Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione

POLITECNICO
MILANO 1863

Corso di Laurea Magistrale in
Ingegneria Informatica

Condivisione equa di risorse in sistemi per l'Internet of Things

Sharing resources fairly in Internet of Things systems

Relatore:
Prof. Luca Mottola

Tesi di Laurea di:
Paolo Manca - 816713

Anno Accademico 2015-2016

Indice

Abstract **ix**

Abstract (ENG) **xi**

1 Introduzione **1**
1.1 Problema e contributo 2
1.2 Struttura del documento 3

2 Background **5**
2.1 Applicazioni 5
2.2 COIN 7
2.2.1 Architettura 7
2.2.2 Actor-like model 9
2.2.3 Publish-subscribe 10
2.2.4 Earliest Deadline First (EDF) 11
2.3 Fairness 12
2.3.1 Equal sharing fairness 13
2.3.2 Proportional fairness 13
2.3.3 Max-min fairness 15
2.4 Sistemi resource-aware 16
2.4.1 ECOSystem 16
2.4.2 Pixie OS 18
2.4.3 Cinder 19
2.4.4 Rialto 20
2.5 Scheduler resource-aware 21
2.5.1 Fair queueing 21

2.5.2 Share 22
2.5.3 Lottery scheduling 23

3 Obiettivi e requisiti **25**
3.1 Obiettivi 25
3.2 Requisiti 26

4 Soluzione **29**
4.1 Architettura 29
4.1.1 Astrazione delle risorse 29
4.1.2 Messaggi e code in ingresso 30
4.2 Allocazione 33
4.3 Arbitraggio 34
4.4 Queuettery 37

5 Prototipo **41**
5.1 Task 41
5.2 Messaggi e code 42
5.3 Ciclo di vita del task 44
5.4 Lottery 46

6 Evaluation **47**
6.1 E1: Capacità di implementazione dell'allocazione e valutazione dei PRNG 49
6.2 E2: Comportamento dell'errore al crescere del numero di task 52
6.3 E3: Costo computazionale al crescere del numero dei task 53
6.4 E4: Overhead in memoria al crescere del numero dei task 55

7 Conclusioni **59**

Elenco delle figure

2.1 Evoluzione dello stato di un task in COIN 8
2.2 Architettura di COIN 9
2.3 Un network di attori che scambiano messaggi con message passing . . 10
2.4 Publish-subscribe: A_1 pubblica un messaggio di argomento E, del quale A_2 e A_4 sono subscriber 11
2.5 Esempio di allocazione con Progressive Filling; per ciascuno dei quattro utilizzatori è rappresentata la domanda non soddisfatta (bianco) e l'allocazione (azzurro), la linea tratteggiata indica la *water line* 16
2.6 Pixie OS, esempio di applicazione per sensori indossabili 18
2.7 Cinder. La reserve di un browser connessa alla reserve sorgente di una batteria da 15kJ attraverso un tap che fornisce 750mW 20
2.8 Inserimento in coda degli input provenienti da tre sorgenti s_1, s_2 e s_3 22

4.1 Sequence Diagram della pubblicazione ed esecuzione di un dato . . . 32
4.2 Schema di gestione delle risorse in Queuettery 38

6.1 Evaluation board STM32F091RC 48
6.2 [E1] Distribuzione della risorsa tra i quattro task 50
6.3 [E1] Comparazione degli errori relativi dei tre PRNG rispetto all'allocazione ideale. 51
6.4 [E2] Comparazione dell'errore relativo nel caso peggiore tra Queuettery e FIFO 53
6.5 [E3] Tempo medio di esecuzione di Queuettery e FIFO al crescere del numero dei task 54
6.6 [E4] Andamento del costo in memoria in funzione del numero di utilizzatori di Queuettery e FIFO per una risorsa 56

Elenco delle tabelle

2.1 Verifica allocazioni con la definizione 14
2.2 Individuazione dell'allocazione proportional fairness tramite teorema . 14

6.1 [E1] Configurazione dei task 49
6.2 Dimensione delle principali strutture dati del prototipo 55

Abstract

Le mobile app offrono ai sistemi embedded l'interfaccia che usualmente non posseggono, ottenendo in cambio capacità e informazioni specializzate e contestuali. La loro interazione apre un gran numero di possibilità per gli sviluppatori ed infine per l'utente. Il sistema operativo COIN nasce per espandere queste possibilità, rendendo i dispositivi embedded accessibili all'installazione ed esecuzione di piccoli task che incapsulano parti di logica applicativa delle mobile app.

Rendere le risorse del dispositivo embedded fruibili da task arbitrari significa doverne condividere le capacità tra loro. In questa tesi si è affrontato il problema di individuare politiche e meccanismi che permettessero l'allocazione equa di porzioni di risorse diverse (radio, sensori, attuatori, etc.) e l'implementazione della stessa. Gli stessi scopi sono presenti anche in lavori precedenti ma con presupposti differenti; infatti, la novità portata dal sistema operativo COIN nell'approccio a questi sistemi, le cui capacità sono di solito rigidamente limitate, richiede una soluzioni nuova, pensata specificamente per questi.

Dopo aver analizzato le soluzioni esistenti, congiuntamente alle esigenze specifiche di COIN, questa tesi introduce Queuettery, un sistema che permette l'astrazione delle risorse e la loro condivisione attraverso un approccio probabilistico all'arbitraggio. Queuettery propone una composizione ed estensione originale di algoritmi noti che esplora il trade-off tra l'arbitraggio e le limitate capacità dei dispositivi in una soluzione che, al meglio delle mie conoscenze, non trova per questo applicazioni precedenti.

Gli esperimenti condotti mostrano che Queuettery è in grado di offrire scostamenti medi relativi dall'allocazione ideale inferiori al 7%, a fronte di un overhead computazionale inferiore al 1% rispetto alla common practice per sistemi a risorse limitate.

Abstract (ENG)

Mobile apps provide embedded systems the interface that they usually lack, getting in return specialized and contextual functionalities and informations. Their interactions enables a wide range of possibilities for developers and eventually for users. COIN is an operating system developed to expand this possibilities, opening up embedded devices to the deployment and execution of small tasks, which are programmer-defined relocatable slices of mobile app logic.

Exposing the devices' resources to arbitrary tasks entails that their limited capabilities have to be shared among them. This thesis goes into the issue of identifying policies and algorithms that are able to establish and enforce a fair allocation of various resources (e.g. radios, sensors, actuators). The same goals are common with previous works but different in the targeted devices and therefore requirements. In fact, the novelty of COIN in the approach to this kind of devices, the resources of which are usually strictly closed, calls for a new dedicated solutions.

After an analysis of existing solutions and COIN's specific requirements, this thesis introduces Queuettery, a system for resources abstraction and fair sharing. Queuettery is a novel composition and extension of known algorithms that explores a trade-off between resource arbitration and limited device capabilities with a solution which, to the best of my knowledge and for the novelty of the COIN approach, hasn't been proposed before.

Experimental evaluations show that Queuettery can achieve an average relative error in the implementation of ideal allocations below 7%, while keeping the computational overhead below 1% with respect to the common practice on limited resources systems.

Capitolo 1

Introduzione

L'informatica ricopre oggi un ruolo fondamentale nella vita individuale e sociale della quasi totalità delle persone, a prescindere dai loro interessi e ambiti lavorativi. Diversamente da quanto si sarebbe potuto pensare fino anche ad un decennio fa, i dispositivi informatici con cui l'utente comune interagisce maggiormente nella quotidianità non sono dei classici ed iconici computer posizionati su una scrivania. Un vasto numero di dispositivi connessi che l'utente porta con sé (e.g. smartphone, smartwatch, wearables) o incontra nell'ambiente in cui si trova (e.g. beacon, sensori, lampadine controllabili) rappresenta attualmente ciò su cui il mondo dell'informatica per il consumatore si sta espandendo maggiormente.

Le mobile app offrono ai sistemi embedded l'interfaccia che usualmente non posseggono, ottenendo in cambio capacità e informazioni specializzate e contestuali. La loro interazione apre un gran numero di possibilità per gli sviluppatori ed infine per l'utente. Il sistema operativo COIN [1] nasce per espandere queste possibilità di interazione tra i dispositivi mobile e gli altri connessi nell'ambiente, rendendo questi ultimi accessibili all'esecuzione di parti di logica applicativa delle mobile app. Gli sviluppatori mobile hanno la possibilità di accedere alle informazioni e risorse del dispositivo su cui COIN è installato in modo aderente alle proprie necessità applicative mediante l'installazione di piccoli task, invece che attraverso API di basso livello come accade abitualmente con i dispositivi commerciali.

