vettori dello stesso colore riguardano uno stesso nodo.

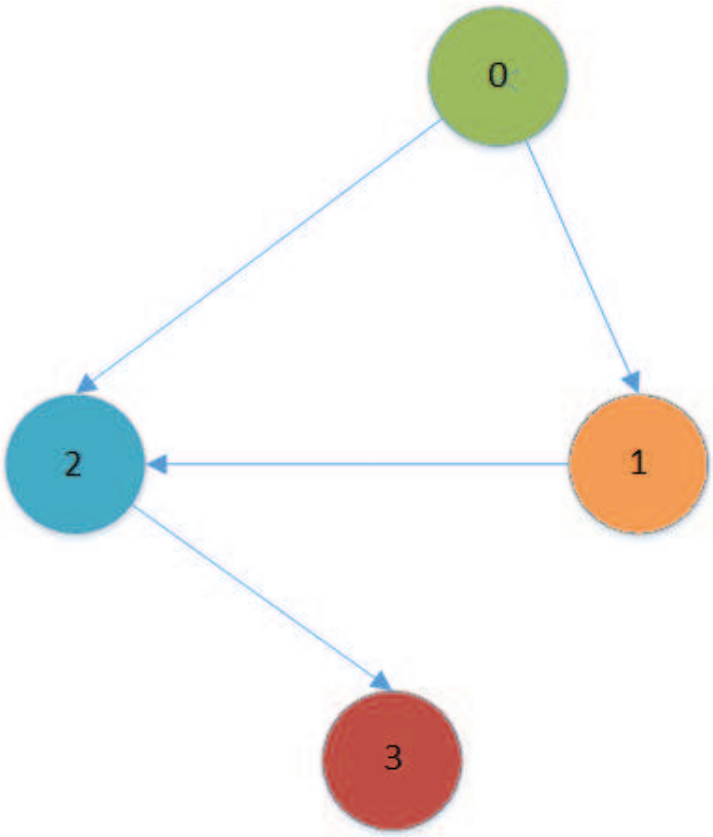

Figura 4.6: Grafo corrispondente al vettore.

Il vettore dei puntatori è da intendersi come una linearizzazione di un ipotetico grafo in cui ogni elemento è un puntatore ad una funzione che a sua volta può invocare un'altra funzione tramite il suo puntatore e così via. Il grafo in figura 4.6 corrisponde ad esempio al vettore di puntatori in figura 4.7.

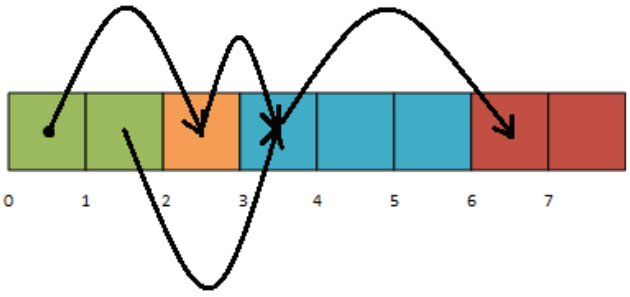

Figura 4.7: Link nel vettore.

Come si nota in figura 4.5, il vettore che indicizza il vettore di puntatori contiene la posizione di inizio del settore assegnato ai `signal` che sono stati inizializzati. La posizione di ogni elemento di questo secondo vettore è corrispondente al valore dell'attributo `id` contenuto nell'oggetto signal proprietario del settore indicizzato dall'elemento stesso. Nell'esempio l'oggetto signal rappresentato dal colore azzurro ad esempio, ha come id 2 e possiede 3 puntatori a nodi successivi memorizzati in modo contiguo a partire dalla posizione 3.

Questa soluzione propone un accesso casuale (non sequenziale) al vettore sia per le letture che per le scritture. Inoltre ad ogni scrittura, e quindi ad ogni aumento di dimensione di un settore, corrisponde lo shift verso destra dei settori successivi a quello che si sta ampliando, questo per mantenere contigui gli elementi del vettore accessibili dallo stesso nodo. Ad esempio se collegassi un nuovo nodo al primo di quelli presenti nell'esempio mostrato in figura 4.5, andrebbe aggiunto al primo vettore un nuovo elemento di colore verde che andrebbe a posizionarsi in posizione 2, shiftando a destra tutti gli altri elementi già presenti (da 2 a 7).
Tutte queste operazioni sono gestite dalla classe `Connector`, presentata poco più avanti.

Il fatto di avere accessi casuali per le letture non implica significativi ritardi dato che si sta usando un vettore e non una lista. Per la scrittura invece si ha una complessità O(n), dato che nel peggiore dei casi, in cui bisogna aggiungere un elemento in posizione 0, si deve scandire per intero il vettore per shiftare gli elementi già presenti. Dato però che la generazione delle dipendenze viene eseguita solamente una volta nel programma, le scritture degli elementi per la generazione della struttura dati hanno un costo in termini di tempo, che ci si può permettere. Nell'immagine 4.7 si nota che gli elementi del vettore appartenenti a oggetti differenti, possono essere collegati tra di loro tramite archi che rappresentano flussi di invocazioni in serie a funzioni 'push', e possono essere equiparati a degli archi che collegano i nodi in un grafo.

Come detto precedentemente il tipo `Vector` come anche il tipo `List`, sono considerate strutture dati dinamiche che, oltre a risiedere interamente nell'area di memoria dedicata allo heap, utilizzano, all'interno della loro classe nell' STD, un oggetto allocatore che ha il compito di riservare una porzione di memoria per ogni oggetto aggiunto. Questo utilizza diciture 'new' oppure 'malloc' che vanno ad aggiungere dell'overhead per ogni blocco di memoria allocato.

Utilizzando invece una struttura dati statica come l'`Array`, si ha l'allocazione della memoria necessaria una sola volta, e si utilizza l'area di memoria RAM adibita a contenere le variabili statiche allocate all'avvio del processo. Questo fa sì di ottenere un risparmio notevole in overhead di memoria per ogni elemento. Vi è stato quindi il passaggio all'utilizzo della memoria allocata in modo statico

utilizzando il tipo `Array`. Questa versione della libreria sarà chiamata da qui in avanti **Static Embed Reactive**.

Sintetizzando tutte le caratteristiche,anche di impatto secondario, di Static Embed Reactive che interessano il risparmio di memoria sono:

- Utilizzo di tipi interi di 8 o 16 bit presi dalla libreria stdint.h invece che ti interi 32 bit.
- Nessuna lista o vettore utilizzati nel singolo oggetto, ma due array statici globali. Come detto il vantaggio di utilizzare una struttura statica è quello di non avere overhead di memoria per ogni elemento aggiunto e di non avere la memoria frammentata. In questo caso ogni elemento puntatore occuperà 4 Byte mentre ogni elemento indice dell'altro array occuperà 1 Byte.

Ipotizzando quindi di avere un albero binario bilanciato con 63 nodi (livello di profondità 5), avrei un'occupazione di memoria dedicata ai 2 array più variabili di: (numero di nodi)63 * 1 + (numero di archi)31 * 4 + (memoria occupata da variabili intere per nodo)24 * 63 = 1512 Byte.
Il valore risulta già essere inferiore rispetto a quello che si ha utilizzando Dynamic Embed Reactive che utilizza un vettore di puntatori di tipo Vector non globale, bensì presente in ogni nodo, che invece occupa a parità di esempio circa il doppio: (dimensione base del Vector vuoto)24 * 63 + (dimensione del singolo elemento)4 * (numero di archi)31 + (memoria occupata da variabili intere per nodo)24 * 63 = 3148 Byte.

4.4 Gestione della Struttura Dati

La gestione dei due array statici è affidata totalmente alla classe `Connector`, che rende trasparenti al `Signal` le operazioni sulla struttura dati. Come detto in precedenza, avendo avuto la necessità di identificare in modo univoco i nodi per poterli riconoscere all'interno del grafo e poterli indicizzare nella struttura dati, è stato necessario l'inserimento dell'attributo `id` all'interno di ogni oggetto Signal. Dell'assegnamento dell'id ad ogni nodo, se ne occupa la classe Connector tramite il suo attributo `counter`.

Oltre al counter in questa classe sono presenti i 2 vettori globali.

```
1 std::array<std::function<void(T)>, V> linkVect;
2 std::array<uint8t, N> starts;
```

Listing 4.1: Dichiarazione dei due vettori

Il primo è un vettore di puntatori a funzioni, ed è utilizzato per salvare i puntatori alle funzioni push dei vari oggetti. Il secondo è il vettore che indicizza il primo.

Infine, oltre ai due vettori globali, vi è un vettore di booleani aggiuntivo, dichiarato solamente con la dichiarazione della costante che attiva la *Push condizionata*.

La dichiarazione dell'oggetto di tipo `Connector` è essenziale per l'utilizzo della libreria, e deve trovarsi all'inizio del codice utilizzante Embed Reactive. L'oggetto `Connector` deve essere configurato e dimensionato dal programmatore per poter creare e collegare correttamente i nodi. Entrambi i vettori contenuti nella classe `Connector` sono dimensionati al momento della loro dichiarazione tramite le costanti V ed N, e sono gestiti per rappresentare un grafo di N nodi e di V archi direzionati. Questi valori li deve fornire il programmatore al momento della dichiarazione dell'oggetto connector. Ciò implica che la struttura che si vuole andare a creare con i vari `Signal` deve essere nota a priori al programmatore. Sicuramente questo tipo di soluzione causa una sostanziale perdita di flessibilità dal punto di vista della creazione delle dipendenze tra oggetti reattivi, è però il prezzo da pagare per poter salvare memoria RAM. Una soluzione di questo tipo che limita la flessibilità in favore del risparmio di memoria è tra l'altro frequente nei sistemi embedded.

Capitolo 5

Valutazione

Di seguito vengono mostrati i risultati di test effettuati con le due versioni della libreria confrontate con la libreria Spira. La scelta di una libreria già esistente da analizzare è ricaduta su Spira perché, pur non essendo una libreria concepita per la programmazione embedded, è la più performante, tra quelle esistenti che possono essere ritenute di tipologia push-based, in termini di consumo di memoria. Questo perché presenta un codice molto semplice implementando un'idea di gestione di valori e flussi di propagazione che è stata presa da esempio per realizzare Embed Reactive.

Visto che come detto la propagazione di valori nella programmazione reattiva può essere sintetizzata con un grafo, sono stati creati dei grafi di test ad-hoc che, compatibilmente con le funzionalità di Spira, hanno portato a ricavare dei grafici di performance in funzione di un parametro misurabile (es. il numero di nodi creati oppure il livello di profondità di un albero binario bilanciato).

Le misurazioni effettuate riguardano quindi:

- Il tempo impiegato a propagare un valore lungo tutta la struttura creata (espresso in micro secondi).
- La quantità di memoria RAM occupata dai vari oggetti creati tramite le librerie.

Per l'esecuzione dei test sono state utilizzate delle piattaforme hardware descritte di seguito.

5.1 Piattaforme

Le piattaforme hardware sono utilizzate oltre che per la realizzazione del codice e la sua esecuzione per scopi di debugging, anche per l'effettuazione di tutti i test prestazionali.
Le piattaforme di test utilizzate sono le board di sviluppo STM32Nucleo. Questa famiglia di board comprende schede che spaziano dalla serie L0 a bassissimo

consumo fino alla serie F7 ad alte prestazioni.

Per avere un ambiente di test con condizioni che fossero restrittive, sono state scelte le board STM32Nucleo a bassissimo consumo L0 e L1. Questo perché presentano caratteristiche identiche a quelle delle schede applicate ai sistemi embedded a risorse limitate. Questi sistemi hanno come priorità il risparmio energetico, a maggior ragione nei casi in cui sono alimentati da una batteria. Queste board sono accomunate dalla presenza di un microprocessore cortex rispettivamente M0 e M3 con architettura ARMv7, e da basse quantità di memoria RAM e flash.

Le due schede di sviluppo sopra citate hanno le seguenti caratteristiche [5]:
Per quanto riguarda la STM32 Nucleo L1 si ha:

- Processore cortex-M3 32 Mhz
- 512 KB di memoria flash
- 80 KB di RAM

Mentre per la STM32 Nucleo L0 si ha:

- Processore cortex-M0 48 Mhz
- 256 KB di memoria flash
- 32 KB di memoria RAM

Per la creazione di un progetto con il firmware della scheda è stato utilizzato Coocox IDE, dato che permette una più rapida inclusione di terze parti nel progetto e un facile bebugging del codice, rispetto ad altri tool online. Per cross compilare è stato utilizzato il toolchain GNU ARM, integrabile direttamente dentro l'IDE.

Il firmware utilizzato per la scheda inizializza gli oggetti interfaccia per accedere alle periferiche della scheda, come ad esempio il timer fisico e la porta seriale. L'interfaccia alla porta seriale serve in questo caso per stampare a video i risultati dei test.

5.2 Metriche

In questa sezione vengono presentate le modalità di misurazione della memoria occupata e del tempo di propagazione di un dato all'interno del grafo delle dipendenze. Inoltre vengono presentate le principali problematiche risolte per raggiungere l'obbiettivo di misurazione.