Con queste caratteristiche, COIN si presta come piattaforma per diverse applicazioni. Ad esempio, le sue funzionalità sono state sperimentate per realizzare un

piccolo *pervasive game* [2, 3] in cui i giocatori possono, come parte della dinamica di gioco, depositare un seme nel dispositivo dalla app del gioco. Il seme (ovvero un task) si sviluppa in una pianta solo se la temperatura (letta dal sensore del dispositivo COIN) è giusta per un tempo sufficiente e, se questo avviene, il giocatore può allora recuperare la pianta e ottenere punti bonus.

1.1 Problema e contributo

Rendere le risorse del dispositivo fruibili da molteplici task arbitrari significa doverne condividere le capacità, imponendo politiche e meccanismi che governino la condivisione tra i task. In questa tesi si è affrontata la necessità di individuare un sistema che permetta ad un'architettura con applicazioni eterogenee di arbitrare l'utilizzo da parte dei task di risorse altrettanto diverse.

In particolare, ci si è posto il problema di come allocare risorse inevitabilmente limitate in maniera equa tra i task e come far sì che le allocazioni vengano rispettate, introducendo un costo addizionale per il sistema ragionevole rispetto ai benefici. Gli stessi scopi sono presenti anche in lavori precedenti che considerano però sistemi diversi nelle caratteristiche e limitazioni, soprattutto computazionali, da quelli a cui COIN permette accesso. Infatti, la novità portata dal sistema operativo COIN nell'approccio a questi sistemi, le cui capacità sono di solito rigidamente limitate, richiama all'esigenza di soluzioni nuove, pensate specificamente per questi.

Dopo aver analizzato letteratura e soluzioni esistenti per questo tipo di problematica, congiuntamente alle esigenze specifiche di COIN, questa tesi introduce Queuettery, un sistema che permette l'astrazione delle risorse e la loro condivisione attraverso un approccio probabilistico all'arbitraggio. Queuettery propone una composizione ed estensione originale di algoritmi noti che esplora il trade-off tra l'arbitraggio e la limitata capacità dei dispositivi in una soluzione che, al meglio delle mie conoscenze, non trova per questo applicazioni precedenti.

Dagli esperimenti effettuati, Queuettery consente di ottenere uno scostamento relativo medio, rispetto ad un'allocazione ideale max-min fair [4], inferiore al 7% con un overhead computazionale rispetto alla politica FIFO, *common practice* per sistemi a risorse limitate [5], inferiore al 1% per un numero di task tipico.

1.2 Struttura del documento

Il presente documento di tesi è strutturato, oltre al presente capitolo introduttivo, nel seguente modo:

- Il Capitolo 2 presenta quanto necessario per la trattazione ed analisi della soluzione proposta in questo lavoro di tesi. Vengono descritti il contesto applicativo generale nel quale ci si pone, il sistema operativo COIN su cui si è lavorato e lo stato dell'arte dei sistemi di gestione delle risorse con diversi esempi di soluzioni esistenti.
- Nel Capitolo 3 viene esposto il lavoro di analisi dei requisiti e obiettivi propri del problema e del contesto applicativo. Questi sono fondamentali nello sviluppo del Capitolo 4, nel quale si descrive prima il processo analitico svolto nella selezione delle soluzioni adottate ed infine il sistema di arbitraggio Queuettery concepito in questa tesi.
- Il Capitolo 5 descrive le scelte implementative affrontate nella realizzazione del prototipo di Queuettery. In esso sono riportate le principali procedure e strutture dati implementate e le difficoltà che alcune di esse hanno presentato assieme alle soluzioni adottate.
- Il Capitolo 6 affronta la valutazione di Queuettery, condotta sulla evaluation board STM32F091RC, illustrando gli esperimenti e le misurazioni delle metriche adeguate ad analizzare performance e trade-off del sistema realizzato.
- Il Capitolo 7 riepiloga i risultati ottenuti in questo lavoro di tesi e propone alcune idee per gli sviluppi futuri di COIN.

Capitolo 2

Background

Un gran numero di tecnologie e applicazioni ricadono nel dominio dell'Internet of Things (IoT), tanto da essere difficile darne una definizione comprensiva. Sensori, beacon e dispositivi embedded di ogni genere affrontano negli ultimi anni una diffusione pervasiva, guidata dal crescere della loro economicità e dall'interesse nel rendere *smart* oggetti di comune utilità. Un fattore importante nel successo di questi dispositivi è facilmente individuabile nell'altrettanto capillare diffusione di tablet e smartphone in grado di comunicare essi e presentare un'interfaccia comune ed immediata all'utente.

2.1 Applicazioni

Nel trasporto pubblico [6], l'installazione di beacon può essere utilizzata per guidare i passeggeri all'interno di una stazione della metropolitana per trovare l'uscita più rapida o per fornire informazioni sullo stato del servizio personalizzate. O, ancora, registrare accuratamente la presenza dei passeggeri a bordo di un bus in sostituzione dei tradizionali biglietti e per agevolare l'introduzione di tariffe di pagamento al minuto che corrispondano all'utilizzo.
Dispositivi per l'acquisizione di parametri fisici (ECG, pressione arteriosa, temperatura, etc.) possono essere collegati per tracciare uno stato della salute di un paziente e comunicare tempestivamente con il personale medico [7–9], o per monitorare i progressi ottenuti nell'allenamento di un atleta [10].

Anche i sistemi elettronici di intrattenimento e controllo dei veicoli [11] offrono notevoli possibilità di interazione ed espansione nella comunicazione con i dispositivi connessi, permettendo l'accesso ad un gran numero di informazioni (e.g. posizione, temperatura interna ed esterna, pressione degli pneumatici, consumi). Inoltre, sono all'orizzonte soluzioni che permettano agli stessi veicoli di comunicare con infrastrutture intelligenti (*V2I*) o con altri veicoli (*V2V*) per ricevere informazioni (e.g traffico, condizioni atmosferiche e/o della strada) e incrementare la sicurezza stradale [12–14].

Considerevole successo stanno avendo lampadine e termostati controllabili da smartphone [15, 16], che appaiono come i pionieri di un'imminente automazione delle abitazioni. Altri esempi di dispositivi di questo genere sono: serrature intelligenti [17, 18] che permettono di gestire l'accesso alla casa tramite smartwatch con la possibilità di personalizzare i permessi (ad esempio per restringere l'accesso in determinati orari); impianti di irrigazione automatici che sfruttano la rilevazione della temperatura e dell'umidità dell'aria assieme alle previsioni del tempo ottenute da Internet [19, 20]; sensori di fumo che notificano il proprietario di casa sullo smartphone [21]; e portachiavi che permettono di essere ritrovati tramite app mobile [22].

Il settore ludico, come spesso accade in ambito tecnologico, non è esente dall'essere influenzato da queste innovazioni. Nei *pervasive game* [2, 3] vengono installati dei dispositivi nell'ambiente di gioco, in modo tale che venga creato per il giocatore un collegamento tra il mondo virtuale e quello reale.
Ad esempio, si pensi al recente successo del gioco "Pokemon Go" che sfrutta la realtà aumentata e la localizzazione per ottenere questo risultato. Un'evoluzione in senso pervasive potrebbe prevedere l'esistenza di "Pokestop" fisici costituiti da un dispositivo a cui l'app si connette e da cui può prelevare e/o rilasciare degli oggetti virtuali (per esempio, il giocatore porta un uovo per essere incubato e lo recupera, tornando nel luogo, alcune ore dopo).

I contesti applicativi sono tutt'altro che esauriti e appaiono in costante espansione, da cui la difficoltà a comprimerli in una definizione univoca. Ciò che si trova è la spinta per offrire un numero crescente di risorse, soprattutto d'informazione ma anche d'utilità, alla portata dei dispositivi che si trovano nella tasca di importanti porzioni della popolazione.

2.2 COIN

COIN [1] è un'architettura software che permette l'esecuzione di una porzione di logica applicativa appartenente ad una mobile app su dispositivi connessi, quali *beacon*, sensori, e altri *smart device*. Questi offrono tipicamente interfacce di basso livello e funzionalità fissate che legano gli sviluppatori agli aggiornamenti del produttore e non permettono facile portabilità delle app. Inoltre, funzionalità anche semplici richiedono molteplici ed energeticamente dispendiose comunicazioni wireless, e vengono limitate al tempo in cui lo smartphone e il dispositivo rimangono connessi.

L'idea di COIN è offrire un substrato programmabile che permetta alle mobile app di installare dinamicamente sul dispositivo dei task definiti sotto forma di script. Attraverso questo approccio, COIN è in grado di introdurre significativi risparmi nel consumo energetico dei dispositivi su cui opera, permettendo la riduzione delle interazioni via radio tra mobile app e dispositivo, e di abilitare nuovi scenari applicativi grazie ad un modello che permette la persistenza dei task sul dispositivo.

2.2.1 Architettura

Una nozione fondamentale in COIN è quindi quella di *task*: esso rappresenta una parte di logica applicativa che l'utente trasferisce da una mobile app al dispositivo. Questa è implementata sotto forma di uno script Python, accompagnato da un *manifest* contenente le informazioni relative ai tipi di dato degli input e output e altre informazioni pertinenti (e.g. nome, tempo di esecuzione, deadline).

Giunti sul dispositivo, lo script e il manifest compongono il task, il primo nella sua parte esecutiva e il secondo nella caratterizzazione. Il task è modellato come attore del sistema (vedi Sezione 2.2.2 per una spiegazione del actor-like model) ed è caratterizzato dalla tipologia dei suoi input ed output. L'evoluzione di un task, sintetizzata nella Figura 2.1, passa attraverso alcuni stati:

- **Installed:** il task è stato creato e si trova nel sistema, ma non è al momento attivo.
- **Started:** il task è attivo ed in attesa di ricevere un dato che corrisponda ad uno dei propri tipi in input.

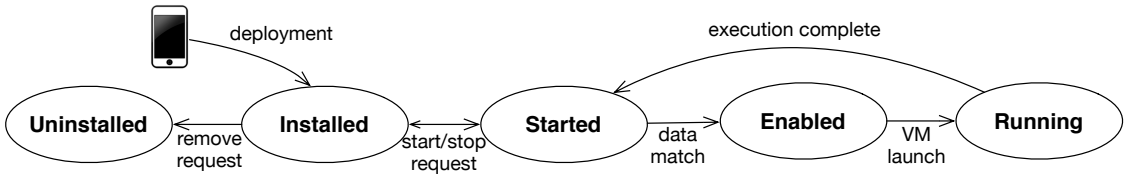


Figura 2.1: Evoluzione dello stato di un task in COIN

- **Enabled:** il task ha ricevuto in ingresso un dato corrispondente ed è in attesa di essere eseguito.
- **Running:** il task è in esecuzione.

L'architettura COIN su cui i task vengono eseguiti è composta da quattro componenti: Task Manager, Broker, Scheduler e Virtual Machine.

Task Manager È il componente che si occupa di gestire i task installati nel sistema. A partire dalla loro installazione, esso esegue le transizioni da uno stato all'altro ed è l'interfaccia attraverso cui si può interagire con i task.

Virtual Machine Si occupa dell'effettiva esecuzione dei task installati dinamicamente dagli utenti. L'uso di una macchina virtuale permette di usare un linguaggio ad astrazione superiore (i.e. Python) rispetto a quello in cui è implementato il sistema e permette di astrarre anche dall'hardware del dispositivo.

Scheduler Gestisce la coda dei task che devono essere mandati in esecuzione sulla Virtual Machine secondo la policy di scheduling Earliest Deadline First, descritta nella Sezione 2.2.4.

Broker Implementa il paradigma di comunicazione publish-subscribe (Sezione 2.2.3) occupandosi dello smistamento dei messaggi tra i task. Nell'implementazione specifica di COIN, gli argomenti sono i tipi di dato ed è possibile per ogni subscriber decidere con quale frequenza ricevere i dati pubblicati.
Ad esempio, un task può sottoscriversi all'argomento "temperatura" su cui pubblica un sensore collegato al sistema alla metà della frequenza di pubblicazione. Il Broker,

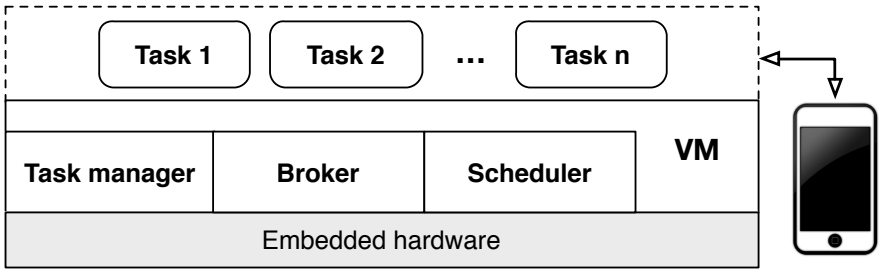


Figura 2.2: Architettura di COIN

ricevendo il dato dal sensore, si occupa di mettere il task nella coda dello Scheduler a ricezioni alterne.

2.2.2 Actor-like model

L'*actor model* [23] è un modello concettuale pensato per l'esecuzione concorrente che permette il disaccoppiamento dei componenti di un sistema. L'attore è un'unità primitiva che può effettuare un ristretto numero di azioni:

- ricevere un messaggio;
- reagire alla ricezione di un messaggio modificando il proprio stato o effettuando un'azione in base ad esso;
- inviare un messaggio.

L'idea è che gli attori possano interagire tra di loro esclusivamente attraverso l'uso di messaggi, non vi sono porzioni di memoria condivisa e un attore non può modificare direttamente lo stato di un altro attore. Nel paradigma originale, la comunicazione avviene tramite un semplice *message passing* (Figura 2.3): i messaggi sono scambiati asincronamente tra gli attori, ognuno dei quali possiede un proprio indirizzo di destinazione che il mittente deve conoscere e specificare al momento dell'invio. Non sono fatte assunzioni sull'ordine di arrivo dei messaggi, essi vengono inviati e alla fine ricevuti del destinatario senza garanzie sul mantenimento della sequenzialità temporale.

In questo modello, le uniche comunicazioni osservabili tra gli attori sono i messaggi, mentre lo stato e l'implementazione interni al singolo attore sono nascosti, permetten-

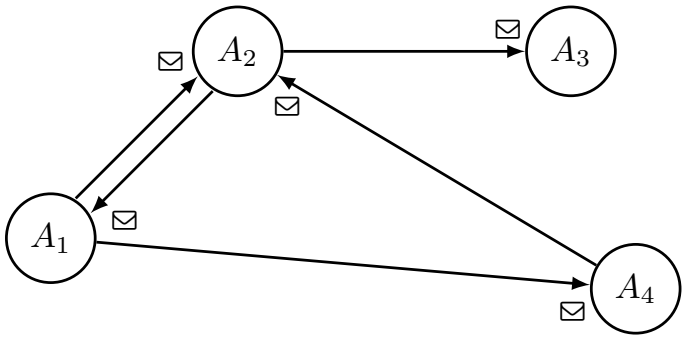


Figura 2.3: Un network di attori che scambiano messaggi con message passing

done l'astrazione e la modularità. Inoltre, questa indipendenza è la chiave per rendere concorrenti i sistemi che adottano questo modello.

2.2.3 Publish-subscribe

Far comunicare direttamente gli attori di un sistema comporta che ognuno di loro abbia la necessità di conoscere, scoprire e manutenere la lista di destinatari dei propri messaggi e di provvedere al loro invio. Questo compito non è banale e rende complesso e oneroso il sistema di comunicazione, specialmente se lo stesso messaggio deve essere inoltrato a più destinatari.

In COIN, il pattern di comunicazione *publish-subscribe* risolve questo problema introducendo un intermediario, il *broker*, che si occupa di smistare i messaggi verso i destinatari, detti *subscriber*. Come mostrato nell'esempio di Figura 2.4, messaggi vengono pubblicati, inviandoli al broker, senza specificare un destinatario ma classificati con un argomento (*topic*). Chi tra gli attori è interessato a ricevere i messaggi di un argomento, lo segnala al broker che tiene una lista di subscriber per ogni argomento ai quali inoltrare i messaggi che gli appartengono.

Questo sistema di comunicazione è usato in COIN anche per introdurre un ulteriore isolamento degli attori del sistema, che scambiano messaggi solamente col broker e non necessitano di essere a conoscenza dell'esistenza di altri attori ma solamente degli argomenti a cui sono interessati e per cui riceveranno i messaggi dal broker.

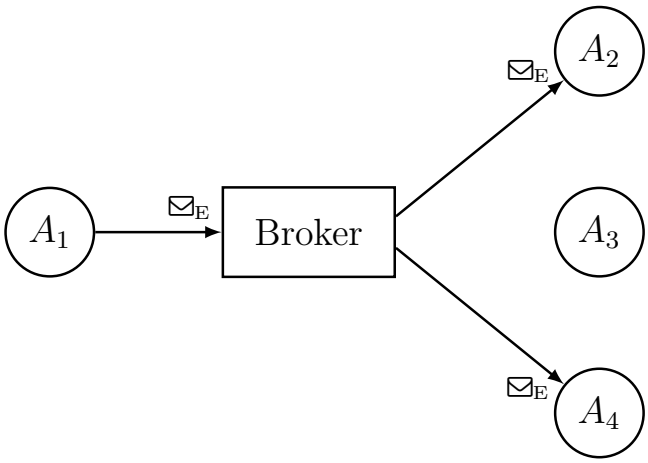


Figura 2.4: Publish-subscribe: A_1 pubblica un messaggio di argomento E, del quale A_2 e A_4 sono subscriber

2.2.4 Earliest Deadline First (EDF)

In termini generici, ci si riferisce con lo *scheduling* ad una classe di problemi che intende determinare un ordinamento temporale delle richieste di utilizzo di una risorsa secondo una determinata politica che rispecchia le priorità del sistema (e.g throughput, qualità del servizio, tempo di risposta). Nel campo dei sistemi operativi, la risorsa per eccellenza, in quanto comune denominatore di tutte le implementazioni, è il processore: il problema di ordinare l'esecuzione di più processi (o task) che desiderano essere eseguiti contemporaneamente è una tematica ampiamente esplorata in letteratura (si veda [24] per uno studio dei principali scheduler per sistemi real-time).