5.2.1 Tempo di Propagazione

Questo tipo di test misura il tempo che intercorre tra le push nei nodi di input della struttura creata, e la propagazione del segnale, fino a coprire tutto il grafo, in tutti gli oggetti connessi tra di loro. Per poter misurare è stato necessario un oggetto dichiarato in timer.h che, interfacciandosi con le periferiche della scheda, utilizza un timer fisico presente su di essa, con una risoluzione di 16 bit ed un clock timer massimo a 32 Mhz. Quindi nello scrivere il codice si è avviato il timer prima di chiamare le push, mentre si è stampato a video il valore in micro secondi contato subito dopo la fine dell'esecuzione delle push come mostrato nell'esempio di codice 5.1.

```
1   timer.reset();
2   timer.start();
3   root.push(1);
4   pc.printf("tempo impiegato: %d   ", timer.read_us());
```

Listing 5.1: Esempio di misurazione del tempo

5.2.2 Memoria Occupata

Per poter misurare la RAM libera (e quindi quella allocata tramite la differenza tra la RAM totale e quella libera) è stato necessario procedere considerando come il C++ gestisce la memoria e in base a questo, costruendo un algoritmo che andasse a contare quanti Byte fossero rimasti liberi dopo l'allocazione di tutto ciò che era necessario per le variabili e gli oggetti creati con la libreria. Metodo sicuramente non con una precisione al Byte, ma che può essere ritenuto affidabile. Un'alternativa per misurare la memoria occupata, era l'utilizzo di un debugger fisico da collegare alla board. Questo metodo risulta più affidabile del precedente ma non praticabile in assenza delle componenti hardware necessarie.

Benché lo spazio di indirizzi virtuali copra un intervallo molto ampio, solo una parte di essi è effettivamente allocato ed utilizzabile da un programma. È pertanto importante capire come viene strutturata la memoria virtuale di un programma C++.
Essa viene divisa in segmenti, cioè un insieme contiguo di indirizzi virtuali ai quali il programma può accedere.

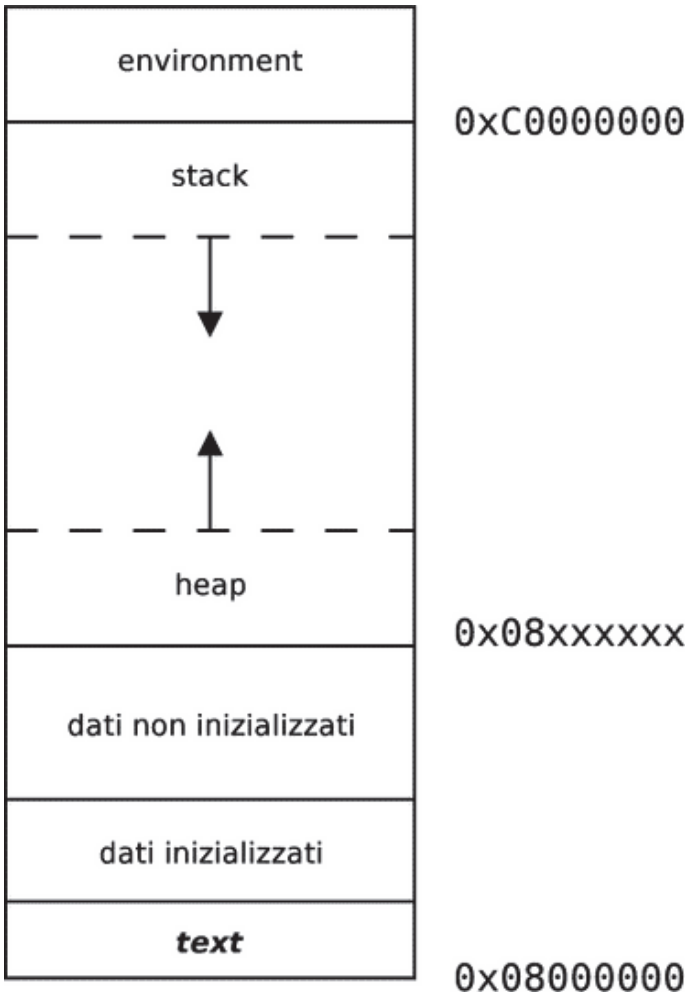

Figura 5.1: Organizzazione della memoria in C++.

Solitamente un programma C++ viene suddiviso nei segmenti mostrati in figura 5.1:

- Il segmento di testo o text segment. Contiene il codice del programma, delle funzioni di librerie da esso utilizzate, e le costanti. Nelle board utilizzate risiede nella memoria Flash data la limitatezza della memoria RAM.
- Il segmento dei dati o data segment. Contiene le variabili inizializzate, sia globali che statiche.
- La parte dei dati non inizializzati chiamata anche BSS (block started by symbol), che contiene le variabili allocate staticamente, il cui valore non è stato assegnato esplicitamente. Ad esempio dichiarando un vettore.

- Lo heap. Tecnicamente lo si può considerare l'estensione del segmento dati, a cui di solito è posto giusto di seguito. È qui che avviene l'allocazione dinamica della memoria.

- Il segmento di stack, che contiene lo stack del programma. Tutte le volte che si effettua una chiamata ad una funzione è qui che viene salvato l'indirizzo di ritorno e le informazioni dello stato del chiamante. Al ritorno della funzione lo spazio è automaticamente rilasciato e "ripulito", con conseguente riduzione dello stack. Il limite superiore dello stack ,nelle board utilizzate rimane fisso, rendendo la memoria sottostante adibita ad un uso esclusivo da parte dello stack. Questa però può essere vista come una pratica logicamente corretta dato che se durante l'esecuzione viene occupata quella quantità di memoria per le chiamate alle funzioni, bisogna garantire che rimanga sempre libera da altri dati appartenenti allo heap che cresce in direzione opposta.

Come si nota la parte di memoria libera si trova tra la fine dello heap e la fine dello stack. Per misurare la RAM residua quindi bisogna procedere nel contare quanti byte ci sono tra le due aree. La prima cosa che viene in mente è quella di ottenere l'indirizzo dell'ultima locazione dello heap e lo stack pointer da cui fare una sottrazione. Così facendo però non si considererebbe la frammentazione della memoria. Infatti potrebbero esserci delle locazioni di memoria vuote nel mezzo dello heap a causa di disallocazione dinamica di qualche oggetto o variabile.

Allora altra cosa che può venire in mente è quella di allocare un byte per volta fino a saturare la memoria RAM, così da contare quanti byte si è riusciti ad allocare. Per fare ciò si usa l'istruzione malloc(1), che alloca un byte e ne restituisce l'indirizzo. Stampando a video gli indirizzi di tutti i byte allocati si nota che in realtà questa allocazione dinamica non va ad occupare 1 byte alla volta, ma salva in memoria anche altre informazioni di servizio legate all'utilizzo della malloc che generano un overhead costante e periodico ogni 4 byte allocati.

Come si è notato durante i test, ogni 4 allocazioni di un byte, si hanno 42 byte occupati invece che 4. Questo overhead è stato tenuto in conto nel calcolo dei byte liberi, infatti il numero totale di allocazioni è stato diviso per 4 (dato che si ha una periodicità di 4) e moltiplicato per 42 (dimensione del blocco da 4). Inoltre la caratteristica delle board utilizzate, del mantenere fisso il limite superiore dello stack, fa sì che un'area di memoria di alcuni Byte pur essendo vuota al momento della misurazione, non può essere allocata. La dimensione di questa area di memoria è stata verificata con degli esperimenti, e poi conteggiata durante il calcolo della memoria occupata. Questo per garantire una misurazione ancora più accurata.

5.3 Baseline

Date le scelte che hanno portato ad un miglioramento prestazionale di Embed Reactive rispetto a Spira, si vuole anche mostrare ciò che quest'ultima libreria offre in termini di caratteristiche e funzionalità. Questo per fare un paragone e una valutazione su cosa si è perso o guadagnato dal punto di vista delle funzionalità rispetto ad Embed Reactive. Di seguito le caratteristiche di Spira, considerando che per tutte le scelte implementative fatte è valso il concetto che questa è una libreria non progettata prettamente per sistemi embedded, ma supporta in modo generalista un qualsiasi stile di programmazione reattiva per qualsiasi piattaforma. Per questo motivo non vengono risparmiate strutture dati, con il fine di rendere il più potente possibile, per quanto riguarda le funzionalità e future implementazioni, la libreria. Viene quindi differenziato ciò che è atto ad un'implementazione accessoria da ciò che è strettamente necessario per rispettare i requisiti iniziali imposti dai criteri di progetto.

Strutture dati essenziali:

- Per ogni oggetto creato è presente una lista di puntatori a funzioni:
  - **lista HOOK**: questa contiene solo i puntatori alle funzioni di propagazione agli oggetti successivi. Viene riempita solamente tramite i costruttori al momento della creazione dell'oggetto.

Strutture dati accessorie:

- Per ogni oggetto creato sono presenti due liste di puntatori a funzioni, utilizzate per separare le tipologie di funzioni linkate:
  - **lista GLUE**: questa lista viene utilizzata per supportare l'implementazione di propagazioni parziali del dato, che ad esempio propagano una parte piuttosto che un'altra a seconda che condizioni booleane siano o meno soddisfatte. Pur essendo molto simile alla lista HOOK, è utilizzata perché nella lista GLUE vengono memorizzate funzioni push ad-hoc diverse da quelle memorizzate nella lista HOOK, proprio perché devono supportare le propagazioni di parti di dati.
  - **lista BIND**: contiene i puntatori a funzioni di side-effects, come ad esempio stampe a video, che necessitano del valore dell'oggetto già aggiornato. Per questo vengono eseguite prima le funzioni puntate in hook e successivamente queste.

Le caratteristiche definite come accessorie, sono quelle non strettamente necessarie a supportare reactive programming, oppure che si possono implementare con soluzioni alternative, risparmiando magari in strutture dati e quindi in memoria occupata.

Le 3 liste appena descritte vengono dichiarate come oggetto STL container di tipo `List`. Ciò implica che per la sola dichiarazione senza elementi si occupino 16 Byte che moltiplicati per 3 sono 48 Byte, più 20 Byte per ogni elemento delle liste: 4 Byte la dimensione del puntatore alla funzione + 16 Byte occupati in più dal nodo della lista, per informazioni e puntatori utilizzati dalla classe List. Ipotizzando quindi di avere un albero binario bilanciato con 63 nodi (livello di profondità 6), avrei un'occupazione di memoria solo dedicata alle 3 liste di: (numero di nodi)63 * 48 + (numero di archi)31 * 20 = 3644 Byte. Detto questo è facile intuire, rapportando il valore con quelli calcolati con il medesimo esempio per Embed Reactive nella sottosezione 4.3.4 (1512 e 3148 Byte), che nei risultati si vedrà un consumo elevato di RAM da parte di Spira, o quanto meno elevato per un sistema embedded a risorse limitate.

A fronte delle funzionalità già descritte nel capitolo 2, Embed Reactive presenta sostanziali differenze rispetto a Spira per ciò che riguarda le strutture dati, la tipologia di dati utilizzata e la modalità di propagazione dei valori. Dato che il target di utilizzo della libreria è il sistema embedded a risorse limitate, il risparmio delle risorse risulta di vitale importanza. Quindi si sono dovute fare delle scelte implementative differenti, e l'effetto di ciò sarà visibile dai risultati dei test seguenti.

5.4 Risultati

Come detto sono state costruite varie strutture di grafi di dipendenze per effettuare i test. La scelta dei grafi da realizzare è ricaduta su quelle tipologie che potessero generalizzare il più possibile il comportamento della libreria, in modo tale da rendere i risultati significativi e rappresentativi di applicazioni della libreria in casi reali. Per ognuna sono state fatte un numero significativo di misurazioni. Nello specifico per i test con strutture ad albero, sono state fatte misurazioni fino alla saturazione della memoria, mentre per altre tipologie un numero prestabilito con incrementi che variano in base a ciò che il parametro in input va a dimensionare (numero di livelli dell'albero, numero di nodi, ecc..). Per il solo test che genera il grafo casuale sono state fatte 4 ripetizioni per ogni valore misurato. Questo perché, mentre i primi test sono deterministici per quanto riguarda il risultato ottenuto, con il grafo casuale si ottiene ad ogni ripetizione un risultato differente. Tra queste ripetizioni è poi stata riportata nei chart la media aritmetica.

Nello specifico sono state costruite 4 strutture:

- Un albero binario con la radice utilizzata come nodo di input.
- Un albero binario con le foglie utilizzate come nodi si input.

- Un grafo lineare di nodi contigui.
- Un grafo costruito in modo casuale.

I risultati presentati con i grafici, mostrano le curve corrispondenti alle misurazioni effettuate con Spira, Embed Reactive con struttura dati dinamica ed Embed Reactive con struttura dati statica. La libreria che utilizza le strutture dati statiche, necessita un dimensionamento degli array da effettuare prima di lanciare il test. Si dimensionano quindi gli array con una stima fatta per eccesso su quanti nodi si prevede che vengano generati durante il test, per evitare failure dovute alla mancanza di elementi negli array statici.

5.4.1 Albero Binario con Input nelle Foglie

Partendo dalle foglie, viene generato un numero di nodi che corrisponde alla seguente relazione:

$$|N| = 2^L \tag{5.1}$$

in cui N è l'insieme dei nodi ed L il numero dei livelli passato come parametro in input. Una volta generati i nodi delle foglie, questi vengono presi a due a due e collegati ad un nuovo nodo figlio tramite un merge. Il figlio sarà legato a questi da una relazione di merging: se uno dei due padri cambia il suo valore, lo propaga al nodo figlio che aggiornerà il proprio valore di conseguenza.

Come si nota si ha una crescita del numero di nodi esponenziale al crescere dei livelli dell'albero. Quindi per ogni iterazione di test effettuata si incrementerà il parametro di input di una unità fino a saturare la memoria (cosa che avviene circa all'ottavo livello dell'albero).

Nella figura 5.2 un esempio dell'albero che in questo caso è di 3 livelli di profondità.

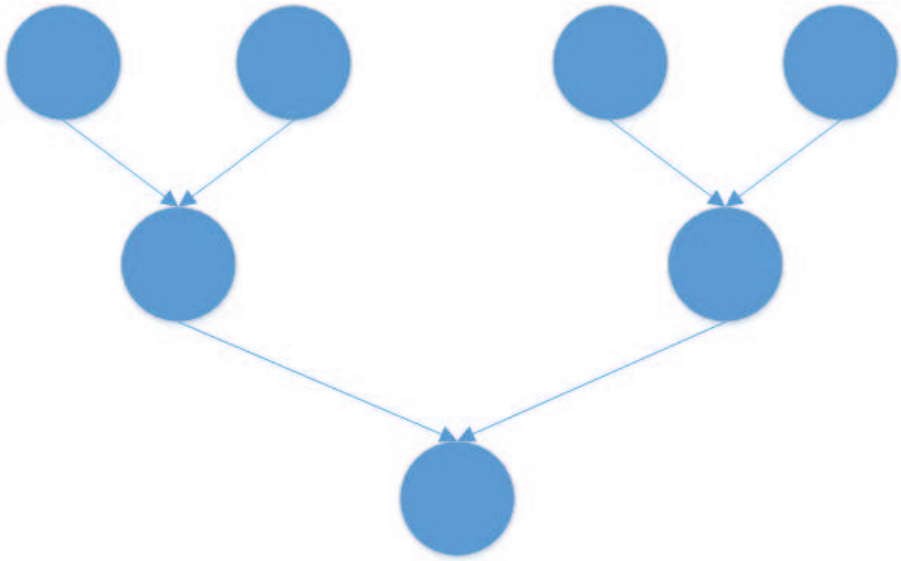

Figura 5.2: Albero binario con nodi di input nelle foglie.

Risultati con board Cortex-M3

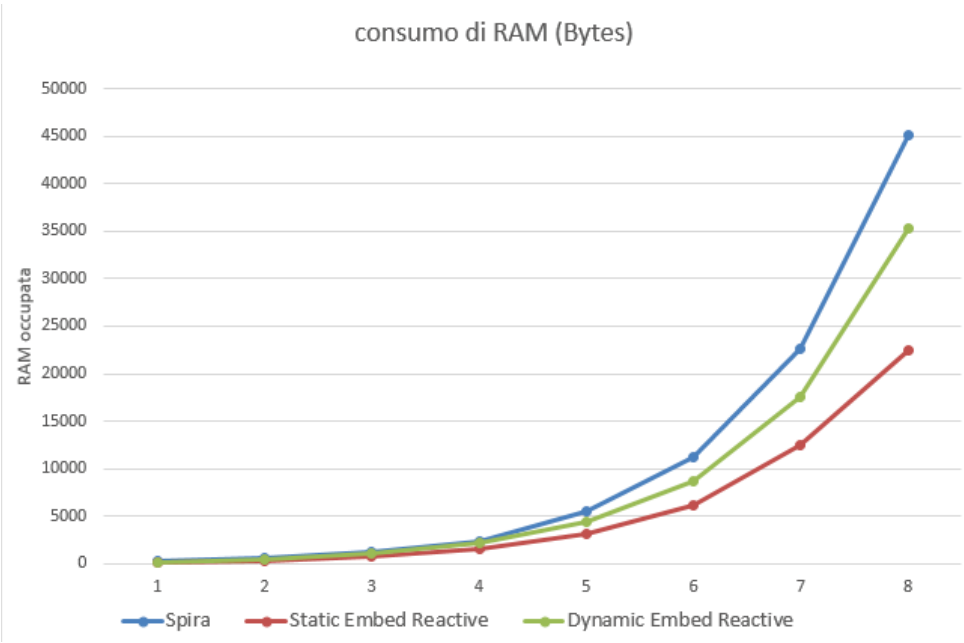

Figura 5.3: RAM occupata da un albero con nodi di input come foglie con Cortex-M3.

Il risparmio di RAM avuto con l'ultima versione della nuova libreria è mediamente del 50% rispetto a Spira e del 30% rispetto alla prima struttura dati di

Embed Reactive. Dopo l'ottavo livello di profondità dell'albero creato con Spira, la memoria della scheda è totalmente saturata, quindi non sono stati effettuati ulteriori misurazioni, anche se con Embed Reactive se ne sarebbero potute effettuare altre.
Come si vede dalla figura 5.3, l'occupazione della RAM cresce in maniera esponenziale al crescere del numero dei livelli, seguendo di fatto la crescita esponenziale del numero di nodi creati per ogni livello.

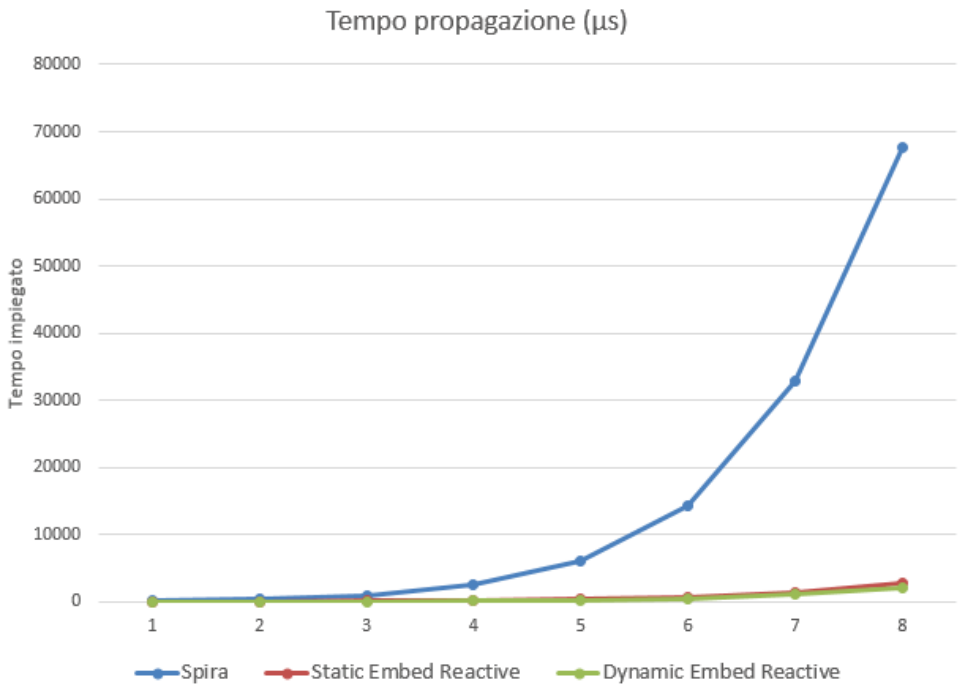

Figura 5.4: Tempo di propagazione con un albero con nodi di input come foglie con Cortex-M3.

In figura 5.4 il risparmio del tempo di esecuzione rispetto a Spira è notevole. Questo perché Spira avendo tre liste di puntatori da considerare, utilizza un algoritmo di propagazione più complesso e dispendioso rispetto ad Embed Reactive, che fa semplicemente la scansione di una lista. Questo causa un trend di crescita più rapido in Spira. Guardando i dati si nota che per ogni livello il tempo impiegato per propagare i dati, raddoppia in Embed Reactive mentre triplica in Spira.

Risultati con board con Cortex-M0

Effettuando lo stesso test con la board con Cortex-M0 sono state possibili meno misurazioni come si evince dalla figura 5.5. Questo perchè la board ha meno me-

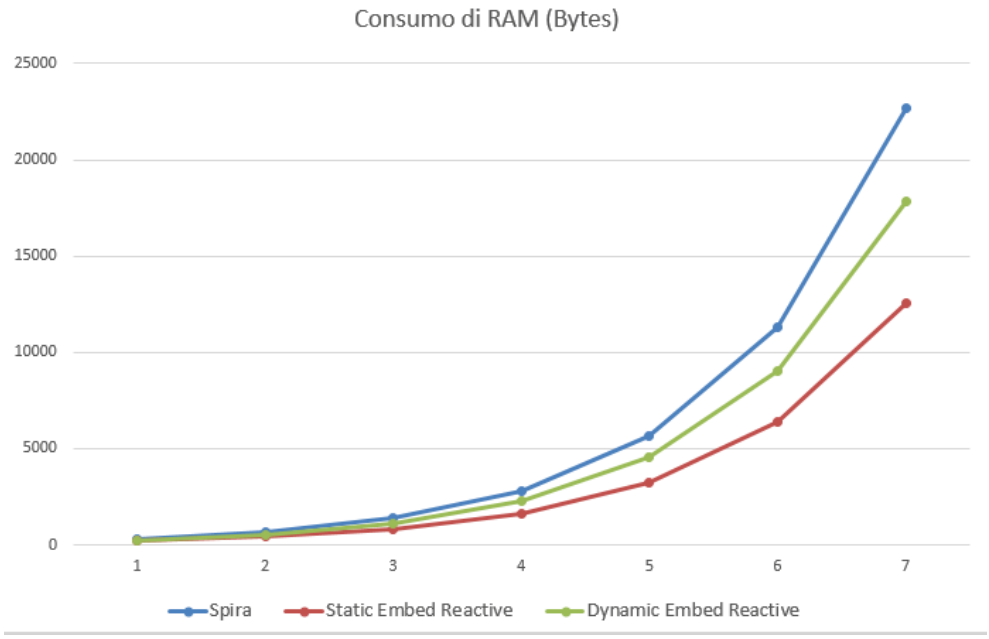

Figura 5.5: RAM occupata da un albero con nodi di input come foglie con Cortex-M0.

moria RAM, quindi si è riusciti ad arrivare massimo al settimo livello di profondità dell'albero.

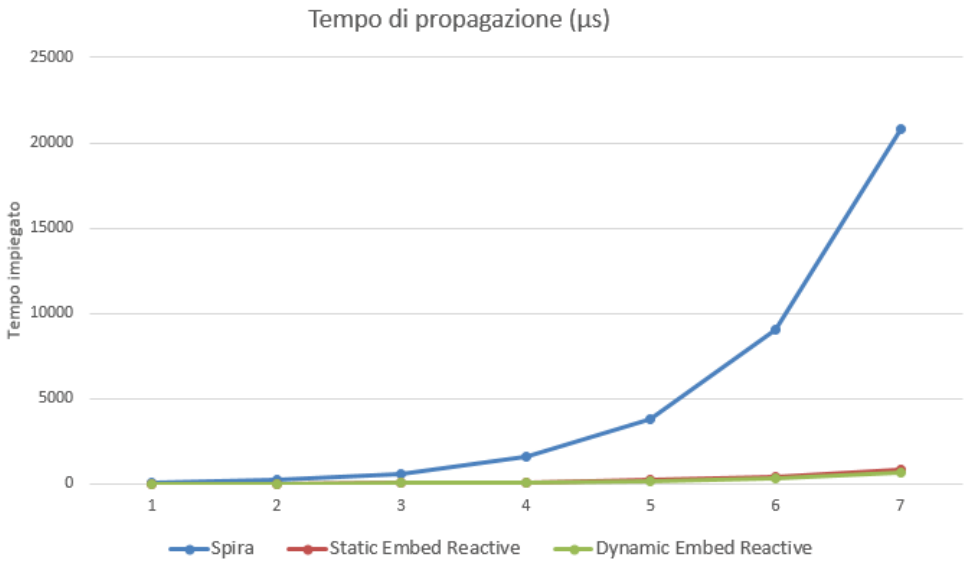

Figura 5.6: Tempo di propagazione con un albero con nodi di input come foglie con Cortex-M0.

Il tempo di esecuzione della propagazione è inferiore rispetto alla board con

Cortex-M3. Osservando la figura 5.6, il valore corrispondente al livello 7, sembra essere di quasi 10 msec inferiore rispetto a quello osservato a parità di livello in figura 5.4.
In termini percentuali si ha un risparmio di quasi il 30% del tempo di propagazione. Questo miglioramento è giustificato dal 50% in più di frequenza di clock che il microcontrollore Cortex-M0 offre rispetto al Cortex-M3. Si evince che la propagazione migliora all'aumentare della frequenza di clock.

5.4.2 Albero Binario con Input nella Radice

L'algoritmo che realizza l'albero in oggetto, considera un input, come nel caso precedente, che indica il numero di livelli da costruire. Partendo dalla radice si procede creando due figli per nodo. I figli saranno legati al padre da una relazione di semplice propagazione: al variare del valore del padre, questo viene propagato ai due figli che aggiornano il proprio con un valore uguale a quello propagato dal padre.

Anche qui si ha una crescita del numero di nodi esponenziale, quindi per ogni iterazione di test effettuata si incrementerà il parametro di input di una unità fino a saturare la memoria.

In figura 5.7 un esempio dell'albero realizzato che in questo caso riceve come input il valore 3 (3 livelli).