Gli scheduler sono classificati secondo le seguenti definizioni:

- **Ottimale o non ottimale:** uno scheduler è ottimale se, dato un qualsiasi insieme di task eseguibili nel tempo dato, è in grado di produrre uno schedule che permetta di eseguirli.
- **Preemptive or non-preemptive:** uno scheduler preemptive può scegliere di fermare un task già in esecuzione e farlo riprendere successivamente, se questo migliora lo schedule. Viceversa, uno scheduler non-preemptive non ferma mai un task, una volta che questo è in esecuzione.
- **Statico o dinamico:** Gli scheduler statici determinano lo schedule prima

che il sistema sia in esecuzione, mentre quelli dinamici se ne occupano e lo modificano a *run-time*. I primi posseggono il vantaggio di introdurre un overhead molto limitato, ma necessitano di una dettagliata conoscenza a priori delle caratteristiche dei task; i secondi, pagano una miglior flessibilità con un maggiore dispendio computazionale durante l'esecuzione.

In COIN è implementato l'Earliest Deadline First (EDF) [25], uno scheduler dinamico per sistemi real-time la cui scelta è stata fatta per via della sua ottimalità [26], che è garantita a patto che l'utilizzazione complessiva dei task in coda non superi il 100%. I task sono corredati dell'informazione sul proprio tempo di esecuzione e la propria deadline (istante di tempo entro il quale il task deve essere eseguito). La politica EDF prevede di ordinare i task in modo tale che sia data precedenza alla deadline più prossima, ossia la priorità di un task è inversamente proporzionale alla sua deadline. L'implementazione adottata in COIN permette sempre ai task di arrivare a termine dell'esecuzione, ma esistono anche applicazioni preemptive di questo scheduler.

2.3 Fairness

In COIN non esiste, fatta eccezione per il caso specifico dello Scheduler, un componente che si occupi delle risorse del sistema e di come i task installati nel dispositivo possano accedervi. Un sistema di arbitraggio necessita, prima di un *chi*, di un *come* suddividere le risorse, ovvero di determinare quale sia il suo obiettivo ideale e secondo quale politica assegnare porzioni di risorsa ad un utilizzatore, piuttosto che ad un altro.

Si consideri quindi il problema di un'entità che deve stabilire con quale criterio allocare, ovvero dividere, una risorsa di capacità $C_{\mathcal{R}}$ limitata tra N individui il cui unico interesse è soddisfare il proprio desiderio di utilizzo della risorsa (*self-interested*). L'obiettivo dell'allocazione è essere tale che ognuno di questi utilizzatori consideri equa (*fair*) la quantità a lui assegnata. Ma cosa significa essere fair?

L'interesse nel problema della *fairness* precede considerevolmente [27] la nascita delle tecnologie informatiche e trova applicazione in svariati contesti sociali ed economici [28–32]: dal classico taglio della torta, alla decisione su quali pazienti debbano ricevere gli organi da trapiantare, passando per la rappresentanza parlamentare e il

controllo del traffico aereo. In ambito IT, la ricerca di sistemi di allocazione equa riveste grande importanza nel controllo di congestione delle reti di telecomunicazione [33–36], dove è necessario condividere capacità di trasmissione limitate, e nello scheduling dei processi nei sistemi operativi [37–39].

Per il gran numero di approcci e campi di interesse, ma anche per la soggettività del concetto di equità, la fairness ha assunto in questi lavori un gran numero di definizioni e formalizzazioni diverse, di seguito sono riportate le più importanti in campo informatico.

2.3.1 Equal sharing fairness

Si può considerare come la definizione *naïf* di fairness: gli utilizzatori hanno diritto a parti uguali della risorsa, ovvero l'allocazione per ognuno degli utilizzatori è pari a $\frac{C_{\mathcal{R}}}{N}$. Non tenendo conto della domanda, gli utilizzatori che chiedono meno di questa allocazione sprecano capacità mentre quelli con richieste superiori vengono limitati.

Si pensi, ad esempio, di dover gestire l'accesso al computer di una biblioteca con 25 utenti aperta per 10 ore al giorno: l'equal sharing alloca 24 minuti al giorno di utilizzo per ogni utente. A ciascuno dei tre utenti che controllano l'email in 10 minuti sono comunque riservati 14 minuti di tempo aggiuntivi, che rimangono inutilizzati da chiunque. Nel contempo, allo studente che necessiterebbe di un'ora di tempo per fare una ricerca viene negato di superare i 24 minuti a lui attribuiti.

2.3.2 Proportional fairness

Un'allocazione $\mathcal{A}$ è *proportionally fair* se e solo se, per ogni altra possibile allocazione $\mathcal{B}$, si ha che

$$\sum_{i=1}^{N} \frac{b_i - a_i}{a_i} \leq 0 \qquad (a_i \in \mathcal{A}, b_i \in \mathcal{B})$$

Il significato è che, cambiando in qualunque modo l'allocazione rispetto all'assegnamento fair, si ha che la variazione media delle singole allocazioni è negativa.

Ad esempio, supponiamo di dover allocare una capacità $C_{\mathcal{R}} = 5$ a tre utilizzatori aventi un set di domande $\mathcal{D} = \{1, 2, 5\}$. Calcolato il set di possibili allocazioni, nella Tabella 2.1 verifichiamo che l'allocazione {1,2,2} è proportionally fair, mentre non lo è {1,1,3}.

$\mathcal{A} \setminus \mathcal{B}$	{0,0,5}	{0,1,4}	{0,2,3}	{1,0,4}	{1,1,3}	{1,2,2}
{1,1,3}	-1.33	-0.67	0	-0.67	-	0.17
{1,2,2}	-0.5	-0.5	-0.5	0	0	-

Tabella 2.1: Verifica allocazioni con la definizione

La definizione non risulta pratica per il calcolo dell'allocazione fair, in quanto è necessario ottenere tutte le possibili allocazioni e per ciascuna di esse verificare che la definizione sia rispettata. La prima operazione risulta già significativamente onerosa al crescere di N e della dimensione della domanda di ciascuno degli utenti, a questa si aggiunge, nel caso peggiore, un numero di calcoli della definizione quadratico nel numero di allocazioni possibili. In [40] viene presentato un teorema che la rende più facilmente individuabile:

Esiste una unica allocazione proportionally fair ottenuta massimizzando sul set di possibili allocazioni

$$J = \sum_{i=1}^{N} ln(\frac{a_i}{d_i}) \qquad (a_i \in \mathcal{A}, d_i \in \mathcal{D})$$

Riprendendo l'esempio, nella Tabella 2.2 si vede come J raggiunga il suo massimo valore per l'allocazione {1,2,2}, che abbiamo già verificato essere proportionally fair in base alla definizione.

	{0,0,5}	{0,1,4}	{0,2,3}	{1,0,4}	{1,1,3}	{1,2,2}
J	$-\infty$	$-\infty$	$-\infty$	$-\infty$	-1.20	-0.41

Tabella 2.2: Individuazione dell'allocazione proportional fairness tramite teorema

Si nota come questo teorema, pur semplificando significativamente rispetto all'uso della definizione, non risolve l'oneroso problema di individuare tutte le combinazioni di allocazioni possibili.

2.3.3 Max-min fairness

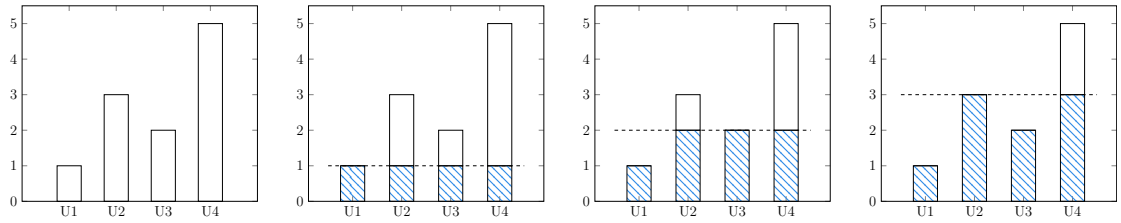
La max-min fairness è definita qualitativamente in [41, 42]:

> *Un'allocazione è max-min fair se non c'è modo di incrementare l'allocazione di un utilizzatore senza decrementare l'allocazione di un altro avente un assegnamento minore o uguale.*

Questa definizione si traduce in una divisione egualitaria, in cui ogni utilizzatore ha eguale diritto alla risorsa. Non tutti gli utilizzatori hanno però, in generale, le stesse domande e alcuni richiedono meno di altri. Una condizione del max-min fairness è che a nessuno venga allocato più di quanto richiede, per cui la parte eccedente la domanda viene distribuita in parti uguali tra gli utilizzatori la cui domanda non è soddisfatta. In questo modo, la priorità è data alla soddisfazione di chi richiede meno risorsa.

In [4] viene presentato un semplice algoritmo per ottenere un'allocazione max-min fair, chiamato *Progressive Filling*.
L'idea è di incrementare, partendo da zero, la quantità di risorsa allocata a ciascun utilizzatore con lo stesso passo (*pace*) finché uno degli utilizzatori non è totalmente soddisfatto. A questo punto, vengono incrementate solo le allocazioni degli utilizzatori parzialmente soddisfatti finché un altro non viene soddisfatto. Il procedimento prosegue in questo modo fino a che la capacità non è esaurita del tutto.
Questo algoritmo è noto anche con il nome di *Water Filling*, per via dell'analogia che viene fatta pensando le domande degli utilizzatori come recipienti nei quali viene versata dell'acqua in modo tale che essi siano tutti riempiti all'orlo (soddisfatti) o fino ad una *water line* che viene man mano innalzata fino ad esaurimento dell'acqua (capacità) disponibile.

Nella Figura 2.5 è illustrato un esempio di allocazione per 4 utilizzatori. Il pace dell'algoritmo è di 1, mentre la capacità totale della risorsa è 9. Inizialmente (Figura 2.5a), le allocazioni degli utilizzatori sono nulle; al primo passo (Figura 2.5b), viene incrementato di una unità l'assegnamento di ciascun utilizzatore. A questo punto, U_1



(a) Inizializzazione (b) Passo 1 (c) Passo 2 (d) Passo 3

Figura 2.5: Esempio di allocazione con Progressive Filling; per ciascuno dei quattro utilizzatori è rappresentata la domanda non soddisfatta (bianco) e l'allocazione (azzurro), la linea tratteggiata indica la *water line*

è soddisfatto, per cui al passo seguente, quando la water line raggiunge il livello 2, la sua allocazione non viene più incrementata (Figura 2.5c). Infine, la Figura 2.5d riporta l'allocazione finale, in quanto non vi è più capacità da distribuire.

2.4 Sistemi resource-aware

Avere una definizione di fairness e un'allocazione teorica non è sufficiente a far sì che tale suddivisione venga rispettata dagli utilizzatori, occorrono meccanismi, policy e astrazioni che consentano di farle rispettare. Dal momento che in questa tesi si desidera rendere COIN un sistema resource-aware, in questa sezione si descrivono alcuni sistemi di questo tipo per avere una panoramica, non necessariamente ristretta all'ambito dei sistemi embedded, sullo stato dell'arte nell'ambito delle architetture per la condivisione di risorse.

2.4.1 ECOSystem

In [43] viene coniato il termine "currentcy", dalla combinazione delle parole *current* ("corrente") e *currency* ("valuta", "moneta"), come astrazione della capacità ed utilizzo della risorsa energetica, attraverso la quale è possibile gestire il consumo dei vari componenti del sistema. Una unità di currentcy rappresenta il diritto di consumare una certa quantità di risorsa entro un determinato periodo di tempo. Il sistema operativo che gli autori hanno realizzato per la sperimentazione di questo modello, chiamato ECOSystem, si pone l'obiettivo primario di amministrare il consumo energetico dei

task affinché la carica della batteria che alimenta il sistema possa raggiungere una durata desiderata.

L'allocazione della currentcy avviene in due fasi. Nella prima fase il sistema determina la capacità totale di currentcy che può essere distribuita ai task del sistema. Il tempo è diviso in *energy-epoch*, all'inizio di ognuna di queste il sistema calcola, in base alla durata totale desiderata e al consumo pregresso, la quantità di currentcy che in essa potrà essere consumata.
Nella seconda fase avviene la distribuzione della currentcy tra i task, che segue proporzioni definite dallo sviluppatore. Ad ogni epoch, ai task viene allocata una porzione proporzionale dell'ammontare di currentcy utilizzabile per quella epoch.

ECOSystem utilizza il modello dei *container* [44] per catturare più facilmente l'attività dei task e dunque monitorare il consumo di energia. I container sono le entità astratte alle quali la currentcy viene effettivamente allocata e dal cui budget vengono sottratte le somme che i task consumano.
L'astrazione permette anche al sistema operativo di rappresentare applicazioni composte di molteplici processi mediante un singolo container, in modo tale da semplificare la gestione.

Risorse diverse hanno consumi diversi, per cui ognuna di esse possiede una politica differente mediante la quale calcola il prezzo del proprio uso. Ad esempio, gli autori presentano una politica per la gestione di un hard disk che tiene conto di quattro tipologie differenti di costo relative ad altrettante operazioni del disco: *spinup, access, spinning* e *spindown*. Infatti, mentre il costo di un access è relativamente semplice da assegnare, a questo vanno aggiunte le altre tre attività che non sono però legate ad una singola scrittura/lettura, ma vengono divise tra i molteplici utilizzatori che ne hanno usufruito. Queste peculiarità non sono condivise da, per esempio, la CPU o la radio, che necessariamente applicano politiche di costo differenti e specifiche.

ECOSystem si concentra sulla gestione dell'energia come risorsa, che non è l'obiettivo di questa tesi, ma fornisce un esempio di uso dell'astrazione, i container, per rendere trasparenti le specificità di risorse diverse che è invece di interesse. Inoltre, fornisce un meccanismo basato sul concetto di budget che, completato dalle epoch, risulta interessante, in quanto molto intuitivo. Il problema delle epoch e dei sistemi a budget in generale, come discuteremo nella Sezione 4.3, è l'imporre un'eccessiva rigidità temporale al sistema.

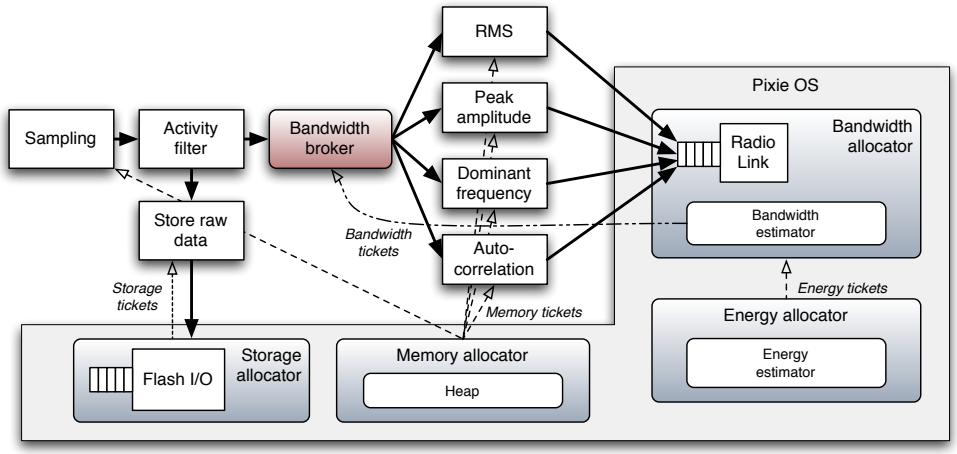


Figura 2.6: Pixie OS, esempio di applicazione per sensori indossabili

2.4.2 Pixie OS

Analogamente a COIN, Pixie OS [45] è un sistema operativo concepito per sensori, la cui specializzazione sono le applicazioni *data-intensive*, per il quale è stato sviluppato un modello di programmazione *resource-aware*. In Pixie è adottata la programmazione dataflow [46], secondo la quale un'applicazione è strutturata sotto forma di un grafo interconnesso di stadi, per permettere al sistema operativo visibilità e controllo delle risorse utilizzate.

Le risorse vengono astratte attraverso l'uso del concetto di *resource ticket*. Un ticket è una tripla $\langle R, c, t_e \rangle$ che rappresenta il diritto di consumare fino a c unità della risorsa R prima dell'istante di tempo t_e. Pixie prevede che sui ticket possono essere effettuate le seguenti operazioni: utilizzo (*redeem*), restituzione (*forfeit*), revoca (*revoke*), unione (*join*) e divisione (*split*).