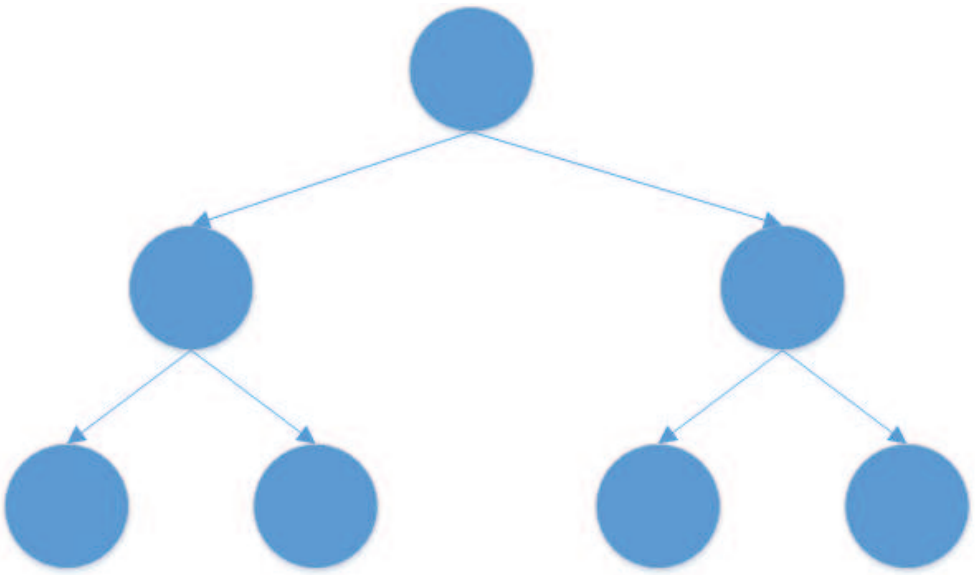

Figura 5.7: Albero binario con nodo di input nella radice con Cortex-M3.

Risultati con board con Cortex-M3

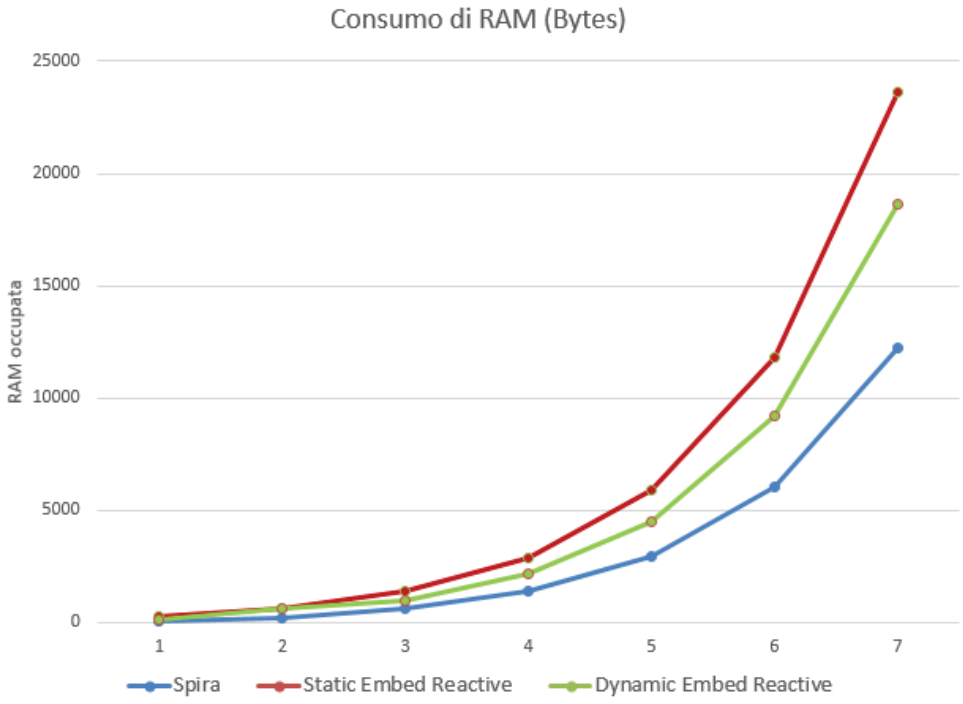

Figura 5.8: RAM occupata da un albero con nodo di input nella radice con Cortex-M3.

In questo test l'occupazione di RAM risulta leggermente superiore di pochi Byte rispetto al test precedente. Questo fa sì però che per Spira la RAM si saturi un livello prima rispetto al test precedente dove infatti al livello 8 si è avuta una quasi totale occupazione della memoria. Leggendo in ogni caso i valori nel grafico in figura 5.8, si nota a parità di livello, un risparmio di memoria di Embed Reactive rispetto a Spira. Inoltre Static Embed Reactive offre nuovamente prestazioni superiori in termini di risparmio di memoria rispetto a Dynamic Embed Reactive. Il risparmio è più marcato all'aumentare del numero di nodi e quindi del numero di allocazioni in memoria statica e dinamica.

I tempi di propagazione visti in figura 5.9 risultano quasi la metà rispetto al test precedente per quanto riguarda Spira. Questo perché ogni nodo ha due figli, quindi due elementi nella lista HOOK. Accedendo quindi una sola volta alla lista per leggere due elementi consecutivi, e risparmiando gli accessi per le liste delle foglie dell'albero che non hanno figli, si fanno circa la metà delle letture fatte nel test precedente.
Lo stesso discorso si può fare per Embed Reactive con struttura dati dinamica. Tutte le strutture dati riferite ai nodi foglia, non vengono esplorate dato che non contengono puntatori a funzioni di nodi figli, risparmiando così nel tempo di ese-

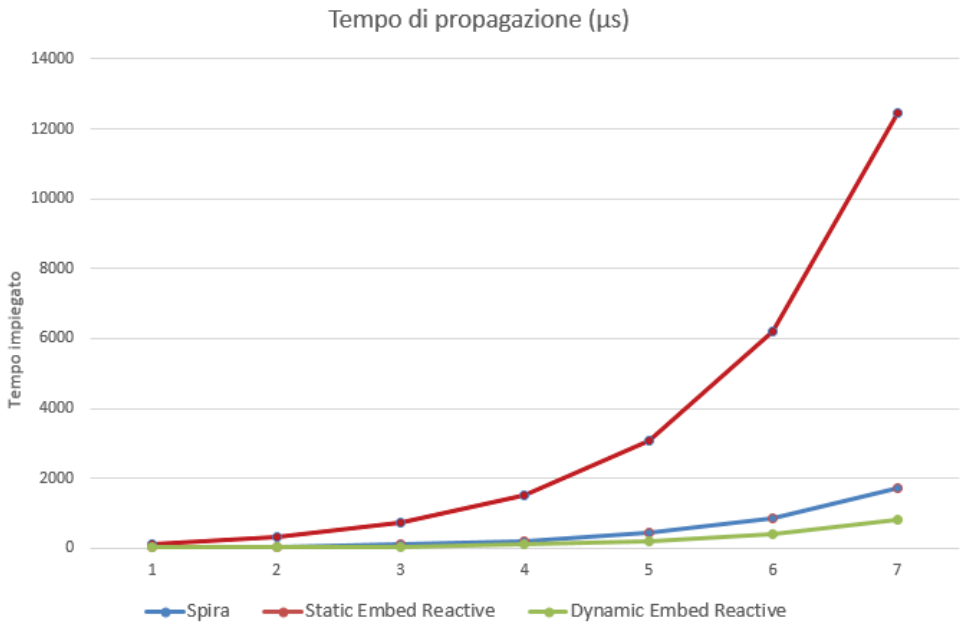

Figura 5.9: Tempo di propagazione con un albero con nodo di input nella radice.

cuzione. Per Embed Reactive con struttura dati statica invece non si riscontrano diminuzioni di tempo di propagazione significative, essendo la struttura dati statica unica per tutti i nodi, e non essendo quindi influenzata dalla presenza di numerosi nodi foglia.

Risultati con board con Cortex-M0

Sono stati effettuati i medesimi test anche con la board con Cortex-M0, e i risultati rispecchiano quanto visto precedentemente. Anche in questo caso il tempo di propagazione risulta inferiore del 50% rispetto alla board con Cortex-M3. Questo perché la lettura di questo tipo di albero con politica di esplorazione dei nodi depth-first, trova giovamento dalla frequenza più alta a cui lavora il microcontrollore Cortex-M0.

5.4.3 Grafo Lineare

L'algoritmo che realizza questo albero riceve in input il numero di nodi da inizializzare. Partendo dal primo si procede creando un figlio per iterazione. Il figlio sarà legato al padre da una relazione di semplice propagazione: al variare del valore del padre, questo viene propagato al figlio e così via.

In questo caso si ha una crescita del numero di nodi lineare, quindi come si intuisce la crescita di memoria occupata è di molto inferiore ai precedenti test, difficile infatti arrivare a saturarla in poche iterazioni. Allora per osservare

meglio l'andamento delle curve al crescere del numero di nodi, l'input è stato incrementato di 10 unità per ogni iterazione del test effettuata.

Di seguito l'esempio della struttura creata. Anche in questo caso la funzione ha come input 3 (3 nodi).

Figura 5.10: Propagazione lineare.

Risultati con board con Cortex-M3

In questa tipologia di test è interessante vedere come l'occupazione di memoria segua un andamento lineare all'aumentare del numero di nodi. Infatti l'incremento del numero dei nodi creati nel grafo è anch'esso lineare, simulando così una tipologia di applicazione di reactive programming più semplice, differente da quelle presentate in precedenza, e confermando il collegamento diretto che c'è tra il numero di nodi e la memoria occupata. Anche in questo caso il risparmio di memoria più marcato lo si ha con Embed Reactive con struttura dati statica. Infatti in figura 5.11 si notano le tre rette con pendenze differenti, e quella con coefficiente angolare inferiore corrisponde a Embed Reactive con struttura dati statica.

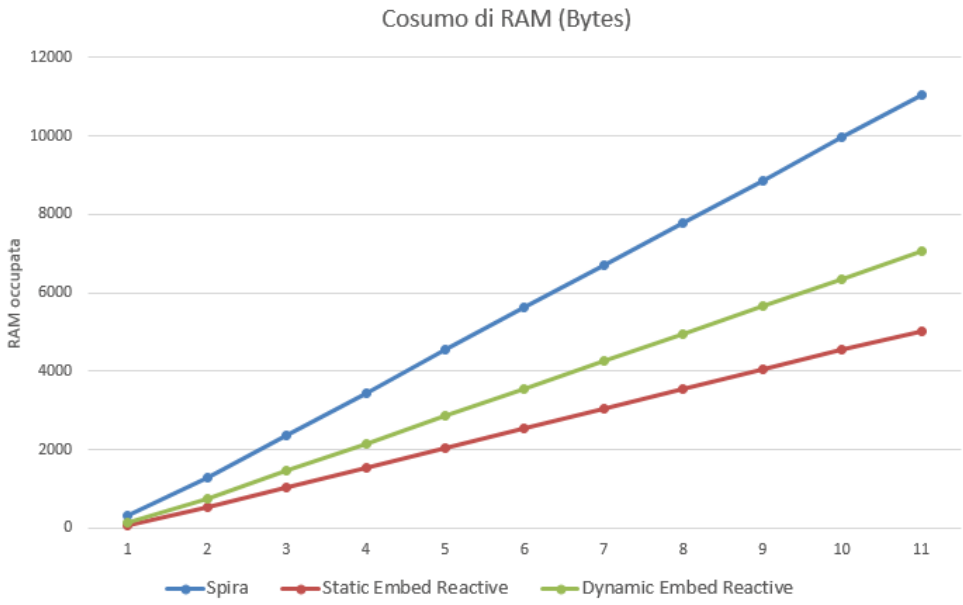

Figura 5.11: RAM occupata da un grafo lineare con Cortex-M3.

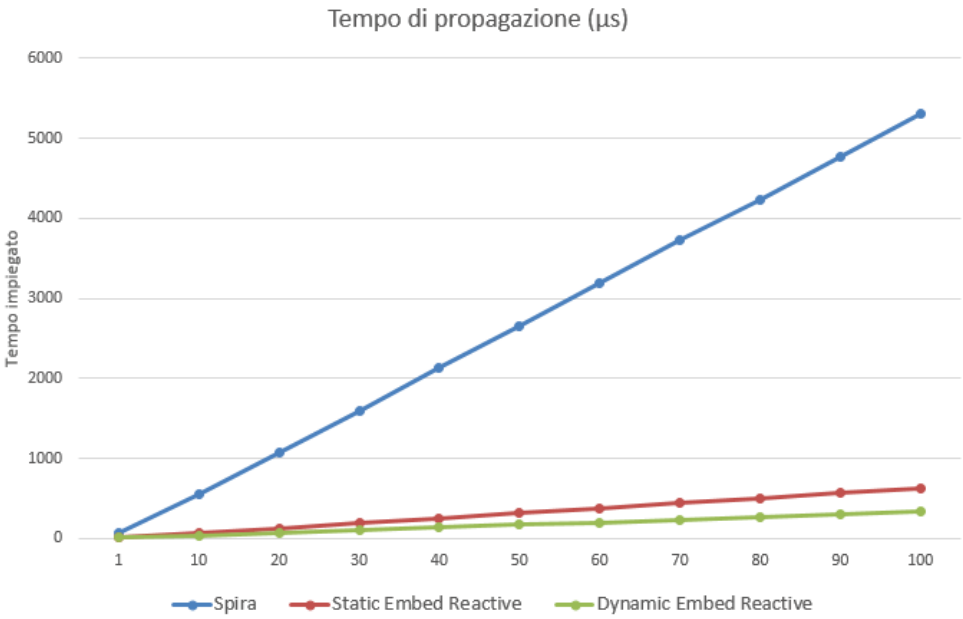

Figura 5.12: Tempo propagazione in un grafo lineare con Cortex-M3.

In figura 5.12 si nota che anche il tempo di propagazione subisce un incremento lineare all'aumentare del numero dei nodi. Quindi si evince anche una correlazione diretta tra il numero di nodi ed il tempo impiegato dagli algoritmi per propagare i dati.

Risultati con board con Cortex-M0

La stessa misurazione per l'occupazione della memoria RAM è stata effettuata con la board con microntrollore Cortex-M0, e i risultati rispettano quanto visto sin'ora.

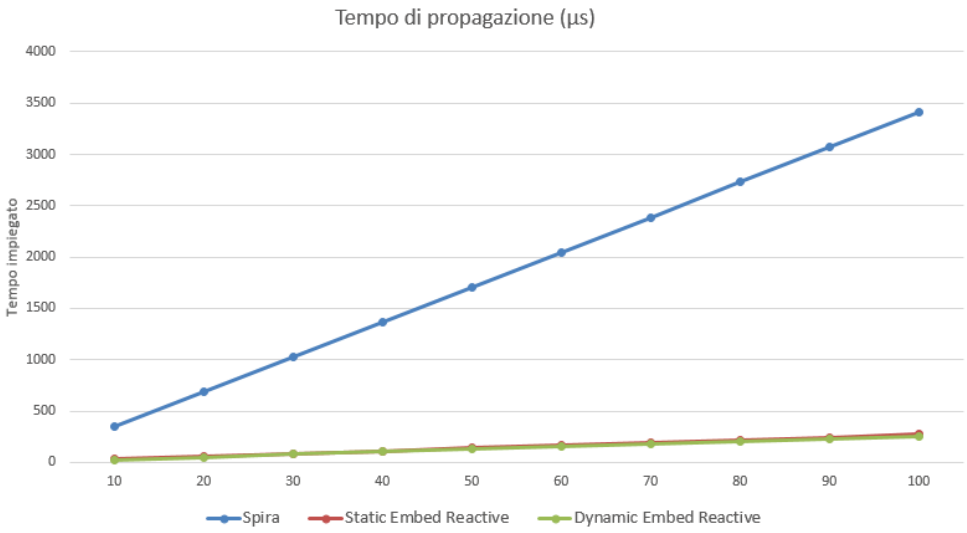

Figura 5.13: Tempo propagazione in un grafo lineare con Cortex-M0.

Per quanto riguarda i tempi di propagazione, si trova il solito miglioramento rispetto al Cortex-M3 giustificato precedentemente. Inoltre si nota che le due strutture dati di Embed Reactive si comportano allo stesso modo a differenza dell'altro microcontrollore, e guardando i dati si è riscontrato che il miglioramento si è avuto da parte di Embed Reactive con struttura dati statica. Ciò è dovuto al fatto che il microcontrollore Cortex-M0 lavorando ad una frequenza superiore, riesce a sfruttare maggiormente la maggiore velocità di calcolo, avendo un inferiore dispendio di tempo per l'accesso alla memoria sfruttando il prefetch delle locazioni di memoria statica contigue in cui dovrà accedere. Se ne deduce quindi che la struttura dati statica tragga dei benefici dall'utilizzo di un clock più elevato, mentre per quanto riguarda la struttura dati dinamica vi è un maggiore tempo di accesso a memoria che costituisce un fattore limitante indipendente dal clock.

5.4.4 Grafo Casuale

In questo test si vuole creare un grafo in maniera casuale, per simulare al meglio il funzionamento della libreria in situazioni non predicibili. Questo per far sì di considerare nei test il più ampio spettro possibile di situazioni in cui può essere applicata Embed Reactive.
Il grafo viene realizzato tramite un algoritmo iterativo e con l'ausilio di valori

generati in modo random. Per ogni iterazione viene deciso, in base ad una variabile random 0-1, se creare un figlio dei due nodi precedenti effettuando un merge, oppure prendere un nodo del grafo e creare n livelli di m figli con n tra 1 e 2 e m tra 2 e 6. L'iterazione successiva dell'algoritmo parte considerando l'ultimo/i nodo/i creato/i.

L'esempio della struttura creata è visibile in figura 5.14. Per realizzare la valutazione sono state prese misurazioni su 4 grafi differenti realizzati per ogni valore sull'asse delle ascisse. Per ottenere il valore finale da rappresentare nei chart è stata effettuata una media aritmetica tra i 4 valori ricavati dai grafi.

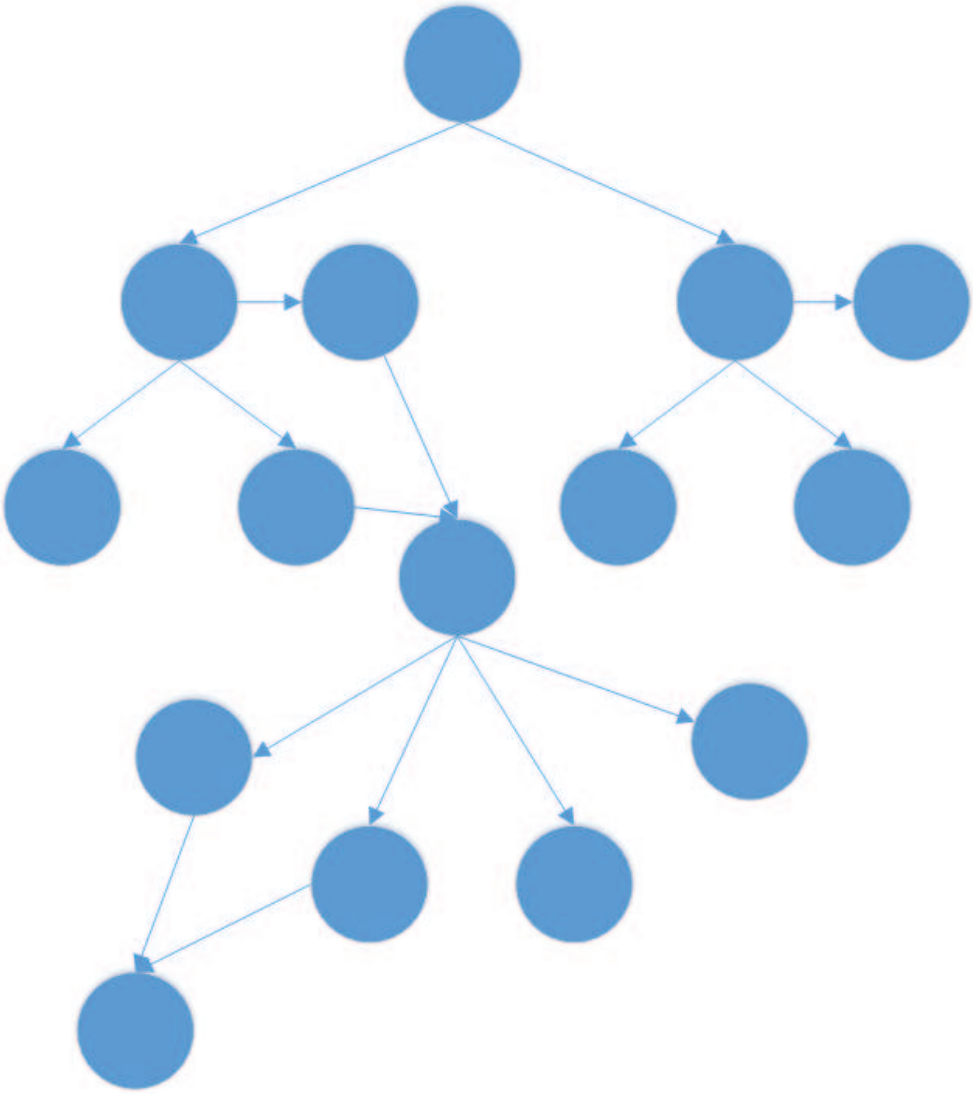

Figura 5.14: Grafo casuale.

Risultati con board con Cortex-M3

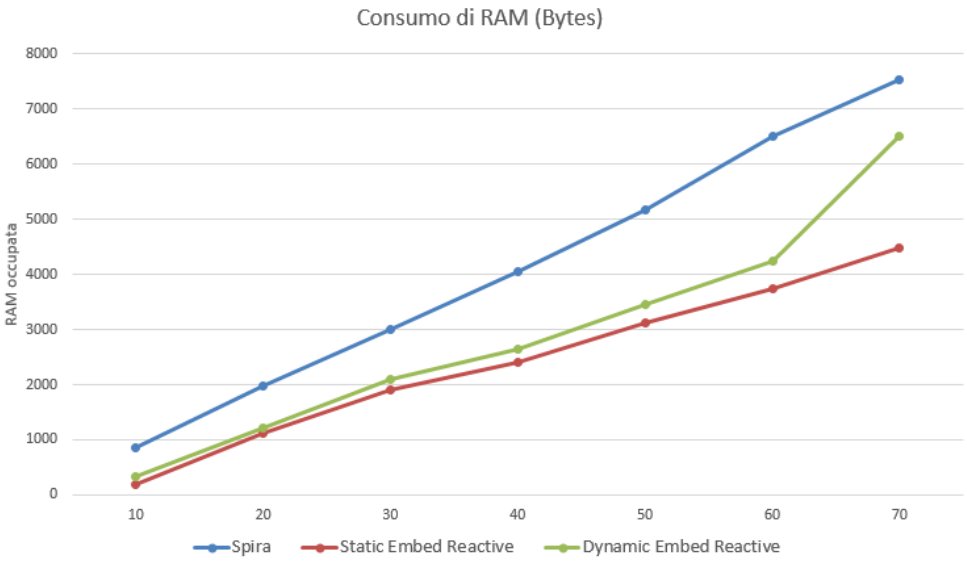

Figura 5.15: RAM occupata dal grafo casuale con Cortex-M3.

Nel caso preso in esame nel charter 5.15, fino alla penultima misurazione, i consumi di RAM tra le due versioni della libreria risultano molto più vicini tra loro rispetto a tutti i test effettuati fino ad ora. Questo causato dal dimensionamento degli array fatto prima dell'inizio del test, che essendo però un test con generazione casuale del grafo, non dà la possibilità prevedere una struttura precisa. Si dimensionano quindi gli array considerando il numero di nodi massimo creato nel peggiore dei casi. Questo quindi causa una maggiore quantità di memoria allocata nel BSS e quindi un aumento complessivo di consumo di memoria. Nell'ultima misurazione si nota un risultato che per Dynamic Embed Reactive non rispetta il trend delle precedenti misurazioni. Questo perchè i grafi generati durante quella misurazione sono stati per casualità più complessi di quelli generati per lo stesso valore con Spira e Static Embed Reactive.

Il tempo di esecuzione della propagazione visibile nel chart 5.16, risulta anche in un contesto di generazione casuale del grafo, elevato per Spira e simile tra le sue strutture dati di Embed Reactive. Il guadagno di Embed Reactive rispetto a Spira aumenta tanto più il grafo diventa complesso. Ciò è dovuto al fatto che la maggior parte del tempo impiegato da Spira per propagare, è causato dall'accesso alle varie liste per ogni oggetto che avviene ogni volta per tutte e tre le liste anche in assenza di elementi in esse. Se ne deduce che più ho un grafo complesso, più ho puntatori a funzioni nelle liste dei vari nodi e più tempo si impiega per scansionare le liste. In Embed Reactive invece l'accesso alla struttura dati risulta essere più veloce essendo un unico vettore, mentre il tempo impiegato per la lettura degli

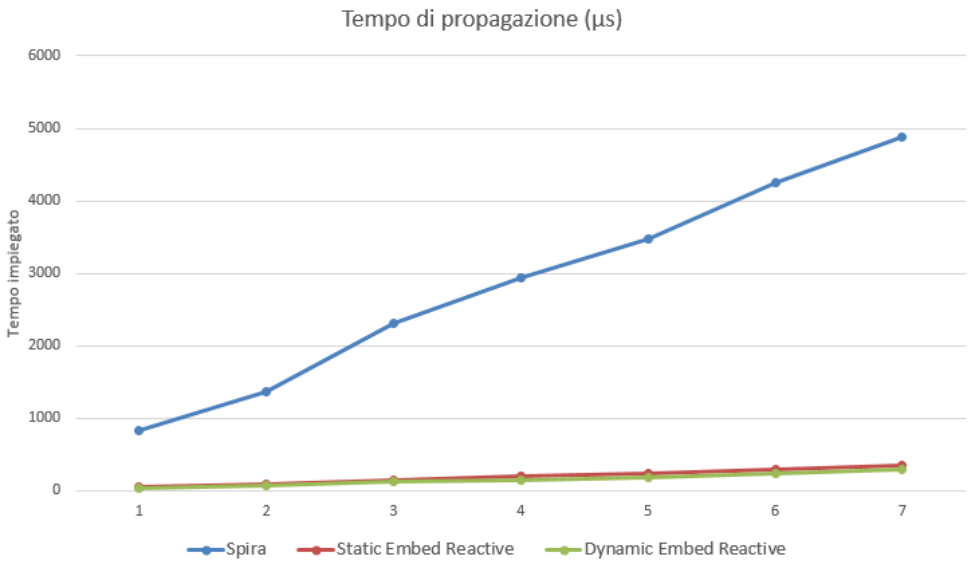

Figura 5.16: Tempo di propagazione in un grafo casuale con Cortex-M3.

elementi è inferiore, trattandosi di un vettore in un caso e di un array statico nell'altro, come evidenziato nella sottosezione 4.3.1.

Risultati con board con Cortex-M0

Sono state effettuate anche con la board con microcontrollore Cortex-M0, le stesse misurazioni effettuate con il Cortex-M3, sempre generando 4 grafi per valore da misurare ed effettuando la media aritmetica dei risultati per ricavarne dei chart. I dati ricavati risultato rispettare i consumi di RAM e i guadagni in tempo di propagazione visti con la Cortex-M3.