L'architettura di Pixie (vedi Figura 2.6) definisce due componenti per ogni risorsa: un *resource allocator* e un *resource broker*. Ogni allocator ha tre principali funzioni: stimare a runtime la disponibilità della risorsa, allocare su richiesta ticket per la risorsa e dare accesso alla risorsa nel rispetto del ticket. I broker sono degli stadi di Pixie specializzati nell'intermediare tra l'allocator e gli stadi applicativi nella richiesta e gestione dei ticket, secondo una propria politica. Al contrario degli allocator, i broker non sono necessari per la gestione di una risorsa, ma forniscono un'astrazione di più alto livello per semplificare l'interazione tra gli stadi e la risorsa.

Pixie è l'unico sistema presentato pensato dall'origine per lo stesso campo applicativo di COIN e i suoi ticket presentano un concetto evoluto di moneta rispetto ai sistemi a budget, come ECOSystem (Sezione 2.4.1), che è però diverso, nonostante la sovrapposizione nella denominazione, da quello che verrà presentato per il Lottery (Sezione 2.5.3). Tuttavia, i due componenti che caratterizzano questa architettura, allocator e broker, sono pensati per il modello dataflow e mal si adattano all'actor-like implementato in COIN, che non contempla il concetto di stadio.

2.4.3 Cinder

Cinder [47] è un sistema operativo per dispositivi mobile sviluppato sulla base del sistema HiStar [48] per il quale vengono pensate due astrazioni al fine di adattarlo alla gestione delle risorse: *reserve* e *tap*.

Una *reserve* descrive il diritto d'uso di una determinata quantità di risorsa; quando un'applicazione consuma risorsa, Cinder decrementa la corrispondente reserve del valore consumato. Se la reserve non contiene un valore sufficiente, il sistema operativo impedisce all'applicazione di eseguire l'azione. Tra una reserve e l'altra sono consentiti trasferimenti, deleghe e divisioni a seconda delle esigenze dell'applicazione.

Le reserve da sole non differiscono significativamente dalla currentcy di ECOSystem o dai ticket di Pixie, ciò che caratterizza la soluzione di Cinder sono i *tap*. Essi sono dei thread speciali il cui compito è il trasferimento efficiente di risorsa tra una riserva e l'altra in modo tale da poterne controllare il flusso.
Ogni task possiede una propria reserve che può essere rifornita attraverso uno o più tap da una reserve sorgente (Figura 2.7), l'uso del tap permette al sistema di gestire quanta risorsa mettere a disposizione del task, eventualmente rendendolo proporzionale ad un parametro scelto.

Per evitare che le risorse allocate rimangano inutilizzate, Cinder utilizza un meccanismo di *decay* (decadimento). Le risorse allocate ad una reserve, se non utilizzate, perdono progressivamente valore, fino ad esaurirsi. Le quantità decadute vengono riportate alla reserve sorgente, che può quindi allocarla nuovamente.

I tap e la decay introducono una variante rispetto ai classici sistemi a budget, il task non è in grado volendo di spendere la propria allocazione tutta in una volta e la

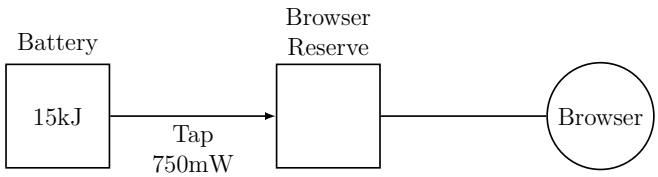


Figura 2.7: Cinder. La reserve di un browser connessa alla reserve sorgente di una batteria da 15kJ attraverso un tap che fornisce 750mW

decay compensa gil sprechi. La controindicazione è che per ogni risorsa ed utilizzatore sarebbe necessario almeno un thread per gestirne l'afflusso di risorsa. In sistemi con capacità computazionali limitate come quelle dei dispositivi che interessano COIN, questi thread avrebbero un impatto troppo alto sulle performance dell'intero sistema per essere giustificato.

2.4.4 Rialto

Rialto [49] introduce un framework per la gestione modulare delle risorse basato sulla negoziazione.

Le risorse del sistema espongono un'interfaccia che permette di conoscere le operazioni che una risorsa mette a disposizione e calcolare in anticipo quali sia il costo di ciascuna operazione (e.g.determinare le richieste per inviare via radio N byte ogni T secondi). I componenti dell'applicazione possono in questo modo calcolare le proprie esigenze e il costo delle operazioni che effettuano, per cui sono a loro volta in grado di esporre una propria interfaccia. Quest'ultima, potendo essere interrogata da altri componenti, consente che l'applicazione sia composta modularmente e l'utilizzo delle risorse venga man mano astratto in moduli e operazioni.

Una volta che l'applicazione ha determinato i suoi requisiti per una risorsa, inizia una negoziazione con il *resource planner*. Questo componente si occupa di arbitrare l'accesso alle risorse secondo una politica data e negozia l'uso delle risorse con gli utilizzatori finché non raggiunge un'allocazione soddisfacente. Una volta determinata l'allocazione, essa viene comunicata al *resource provider*, che si occupa della gestione effettiva della specifica risorsa, perché fornisca all'utilizzatore la quantità assegnata.

Implementare in COIN un meccanismo di negoziazione come Rialto implicherebbe un gran numero di scambi e comunicazioni diretti e specifici tra planner e attore. Questo tipo di relazione tra entità non rientra nel paradigma actor-like, per cui snaturerebbe il sistema; inoltre, i task di COIN non sono pensati per esporre molteplici funzionalità sotto forma di interfaccia.

2.5 Scheduler resource-aware

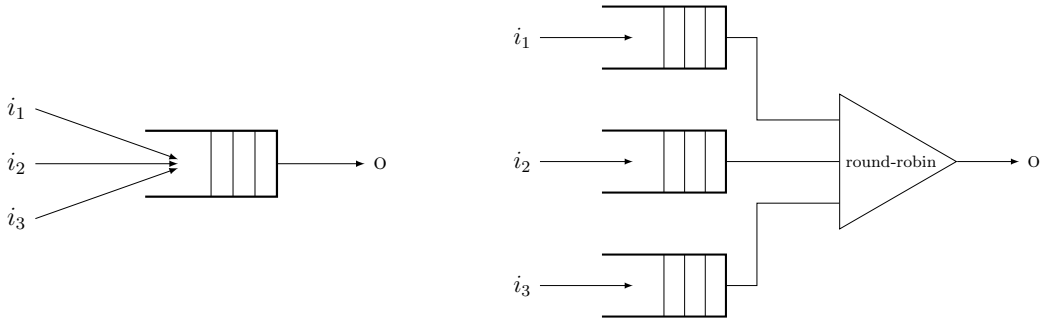
Oltre alle architetture, esiste una classe di algoritmi, quella degli scheduler, il cui obiettivo è di condividere una risorsa attraverso l'uso di politiche che governino l'ordinamento delle richieste. Esiste un gran numero [24, 50, 51] di algoritmi specializzati nello scheduling per ottimizzare l'uso dei processori, in questa sezione vengono illustrati alcuni scheduler la cui generalità permette l'applicazione al caso di risorse diverse.

2.5.1 Fair queueing

Il Fair Queueing nasce nel campo delle telecomunicazioni [52, 53] per il controllo dei flussi di pacchetti che condividono un canale di comunicazione. Il problema che si pone è che i buffer amministrati con comuni code di tipo FIFO non offrono protezione nei confronti di una sorgente di pacchetti che invii pacchetti nel canale ad un rate più alto dei suoi concorrenti.

Per contrastare questo problema viene proposto un algoritmo nel quale ogni sorgente possiede una propria coda (vedi la Figura 2.8 che compara lo schema FIFO con quello Fair Queue). Queste vengono servite con una politica a turni (*round-robin*), per cui ad ogni sorgente con pacchetti in coda viene garantito lo stesso numero di invii. Il risultato è che una sorgente che dovesse incrementare il proprio volume di pacchetti finirebbe solo per allungare la propria coda e dunque danneggiare se stessa.

La fairness implementata da questo algoritmo è quella egualitaria (Sezione 2.3.1), che abbiamo visto essere troppo semplicistica e inadatta ad un sistema che voglia utilizzare in modo efficiente le risorse.



(a) FIFO (b) Fair Queueing

Figura 2.8: Inserimento in coda degli input provenienti da tre sorgenti s_1, s_2 e s_3

2.5.2 Share

Share [37] è uno scheduler nato in ambito accademico per gestire l'uso condiviso di un mainframe da parte del personale universitario e degli studenti. Questo sistema introduce l'idea di unire un concetto comunemente usato negli scheduler, la *usage decay*, con un meccanismo basato sull'assegnazione di budget.

Nella usage decay risiede il concetto che debba esser data maggiore priorità a chi recentemente ha utilizzato meno la risorsa. Ad un uso effettivo della risorsa viene applicato un fattore di decay che ne decrementa la rilevanza proporzionalmente a quanto tempo sia passato da quando esso è avvenuto.

In Share gli utilizzatori vengono schedulati per una qualche risorsa in base a due parametri: i loro *share*, ovvero una misura della quantità di risorsa a cui l'utilizzatore ha diritto, e uno storico degli utilizzi della risorsa. La priorità più alta è data alle richieste d'uso di coloro che hanno i maggiori share e uno storico d'utilizzo minore. Due utenti con lo stesso numero di share e che abbiano nello storico un unico utilizzo di 30 minuti posseggono la stessa priorità. Ipotizziamo però che l'utilizzo del primo utente risalga a 24 ore fa, mentre quello del secondo sia precedente di 48 ore. Dal momento che lo storico segue la usage decay, il secondo utente riceverà maggiore priorità rispetto al primo.

Share si distacca dai meccanismi con budget in quanto gli share non vengono consumati ma sono usati per quantificare, assieme allo storico, il diritto d'uso di un utilizzatore rispetto agli altri che contendono l'uso della risorsa. In contrasto, i sistemi con budget fisso permettono l'uso di una risorsa a patto che il budget del

richiedente sia non nullo, tanto che, nel caso estremo in cui tutti gli utilizzatori lo abbiano terminato, la risorsa rimarrebbe inaccessibile ed inutilizzata.

La difficoltà che si presenta con questo sistema è nel bilanciare la lunghezza dello storico e il peso della decay, che influenzano la precisione dell'algoritmo nell'implementare l'allocazione, con la reattività nell'adeguarsi ad un mutamento nell'allocazione stessa. In COIN un task potrebbe rimanere installato il tempo -pochi secondi- necessario per acquisire un dato dai sensori o per settimane, per cui può risultare complesso, se non impossibile, per lo sviluppatore determinare questo genere di parametri.

2.5.3 Lottery scheduling

Il Lottery Scheduling [54, 55] è un meccanismo probabilistico che fornisce un controllo responsivo e *proportional-share* sulle risorse, dove con proportional-share si intende che i rate di consumo degli utilizzatori sono tenuti proporzionali all'allocazione relativa di ciascuno di essi. Questo meccanismo, realizzato per la gestione delle risorse computazionali, offre un'astrazione facilmente generalizzabile a differenti risorse.

L'analogia con una lotteria si fonda nel concetto di *ticket* ("biglietto"), ognuno di essi rappresenta una determinata quantità di diritti d'uso di una risorsa. Gli autori evidenziano per i ticket le seguenti proprietà:

- *Astratti:* quantificano diritti d'uso per una risorsa indipendentemente dai dettagli della specifica risorsa.
- *Relativi:* la porzione di risorsa che rappresentano dipende dal numero totale di ticket assegnati per la risorsa, per cui questa cambia dinamicamente in conseguenza di una maggiore o minore contesa della risorsa.
- *Uniformi:* permettono di rappresentare risorse eterogenee in modo omogeneo, in modo tale che possano essere effettuate comparazioni quantitative.

I ticket sono per molti versi simili ad una moneta, con la differenza sostanziale che non vengono consumati. Il modello permette che essi possano essere scambiati tra gli utilizzatori anche in cambio di ticket per risorse differenti, attraverso una sorta di tasso di cambio.

Come in una lotteria reale, viene estratto casualmente un ticket vincitore, il cui

possessore riceve il diritto d'utilizzare la risorsa. Dal momento che ogni ticket possiede la stessa probabilità di essere estratto, la probabilità che uno specifico utilizzatore vinca la lotteria è direttamente proporzionale al numero di ticket che possiede. Per questa ragione, viene definito come "*probabilistically fair*".
Il meccanismo è probabilistico, per cui non vi è garanzia di aderenza perfetta all'allocazione attesa ma l'errore decresce con l'aumentare delle estrazioni.

In generale, se ci sono N utilizzatori che competono per una risorsa, e l'i-esimo utilizzatore u_i $(1 \leq i \leq N)$ possiede un numero t_i ticket, vi è un totale di $T = \sum_{i=1}^{N} t_i$ ticket in circolazione. Considerato che il numero di estrazioni vinte da un utilizzatore segue una distribuzione binomiale, la probabilità p_a che l'utilizzatore u_a vinca una particolare lotteria è t_a/T. Dopo n_e estrazioni, il numero di vittorie w_a ottenute da u_a ha un valore atteso pari a $E[w_a] = n_e p_a$ e varianza $\sigma^2_{w_a} = n_e p_a(1 - p_a)$.
Il coefficiente di variazione della proporzione delle vittorie osservate è $\sigma_{w_a}/E[w_a] = \sqrt{(1-p_a)/n_e p_a}$, da cui si deduce che l'accuratezza migliora con $\sqrt{n_e}$.
Inoltre, il numero di lotterie n_a necessario per la prima vittoria dell'utilizzatore u_a segue una distribuzione geometrica con $E[n_a] = 1/p_a$ e $\sigma^2_{n_a} = (1-p_a)/p^2$, per cui il tempo di attesa medio è inversamente proporzionale al numero dei ticket che possiede.

Si nota che ogni utilizzatore con un numero non nullo di ticket ad un certo punto vincerà la lotteria, per cui non è possibile che esso sia totalmente escluso dall'uso della risorsa (*starvation*). Infine, il meccanismo della Lottery è responsivo rispetto alle variazioni nell'allocazione dei ticket, in quanto l'effetto del nuovo assegnamento si riflette immediatamente nella distribuzione di probabilità della seguente estrazione.

Capitolo 3

Obiettivi e requisiti

Nel presente capitolo vengono descritti gli obiettivi e i requisiti che hanno guidato lo sviluppo di questa tesi.

3.1 Obiettivi

La tesi si propone di individuare e sviluppare una soluzione al problema dell'arbitraggio delle risorse che si inserisca come evoluzione e, dunque, nel contesto dell'architettura COIN (vedi Sezione 2.2).

Nel campo dei sistemi embedded e delle reti di sensori una porzione dominante delle applicazioni è realizzata per dispositivi la cui unica fonte di alimentazione è rappresentata da una batteria e il cui collocamento non ne permette una frequente sostituzione. Ne sono esempi significativi il monitoraggio di habitat e animali selvatici [56, 57], dell'attività vulcanica [58] e delle condizioni strutturali dei ponti [59].
In tali condizioni è nata la necessità di preservare l'autonomia dei dispositivi tramite lo sviluppo di tecniche di recupero energetico da fonti naturali (*energy harvesting*), quali sole, vento e vibrazioni [60–62], e la realizzazione di sistemi operativi [43, 63–65] e linguaggi [66, 67] che permettono di ottimizzare il consumo energetico. Per queste ragioni, la letteratura in questi campi si è focalizzata estensivamente sulla gestione di una singola risorsa, l'energia; mentre ha prodotto poche soluzioni, tra le quali citiamo ad esempio PixieOS e Rialto visti nella Sezione 2.4, che permettano di amministrare

la capacità specifica delle singole risorse del sistema (radio, attuatori, sensori, etc.), piuttosto che il loro consumo energetico.

COIN si propone con un'approccio innovativo, permettendo agli utenti di eseguire sul dispositivo dei task, porzioni di logica applicativa della propria mobile app. Tale visione permette al sistema un certo grado di agnosticismo rispetto al contesto applicativo in cui viene utilizzato, mettendo le capacità del sistema a disposizione dell'utente.
Ad esempio, ipotizziamo che il dispositivo sia installato su un bus pubblico e collegato ad un sensore di temperatura e un localizzatore GPS. COIN può essere usato da un app della manutenzione che desidera registrare la temperatura media interna all'abitacolo per individuare eventuali malfunzionamenti e contemporaneamente da un app che usa il GPS per notificare il passeggero quando si avvicina la sua fermata. Questa flessibilità comporta però che molteplici task si possono trovare a desiderare l'utilizzo di una risorsa del sistema con quantità di richieste e concorrenti all'uso ignoti a priori, per cui potenzialmente non è sufficiente la capacità a disposizione.

Lo studio di questa tesi è indirizzato alla gestione delle risorse in COIN, con lo scopo di arbitrarne le richieste in modo tale che porzioni di risorsa siano allocate ai diversi task in maniera equa (*fair*) e in particolare prevenendone la monopolizzazione da parte di alcuni task a discapito di altri. Ad ogni modo, l'analisi e le scelte fatte per raggiungere questo obiettivo non hanno comunque potuto prescindere dal considerare le limitazioni, specialmente energetiche e computazionali, dei dispositivi per i quali questo sistema è concepito.

3.2 Requisiti

Lo studio approfondito degli obiettivi, delle applicazioni e della letteratura (vedi Capitolo 2) ha portato a sintetizzare i seguenti requisiti per il lavoro da realizzare.