5.5 Commenti sui Risultati

Nella totalità dei test si evince che Static Embed Reactive sia mediamente più performante del 50% in termini di consumo di memoria rispetto a Spira. Tale miglioramento è stato possibile grazie all'utilizzo di strutture statiche e globali, condivise tra i vari nodi, oltre che per l'eliminazione di strutture dati non strettamente necessarie per le funzionalità necessarie.

Per quanto riguarda il tempo di esecuzione della propagazione, il risparmio di Embed Reactive rispetto a Spira è visibile per ogni test effettuato. Questo è dovuto alle differenze nell'algoritmo di propagazione, che nel caso di Spira deve considerare di accedere ogni volta a tutte e tre le liste di ogni oggetto, mentre nel caso di Embed Reactive ha una gestione più semplificata del singolo vettore

presente. Notando infatti i chart dei tempi di propagazione, che sono esponenziali per tutti i test ad eccezione di quello in sezione 5.4.3 a causa dell'aumento esponenziale dei nodi per ogni step di misurazione, è subito visibile la differenza che intercorre tra le due librerie nella crescita della curva esponenziale. Infatti per Spira accedendo a tre liste per nodo, avrò una base esponenziale di tre volte rispetto a quella di Embed Reactive, spiegando di fatto la diversità nei trend.

Come si nota, la libreria con struttura dati statica impiega qualche micro secondo in più visto che utilizza una funzione di propagazione più complessa (vi è un'unico array globale per i puntatori gestito da una classe a parte), e utilizza i tipi di variabili interi presi dalla libreria stdint.h, che dopo i test sono risultati peggiorativi per il tempo di esecuzione di pochi micro secondi.

Nel test descritto in sottosezione 5.4.2, vi è una differenza nel tempo di esecuzione. Infatti in questa struttura le foglie, che sono numericamente superiori a tutti gli altri nodi padri, non propagano nulla, e non eseguono quindi alcuna funzione di propagazione risparmiando la metà del tempo rispetto a prima. Nel test in 5.4.1 vi era un unico nodo a non propagare mentre adesso 2^n nodi.

Nel test in 5.4.3 la struttura dati cresce in maniera lineare con l'aumentare dei nodi, dato che è semplicemente un branch lungo n nodi. Questo si evince anche dai risultati, che rispecchiano i risparmi di memoria visti precedentemente, ma che sono rappresentati tramite delle rette con coefficiente angolare differente in base al consumo di memoria o tempo impiegato.

Altra osservazione che si può fare sui test è quella sulle differenze riscontrate tra la board con microcontrollore Cortex-M3 e quella con Cortex-M0. Per quanto riguarda il consumo di memoria RAM è praticamente immutato in entrambi i casi. Per quanto riguarda il tempo di esecuzione invece, si riscontra un miglioramento con la seconda board (con M0). Pur essendo M0 un processore che lavora con un clock più elevato, si caratterizza maggiormente per il risparmio energetico. Il microcontrollore M3 invece lavora a una frequenza più bassa, ma possiede delle feature in più che ne migliorano le prestazioni. Eccelle infatti tra le altre cose, per l'ottimizzazione del suo set di istruzioni e per una pipeline più lunga [2]. Nonostante ciò i test sembrano premiare il processore con il clock più elevato, quindi la sequenza di operazioni che avvengono durante la propagazione, non trova giovamento dalle feature dell'M3 bensì trae vantaggio dalla frequenza di clock del Cortex-M0 che è il 50% superiore all'altro. Infatti i tempi misurati con la board con Cortex-M3 risultano essere esattamente più elevati del 50% rispetto all'altra board.

Capitolo 6

Caso di Studio

In questo capitolo si parla dell'applicazione della libreria realizzata ad un caso concreto. Nello specifico si è voluta applicare Embed Reactive al firmware di un nano-drone, e di seguito saranno descritte le modifiche apportate al firmware stesso per l'inclusione della libreria, la creazione di un esempio di grafo fatto con oggetti reattivi e il test di volo effettuato con il firmware customizzato con la libreria Embed Reactive.

6.1 Il nano-drone

I droni sono dei velivoli radiocomandati con pilota remoto, molto utilizzati per le riprese video e trovano larga applicazione in molte delle attività civili, dal controllo del territorio all'analisi dei terreni fino alla ricerca di dispersi dopo una calamità naturale. Variano molto anche in dimensioni, infatti in questo caso la prova è stata eseguita su un nano drone crazyflie, prodotto da bitcraze. Come dice la parola stessa questi droni sono caratterizzati da dimensioni molto ridotte rispetto alla media, e da un peso irrisorio di circa 20 grammi. E' quindi ovviamente necessario un basso consumo di energia per garantire i 7 minuti di volo promessi con una carica completa della batteria in dotazione standard. Questa d'altra parte, visto le ridotte dimensioni della struttura generale, non potrà eccedere in dimensioni e peso, e quindi in capacità. Per garantire una buona economicità energetica, viene utilizzato un sistema embedded con specifiche hardware quali RAM e CPU molto limitate, dotato di un firmware che occupa quasi interamente la memoria disponibile, dando per contro poca possibilità di customizzazioni o di aggiunte di terze parti.

Per questo progetto è stato utilizzato come caso di studio un crazyflie 1.0 che è la prima versione creata da bitcraze, e anche quella più limitata in caratteristiche hardware.

Figura 6.1: Il crazyflie 1.0

Come detto prima il crazyflie si presenta con una configurazione minimale, però allo stesso tempo non mancano sensori e periferiche, molto utili in fase di volo o in fase di programmazione e controllo. Più precisamente il crazyflie ha come caratteristiche principali:

- Piccolo e leggero, circa 19g di peso e circa 90mm di distanza motore-motore
- Tempo di volo fino a 7 minuti con la batteria standard standard 170mAh Li-Po
- Una board STM32F103CB con microcontrollore con frequenza di clock a 72 MHz, 128kb flash e 20kb RAM.
- Giroscopio e accelerometro a 3 assi.
- Magnetometro a 3 assi HMC5883L (bussola).
- Altimetro ad alta precisione.

Per il controllo remoto del drone si può utilizzare il client CF, che grazie al Crazyradio permette di comunicare con il nano drone oltre che programmarlo. Il client CF è un'applicativo che permette di collegarsi al crazyflie.

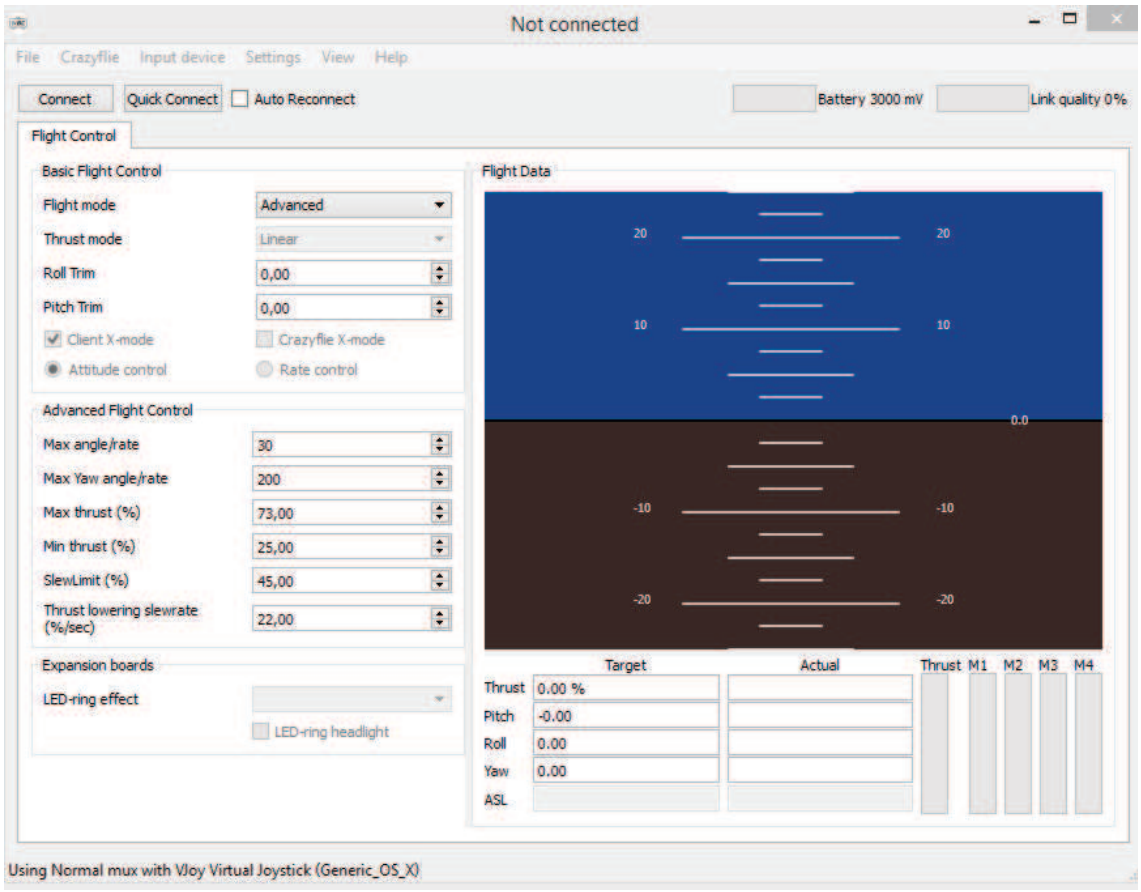

Figura 6.2: Interfaccia grafica del client

Il client CF è scritto interamente in python e ha un'interfaccia grafica in grado di visualizzare in tempo reale dati provenienti dal drone come in figura 6.2. Sono visibili ad esempio inclinazione dell' asse del drone rilevata tramite giroscopio, livello della batteria e potenza erogata ai motori.

Il client è composto da una GUI ma anche da una console, che rimane aperta durante tutto il tempo di funzionamento del client, e rileva eventuali errori, oltre che fare un report in tempo reale, dei messaggi scambiati tra client e drone tramite il segnale radio.

Il firmware si occupa di tutta la gestione hardware su ogni livello e incorpora un sistema operativo real time con un loop di controllo, che permette al drone di regolare le rotazioni delle eliche, per effettuare i movimenti sui 3 assi.

Il firmware è fornito dalla casa produttrice ed è un progetto open source scritto in C, per questo non immediatamente integrabile con Embed Reactive scritta invece in C++.

Come detto precedentemente il firmware si occupa di tutta la gestione hardware, ed è diviso in più livelli:

- Il livello di configurazione, che serve ad attivare e disattivare funzionalità del nano drone. E' possibile dimensionare anche le aree di memoria dedicate allo stack e allo heap.
- Vi è poi una parte di hardware driver layer. Questa contiene tutti i driver delle board dei droni e si occupa del mapping della memoria e dei registri. E' in assoluto la parte che agisce più a basso livello nel firmware.
- La precedente parte si interfaccia con la parte di HAL (hardware abstraction layer). Questa contiene il codice necessario al controllo dei singoli elementi hardware quali LED, timers e antenna per Crazyradio.
- Successivamente vi è la sezione dei moduli, che contiene tutto ciò che che si interfaccia con l'ambiente: lettura di valori dai sensori, calcolo di accelerazioni da imprimere alle eliche, controllore PID per gli attuatori, servizi per il logging, servizi per i comandi da parte dell'utente e infine le funzioni che inizializzano il sistema e fanno partire il loop di controllo dopo le verifiche del caso.
- Poi vi è tutta una parte di funzionalità accessorie. Sono per lo più funzionalità utili in fase di debugging o comunicazione con terze parti.

6.2 Code Refactoring

Il firmware è scritto in C, per poter integrare quindi Embed Reactive sono necessarie alcune modifiche del progetto, che lo rendano adatto ad ospitare una libreria C++. Queste modifiche interessano la parte di compilazione, la parte di configurazione e la conversione di alcuni file in formato `cpp`. La prima cosa da fare è stata inserire i file di Embed Reactive dentro la directory del firmware dedicata alle librerie esterne. Poi sono stati effettuati cambiamenti nel firmware che hanno permesso di includere la libreria e compilarla, sono stati quindi riscritti in C++ i file destinati a contenere gli oggetti di Embed Reactive in modo tale da renderli compatibili.
E' stato poi identificato il loop di controllo, caratterizzato da un ciclo infinito all'interno del quale sono presenti chiamate a funzioni che si interfacciano con i sensori, per leggere le variabili dall'ambiente.

```
1 //dichiaro 3 oggetti reattivi con valore float
2 Signal<float> reactAX, reactAY, reactAZ;
3
4 //dichiaro un nodo dipendente dai 3 sopra tramite una formula
5 Signal<float> reactTOT(reactAX*reactAX + reactAY*reactAY
```

```
6                   + reactAZ*reactAZ);
7
8 //loop di controllo
9 while(1){
10    sensReadAcceleration(ax, ay, az);
11
12    AX.push(ax);
13    AY.push(ay);
14    AZ.push(az);
15
16    sendAccToPID(reactTOT.getValue());
17 }
```

Listing 6.1: Loop di controllo con Embed Reactive

Nel listato 6.1 è presente uno pseudo loop di controllo ispirato a quello del firmware originale del crazyflie. In questo vengono lette le accelerazioni sui 3 assi dai sensori e poi vengono propagati i valori alla funzione di interfaccia con il PID. E' chiaro che all'interno di questo loop si possa applicare reactive programming alle variabili lette dai sensori, che sono state copiate dopo ogni lettura in oggetti reattivi. Questi oggetti salvano i valori ricevuti in input all'interno della push, e nel caso in cui questi cambino, li propagano ad un altro oggetto reattivo che sarà passato in input al PID.

6.3 Configurazione

Come detto nella sezione 6.1, il nano-drone CrazyFlie 1.0 è fornito di 20 KB di memoria RAM.
Questa, a differenza della memoria Flash che non costituisce un problema per la customizzazione del firmware, è sufficiente appena a contenere il firmware con le sue variabili e le sue funzioni.
Questo causa un problema quando si applica Embed Reactive, dato che si vanno ad allocare nuovi oggetti e si vanno a fare chiamate di funzioni durante la propagazione andando ad incrementare la dimensione dello stack. Proprio la mancanza di spazio libero ulteriore da poter dedicare allo stack, fa sì che non si possano creare oggetti reattivi che propaghino dati, all'interno di funzioni annidate oltre il secondo livello, come mostrato nel listato 6.2.

```
1 int main(){
2    funct1();
3 }
4
5 void funct1(){
6    ...
7    funct2();
8    ...
9 }
```

```
10
11 void funct2(){
12     ...
13     aX.push();
14     ...
15 }
```

Listing 6.2: Chiamate a funzioni annidate

Per poter guadagnare quindi dello spazio per creare un grafo di propagazione all'interno della funzione che contiene il loop di controllo, si è dovuto agire sul file di configurazione del firmware che permette di impostare come è suddivisa la memoria tra stack e heap. Per fare ciò si sono fatti alcuni test per capire fino a che dimensione si potesse ridurre il valore dello heap per guadagnare spazio per lo stack.

Un'eccessiva limitazione dell'area adibita al contenimento dello heap, causa tuttavia un malfunzionamento del drone che, non riuscendo ad allocare dinamicamente all'avvio tutte le variabili di cui ha bisogno il firmware per funzionare, raggiunge uno stato di errore, mandando in timeout il sistema. Quindi da quanto detto si apprende che si possono avere due tipologie di problemi legati alla memoria RAM che non fanno volare il drone:

- Uno causato dall'assenza di memoria disponibile per permettere allo stack di crescere, ed effettuare quindi propagazioni di un certo livello di profondità nel grafo logico delle dipendenze, dato che l'algoritmo di propagazione prevede chiamate annidate di funzioni push e propagate.
- Un altro causato da un'eccessiva riduzione dell'area di memoria dedicata allo heap per fare spazio allo stack, dato che un heap troppo ridotto implica la mancanza di spazio per allocare tutte le variabili necessarie al funzionamento del firmware del drone.

Risulta quindi chiaro che nell'applicazione di reactive programming all'interno del loop di controllo del drone, è stato necessario trovare il giusto equilibrio tra dimensionamento dello heap e spazio da lasciare a disposizione dello stack. L'equilibrio trovato ha permesso al drone di funzionare correttamente e al tempo stesso di inserire un grafo di dipendenze fatto da una relazione padre-figlio tra 4 nodi totali.

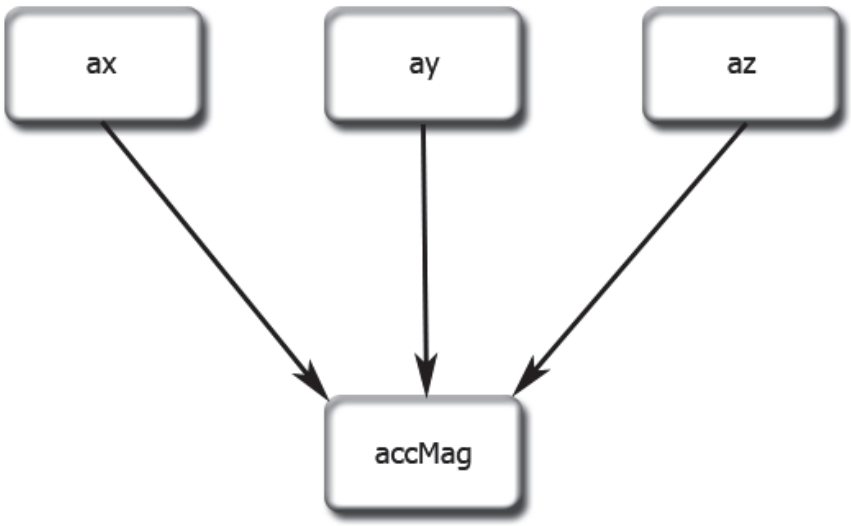

Figura 6.3: Grafo creato

In una situazione in cui idealmente non ci fossero stati limiti hardware, si sarebbe potuto estendere l'applicazione di Embed Reactive ad altre variabili presenti nel loop di controllo. Nella situazione reale però, a causa di limiti hardware del nano-drone, si è creato il grafo massimo che si potesse creare come quello in figura 6.3.
Il grafo è stato fatto con 4 oggetti reattivi di cui 3 applicati alle accelerazioni attuali lette sui tre assi (x,y,z), e uno che è l'oggetto risultante dai calcoli effettuati con le 3 accelerazioni precedenti. Il valore contenuto in tale oggetto sarà poi utilizzato dal PID per regolare le nuove velocità di rotazione dei motori.
Con i nuovi oggetti reattivi, nel caso in cui una o più accelerazioni sui vari assi, risulti immutata rispetto alla lettura precedente, non vengono effettuati alcuni calcoli avendo quindi un possibile risparmio potenziale per ogni iterazione del loop di controllo.
Nel caso in cui si voglia generare un grafo di dipendenze più complesso, si causerebbe il non funzionamento del firmware a causa della mancanza di RAM.

6.4 Test

Dopo aver applicato il firmware reattivo al nano-drone, è stato effettuato un test di confronto tra il drone con il firmware modificato e il drone con il firmware originale.

L'esperimento consiste in un volo indoor del nano-drone che sfrutta un sistema di localizzazione 3D, effettuato in laboratorio tramite un lavoro di ricerca inerente a quest'ambito. Il risultato del test è espresso come la precisione nel mantenere la propria posizione nel tempo. In questo tipo di test si può valutare la qualità del loop di controllo del firmware del drone, dato che ad un buon controllo corrisponde una buona posizione. Per poter identificare la posizione del drone si usa localizzazione vision-based, utilizzando Microsoft Kinect come input device.

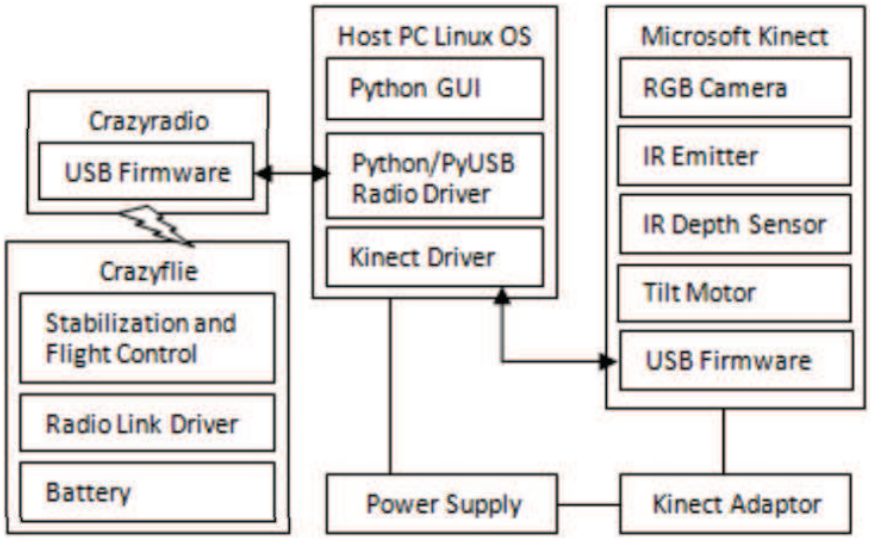

Figura 6.4: Architettura

L'architettura del sistema è quella mostrata in figura 6.4, in cui si può notare l'aggiunta dei Kinect interfacciati al crazyflie tramite il client CF e crazyradio.

Figura 6.5: Test di precisione

L'errore di posizionamento rispetto alla posizione ottimale è stato misurato tramite l'utilizzo di una palla rossa montata sopra al drone di facile identificazione da parte dei Kinect. Grazie a questa si riesce a calcolare l'errore della posizione del nano-drone rispetto ad un riferimento, come può essere l'altro cerchio di colore blu.

Gli esperimenti sono stati fatti con il Kinect posizionato a 1 m da terra e il target esattamente alla stessa altezza e 2m davanti al kinect in una direzione perpendicolare rispetto ad esso. Le misurazioni sono state fatte ad una frequenza di 2 Hz per un intervallo di 3000 secondi. Per leggibilità dei grafici però vengono riportate nel chart solamente 200 misurazioni corrispondenti ad un periodo di 100 secondi.

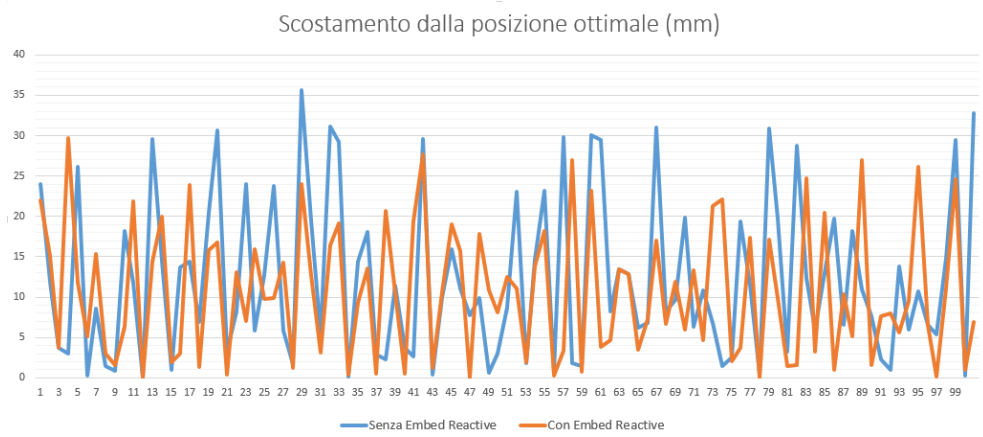

Figura 6.6: Errore assoluto del posizionamento del nano-drone nel tempo

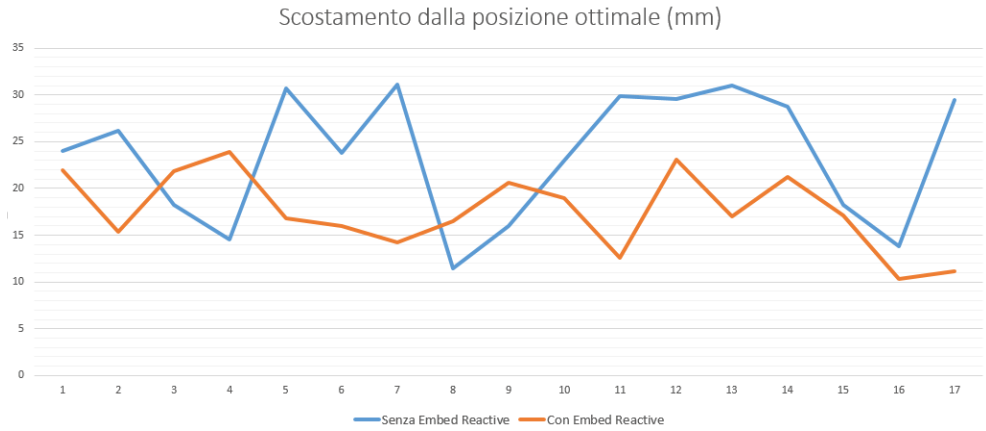

Figura 6.7: Errore assoluto interpolato

Il grafico 6.6 è stato costruito calcolando il valore assoluto della precisione in millimetri, che essendo stata rilevata rispetto ad un punto di riferimento presenta sia valori negativi che positivi. Sono stati presi i valori assoluti perché il dato che interessa per ricavare un'informazione di confronto, è lo scostamento rispetto al valore 0 cioè l'errore assoluto. Inoltre per semplicità di lettura è stato fornito un altro grafico in figura 6.7, costruito facendo interpolazioni tra più punti intermedi in modo tale da ridurre la densità dei picchi.

Le misurazioni sono state iterate 5 volte sia utilizzando il firmware originale che utilizzando quello con Embed Reactive. Nel grafico il valori del primo sono contraddistinti da una curva color blu, mentre i valori per il firmware reattivo sono quelli di colore arancione.

Le curve sono abbastanza simili tra loro essendo comprese tra valori assoluti

di spostamenti compresi tra 0 e 40 mm. Per la curva arancione, corrispondente al drone con il firmware reattivo, risulta un lieve miglioramento nella posizione. Infatti calcolando l'errore medio si trova che per il firmware modificato si ha una media dello scostamento dalla posizione ottimale di 10.14 mm, mentre con l'altro si ha un errore medio di 13.85 mm evidenziando un miglioramento di circa il 25% nel mantenere la posizione.

Capitolo 7

Conclusioni

Il progetto descritto in questo lavoro di tesi ha voluto sviluppare Embed Reactive, una libreria di supporto per chi tra i programmatori di sistemi embedded a risorse limitate, voglia utilizzare il paradigma Reactive Programming.

L'idea della progettazione di una libreria per sistemi embedded a risorse limitate è nata dal presupposto che non ne fosse ancora stata realizzata, a maggior ragione con linguaggio C++, una compatibile con questo tipo di sitemi, dato che quelle esistenti non avevano come scopo primario il risparmio di risorse. Ora invece si fornisce ai programmatori un supporto per reactive programming con consumi di memoria irrisori. Consumi questi dimostrati dai test, che hanno evidenziato un miglioramento sull'utilizzo di memoria RAM di circa il 50% rispetto a Spira, libreria già esistente ma non per sistemi embedded a cui è ispirata questa realizzata. Nei test si è riscontrato anche un miglioramento superiore all' 80% nei tempi di esecuzione sempre con un paragone con Spira effettuato prima. Questi risultati sono stati raggiunti grazie all'ideazione di una struttura dati più efficiente che viene allocata in memoria statica, annullando di fatto ogni overhead causato dalle allocazioni dinamiche e risparmiando più memoria possibile. Per arrivare a tale soluzione si è partiti da una struttura dati formata da vettori dinamici distribuiti sugli oggetti creati tramite Embed Reactive fino ad arrivare ad un unico array statico centralizzato.

L'applicazione di Embed Reactive ad un caso reale ha mostrato risultati in linea con i criteri di progetto definiti a inizio lavoro, il tutto nei limiti delle possibilità offerte dall'hardware scelto per l'applicazione. I test sulla precisione di posizionamento nello spazio effettuati sul nano-drone, confrontando il firmware originale con quello modificato con reactive programming usando la libreria, evidenziano un miglioramento di circa il 25% nel mantenere la posizione, nel caso del firmware modificato.

L'hardware, scelto volutamente molto restrittivo, voleva essere un buon test per

appurare l'applicazione della libreria ad un numero molto ampio di ambienti. Infatti il fatto di funzionare sul nano-drone con un grafo semplice, implica che in sistemi più performanti di questo, come gran parte dei sistemi embedded, la libreria possa essere integrata permettendo di creare grafi di dipendenze di complessità superiore.

La metodologia ideata per la gestione delle risorse, il design impiegato per la realizzazione e la valutazione effettuata, permettono di affermare che l'applicazione della libreria in un contesto di sistemi embedded a risorse limitate, abbia un impatto positivo nel caso si cerchi un supporto per programmare con Reactive Programming, perché permette di migliorare l'efficienza dei sistemi che basano sulla reattività tutto il loro funzionamento.

Alcune estensioni possono essere effettuate per gestire altre tipologie di strategie di propagazione dei dati nel grafo, oltre all'attuale depth first. Inoltre la libreria è costruita in modo tale da poter permette l'aggiunta di nuovi tipi di collegamenti all'interno dei grafi. Infatti basterà aggiungere nuovi costruttori per implementare nuove funzionalità per quanto riguarda la costruzione delle strutture di dipendenze tra oggetti reattivi. Infine la bidirezionalità dei flussi di dati all'interno del grafo delle dipendenze, può essere un buono spunto per migliorie future da apportare ad Embed Reactive

Concludendo, tramite il presente lavoro di tesi spero di aver dato un contributo nel fornire un supporto per Reactive Programming ai programmatori di quei sistemi embedded che hanno a che fare con problematiche di risorse limitate. Di permettere altresì una semplificazione nell'utilizzo di un paradigma che sta prendendo sempre più piede, grazie all'affermazione di applicazioni sempre più dipendenti da aggiornamenti in tempo reale, e alla diffusione di sistemi che necessitano di comunicare con l'ambiente e svolgere operazioni con reattività.

Appendice A

Appendice

A.1 Documentazione delle API

In questa sezione sono presenti le API della libreria riportate sottoforma di codice dichiarativo del prototipo delle funzioni e dei costruttori. Il codice è accompagnato dalla documentazione delle funzioni come commento, con all'interno input, output e descrizione dei metodi e costruttori. Per entrambe le classi della libreria sono presentate le variabili, i costruttori e i metodi.

A.1.1 Classe Signal

Variabili

- **Struct signal<T>:** Questa è la struttura principale della classe, che contiene tutte le variabili che verranno poi utilizzate nell'implementazione. Il tipo T della struct serve a specializzare l'oggetto con il tipo scelto dal programmatore
  La struct contiene:
  - **T value**: è il valore, che può essere di qualsiasi tipo, memorizzato dall'oggetto reactive.
  - **uint16t rate**: Indica con che frequenza il signal propagherà il suo valore. Questa variabile assume un valore intero che sarà decrementato di una unità, ogni volta che ci sarà un aggiornamento del valore del signal. Quando raggiunge 0, il signal propagherà il dato ai nodi successivi. Se ne deduce che più il rate assume un valore elevato, meno frequentemente avverrà una propagazione, e che se il rate è 0, ogni aggiornamento sarà propagato.
  - **uint8t dim**: Questa variabile rappresenta il numero di signal dipendenti successivi a cui propagare il valore. Viene incrementata in modo

automatico ogni volta che un nuovo collegamento è creato tra il signal corrente e uno dipendente a questo.

– **uint8t id**: E' l'id univoco del signal, assegnato alla creazione dell'oggetto.

Costruttori

1.Costruttore di default

```
inline signal();
```

Inizializza il valore a NULL e assegna 0 al rate.

Esempio d'uso:

```
mylib::signal<T> sA
```

2.Costruttore per il merging

```
inline signal(signal<T>& s1, signal<T>& s2, uint16_t  rate);
```

Input:
s1,s2: entrambi altri oggetti signal dichiarati precedentemente
rate: frequenza di aggiornamento del value

Alloca un nuovo oggetto il cui value sarà aggiornato con il value di s1 o s2, ogni volta che uno di questi cambierà.

Esempio d'uso:

```
mylib::signal<T> sA(s1, s2, 0)
```

3.Costruttore con funzione di aggiornamento

```
inline signal(uint8_t num, uint16_t  rate, const std::function<T()> func, ...)
```

Input:
num: numero di oggetti signal da cui dipende il nuovo oggetto allocato
rate: frequenza di aggiornamento del value
func: puntatore alla funzione il cui risultato aggiorna il value del nuovo oggetto scritta come lambda expression

...: numero variabile di argomenti ognuno dei quali corrisponde ad un oggetto da cui dipende

Alloca un nuovo oggetto il cui value sarà aggiornato con il value restituito da func ogni qual volta che uno degli oggetti da cui dipende propagano il loro value.

Esempio d'uso:

```
mylib::signal<int> sC(2, 0, std::function<int()>(
          [&sA, &sB](){return sA.getValue() + sB.getValue();
      }), sA, sB);
```

4.Costruttore di merging con funzione di aggiornamento

```
inline signal(signal<T>& s1, signal<T>& s2,
              uint16_t  rate, const T (*func)(T,T));
```

Input:
s1,s2 entrambi altri oggetti signal dichiarati precedentemente da cui dipenderà il nuovo oggetto
rate: frequenza di aggiornamento del value
(*func)(T,T): puntatore alla funzione il cui risultato aggiorna il value del nuovo oggetto, non scritta come lambda expression e che accetta solamente 2 valori in input

Alloca un nuovo oggetto il cui value sarà aggiornato con il value restituito dalla funzione puntata da *func ogni qual volta che uno degli oggetti da cui dipende propagano il loro value. La funzione scritta dal programmatore accetta solamente 2 input, inoltre questi devono corrispondere agli oggetti s1 ed s2

Esempio d'uso:

```
int foo(int v1, int v2){
   return v1+v2;
}

mylib::signal<int> sC(sA, sB, 0, &foo);
```

5.Costruttore per collegamento semplice padre-figlio

```
inline signal(uint16_t  rate, signal<T>& s1);
```

Input:
rate: frequenza di aggiornamento del value
s1: oggetto signal dichiarato precedentemente

Alloca un nuovo oggetto il cui value sarà aggiornato con il value di s1 quando s1 propaga lo.

Esempio d'uso:

```
mylib::signal<int> sB(0, sA);
```

Metodi

1.Get value

```
template<typename T>
T getValue();
```

Input: -

Funzione di interfaccia che restituisce il value.

Output: T value

Esempio d'uso:

```
sC.getValue();
```

2.Funzione Bind

```
void bind(int num, const std::function<T()> func, ...Arg);
```

Input:
num: numero di oggetti signal da cui dipende il nuovo oggetto allocato
func: puntatore alla funzione il cui risultato aggiorna il value del nuovo oggetto
...Arg: numero variabile di argomenti ognuno dei quali corrisponde ad un oggetto da cui dipende il chiamante della bind

Aggiunge uno o più nuovi oggetti signal da cui il chiamante dipenderà e specifica una funzione di aggiornamento.

Output: -

Esempio d'uso:

```
sC.bind(2, std::function<int()>([&sA, &sB](){return sA.getValue() +
sB.getValue();}), sA, sB);
```

3.Funzione Push

```
void push(T value);
```

Input:
value: valore del tipo scelto

Aggiorna il value con quello in input e lo propaga lungo i nodi successivi, che sono gli oggetti i cui valori dipendono dal chiamante. Se non vi sono nodi successivi aggiorna solamente il value con quello in input.

Output: -

Esempio d'uso:

```
//in caso di ipotetico tipo T scelto come int
sA.push(2);
```

4.Funzione Push con condizione

```
void push(T value, bool ver);
```

Input:
value: valore del tipo scelto
ver: comando per azionare la propagazione

Come la precedente push, però porta a termine l'aggiornamento solo se la booleana in input è true.

Output: -

Esempio d'uso:

```
sA.push(2,false);
```

5.Funzione di impostazione del Rate

```
void setRate(uint16_t  rate);
```

Input:
rate: frequenza di aggiornamento

Aggiorna il rate con quello in input

Output: -

Esempio d'uso:

```
sA.setRate(0);
```

A.1.2 Classe Connector

Variabili

- uint8t count: contatore del numero di oggetti signal creati. Viene usato per l'assegnamento di id univoci agli oggetti signal.
- std::array<std::function<void(T)>, V> link-vect: array statico di puntatori alle funzioni lambda che contengono le push dei vari oggetti signal. T è il tipo di value scelto, V è il numero di elementi dell'array.
- std::array<uint8t, N> starts: array statico che per ogni signal esistente, memorizza l'indice di partenza, all'interno di link-vect, della porzione di array che contiene i propri puntatori alle lambda. Ogni elemento dell'array è nella posizione corrispondente all'id dell'oggetto signal cui è riferito il dato. N è il numero di oggetti dell'array.

Costruttori

1.Costruttore di default

```
template<typename T, unsigned int N, unsigned int V>
linker();
```

Inizializza count a 0.

Esempio d'uso:

```
mylib::linker<T,N,V> l
```

Metodi

1.Get Count

```
uint8t getCount();
```

Input: -

Funzione che restituisce il valore di count e lo incrementa di 1. Viene utilizzata per assegnare un id ad ogni oggetto signal creato

Output: uint8t count

Esempio d'uso:

```
l.getCount();
```

2.Funzione di shift del vettore

```
void shiftVect(uint16t pos);
```

Input:
pos: posizione di linkVect da cui iniziare lo shift a destra

Esegue lo shift verso destra di tutti gli elementi del vettore successivi all'elemento pos.

Output: -

Esempio d'uso:

```
l.shiftVect(5);
```

Bibliografia

[1] I processi e l'uso della memoria. http://users.lilik.it/mirko/gapil/gapilsu20.html. Accessed: 2015.

[2] Cortex-m series documentation. http://www.arm.com/products/processors/cortex-m/, Accessed: 2015.

[3] Gcc documentation - binary optimization. https://wiki.gentoo.org, Accessed: 2015.

[4] Stl container memory usage when developing with c++. http://info.prelert.com/blog/stl-container-memory-usage, Accessed: 2015.

[5] Stm32 mcu nucleo documentation. http://www.st.com, Accessed: 2015.

[6] Edward Amsden. Timeflies: Push-pull signal-function functional reactive programming. 2013.

[7] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 411–422, New York, NY, USA, June 2013. ACM Press.

[8] Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009.

[9] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[10] Shriram Krishnamurthi Gregory H. Cooper. Embedding dynamic dataflow in a call-by-value language. pages 294–308, 2006.

[11] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Summer School on Advanced Functional Programming 2002, Oxford University*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003.

[12] U. Khan Z. A. Javaid N. Ilahi M. Khan R. D. Ali L. Qasim. Evaluating wireless reactive routing protocols with linear programming model for wireless ad-hoc networks. 2013.

[13] Andoni Lombide Carreton Stijn Mostinckx Tom Van Cutsem Wolfgang De Meuter. Loosely-coupled distributed reactive programming in mobile ad hoc networks. In *Objects, Models, Components, Patterns*, volume 6141 of *Lecture Notes in Computer Science*, pages 41–60. Springer Berlin Heidelberg, 2010.

[14] Bainomugisha E. Lombide Carreton A. Van Cutsem T. Mostinckx S. De Meuter. A survey on reactive programming. (52):2–11, 2013.

[15] R. R. Pucella. Reactive programming in standard ml. In *Computer Languages, 1998. Proceedings. 1998 International Conference on*, pages 48 – 57. IEEE, 1998.

[16] Brian C. Williams Phil Kim Michael Hofbaur Jonathan How Jon Kennell Jason Loy Robert Ragno John Stedl and Aisha Walcott. Model-based reactive programming of cooperative vehicles for mars exploration. 2001.

[17] Zhanyong Wan and Paul Hudak. Functional Reactive Programming from first principles. In *Proc. ACM SIGPLAN'00 Conference on Programming Language Design and Implementation (PLDI'00)*, 2000.

[18] Baptiste Wicht. C++ benchmark - std::vector vs std::list. 2012.

[19] Fang Zhou. Graph compression. 2010.