(A) Astrazione
L'architettura COIN è concepita per coprire applicazioni diverse non solo per funzionalità ma anche per configurazione. Al dispositivo lo sviluppatore può collegare sensori, attuatori e apparecchiature di vario genere (e.g. ECG, display,

GPS), a seconda del contesto in cui vuole inserirlo.
La natura delle risorse che si desidera gestire è quindi altrettanto varia e il sistema di arbitraggio individuato dovrà rispettare questa generalità, evitando soluzioni eccessivamente indirizzate a risorse o applicazioni specifiche. L'adozione di un'astrazione adeguata permette il riutilizzo dei componenti del sistema e semplifica il lavoro di sviluppo necessario all'introduzione di nuove risorse.

(B) Adattabilità alla dinamica dei task
Il numero di task, e dunque di utilizzatori delle risorse, in COIN è intrinsecamente variabile. Gli utenti possono aggiungere e/o rimuovere task attraverso una mobile app in qualunque momento della vita del sistema. Conseguentemente, al sistema di arbitraggio è richiesta la capacità di adattarsi alla dinamica delle richieste, variando l'allocazione delle risorse per assecondare incrementi e decrementi nella loro domanda.

(C) Fairness
Lo scopo primario di questa tesi è garantire una suddivisione equa delle risorse disponibili tra i task presenti nel sistema.
Il concetto di equità, o fairness, è stato descritto nella Sezione 2.3.3. Sinteticamente, si desidera che ad ogni task utilizzatore sia assegnata una porzione della risorsa secondo un criterio che soddisfi il più possibile la richiesta di ogni task, a patto che questo non avvenga a discapito del diritto di un altro.

(D) Efficienza dell'allocazione
L'assegnazione delle risorse deve avvenire in maniera tale che ne sia massimizzato l'utilizzo e minimizzato lo spreco. Per quanto possibile, si desidera evitare che sia allocato ad un task un numero maggiore di risorse di quel che necessita.

(E) Limitazione dell'overhead
Per ragioni di spazio fisico, consumo energetico e/o costo, nel campo dell'IoT si adoperano dispositivi con significative limitazioni di memoria e capacità computazionali rispetto alle comuni piattaforme odierne, persino quelle mobile. Requisito non funzionale dell'intera architettura di COIN, compreso per cui il sistema di arbitraggio qui trattato, è di minimizzare il consumo di memoria (RAM e flash) e l'overhead computazionale introdotto.
Questo si traduce nella necessità di selezionare soluzioni che siano di compromesso tra il raggiungimento dei requisiti funzionali e la minimizzazione della complessità

introdotta. L'esattezza e/o efficienza dell'allocazione possono quindi essere parzialmente sacrificate per ottenere un guadagno nell'ambito della quantità di memoria e/o tempo di esecuzione necessari per la realizzazione del sistema di arbitraggio.

Capitolo 4

Soluzione

Nel presente capitolo vengono illustrate le ragioni e considerazioni sottostanti alla soluzione adottata.
Il capitolo è diviso in quattro sezioni: la prima tratta le modifiche all'architettura generale di COIN; la seconda e la terza descrivono la scelta dei meccanismi utilizzati nel sistema di arbitraggio; ed infine, la quarta presenta Queuettery, lo schema di gestione delle risorse realizzato per questa tesi.

4.1 Architettura

Nella Sezione 2.2 è descritto come COIN possegga una caratterizzazione architetturale ben definita, la quale si basa essenzialmente sull'applicazione dei paradigmi actor-like e publish-subscribe. Questa architettura è modellata su concetti ben noti e la sua ridefinizione esula dagli scopi della tesi, pertanto si è lavorato con l'intento di introdurre il sistema di arbitraggio sfruttando, e al più adattando con interventi di minima invasività, la struttura di COIN.

4.1.1 Astrazione delle risorse

La prima necessità nel progettare questo sistema è quella di individuare un'astrazione delle risorse. Infatti, il requisito posto di realizzare un sistema di arbitraggio sufficien-

temente generale da adattarsi a risorse di varia natura, ci impone di astrarre prima di tutto le risorse stesse.

A tale scopo, la generalità del paradigma actor-like ha offerto la possibilità di estendere la definizione di attore utilizzata per i task anche alle risorse. Realizzare astrazioni specifiche che si occupassero delle risorse, come i broker e gli allocator di Pixie (Sezione 2.4.2), avrebbe reso più distinta la rappresentazione della risorsa, ma allo stesso tempo complicato l'architettura, necessitando di ulteriori sistemi di comunicazione o interfaccia, e richiedendo una maggiore conoscenza delle risorse da parte dei task.

Le risorse condividono quindi con i task la stessa struttura di alto livello, caratterizzata dalla definizione dei dati di input e output e da un metodo `run`. Quest'ultimo è l'elemento che permette inoltre di rendere trasparente la differenza tra i task e le risorse: nei primi, il metodo racchiude l'esecuzione -mediante l'interpretazione offerta dalla Virtual Machine descritta nella Sezione 2.2.1- dello script Python dell'utente; nelle seconde, include tutte le procedure necessarie all'uso della risorsa specifica.
Una risorsa è pertanto a tutti gli effetti un task offerto nativamente con cui gli altri task possono comunicare secondo il modello actor-like. Eliminando la distinzione, in termini d'astrazione, tra task e risorse (si noti che, per chiarezza nell'esposizione, continueremo comunque ad usare termini distinti) è possibile usare gli stessi componenti (i.e. broker, task manager, scheduler) senza che questi aumentino in complessità ed evitando di doverne creare di nuovi.

Questa scelta offre anche il vantaggio di rendere COIN facilmente adattabile ad applicazioni con configurazioni diverse. Infatti, l'astrazione permette allo sviluppatore di introdurre una nuova risorsa senza intervenire sul core del sistema, ma creando un task nativo il cui metodo `run` racchiude le implementazioni specifiche della risorsa e installandolo nel sistema tramite il task manager.

4.1.2 Messaggi e code in ingresso

In COIN la comunicazione tra task avviene secondo il paradigma publish-subscribe (vedi Sezione 2.2.3), lo stesso è applicato ora alle comunicazioni tra task e risorse, ovvero alle richieste. In termini pratici non vi sono differenze sostanziali, ma concettualmente, mentre nel modello precedente i topic rappresentavano esclusivamente i tipi di dato, con

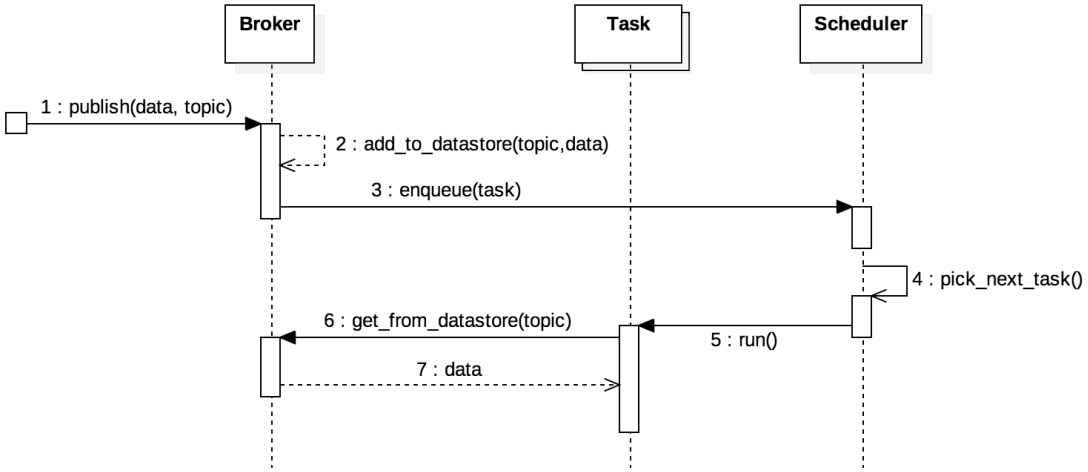
l'introduzione delle risorse anche le richieste di un tipo di dato o di azione divengono argomento di pubblicazione.

Ad esempio, in presenza di un localizzatore GPS come risorsa del sistema, i task che lo desiderano si sottoscrivono al topic "geolocalizzazione" per ricevere la posizione. Al contempo, il GPS è sottoscritto ad un topic "richiesta-geolocalizzazione" su cui i task possono pubblicare per inoltrare al GPS le proprie richieste di localizzazione. Naturalmente, non tutte le risorse necessitano di rispondere alle richieste pubblicando un dato, per cui la richiesta può anche essere per un tipo di azione. Nel caso di un attuatore meccanico, esso sarà sottoscritto al topic "richiesta-movimento" per il quale riceve i messaggi che gli indicano quali movimenti effettuare ma senza pubblicare alcunché.

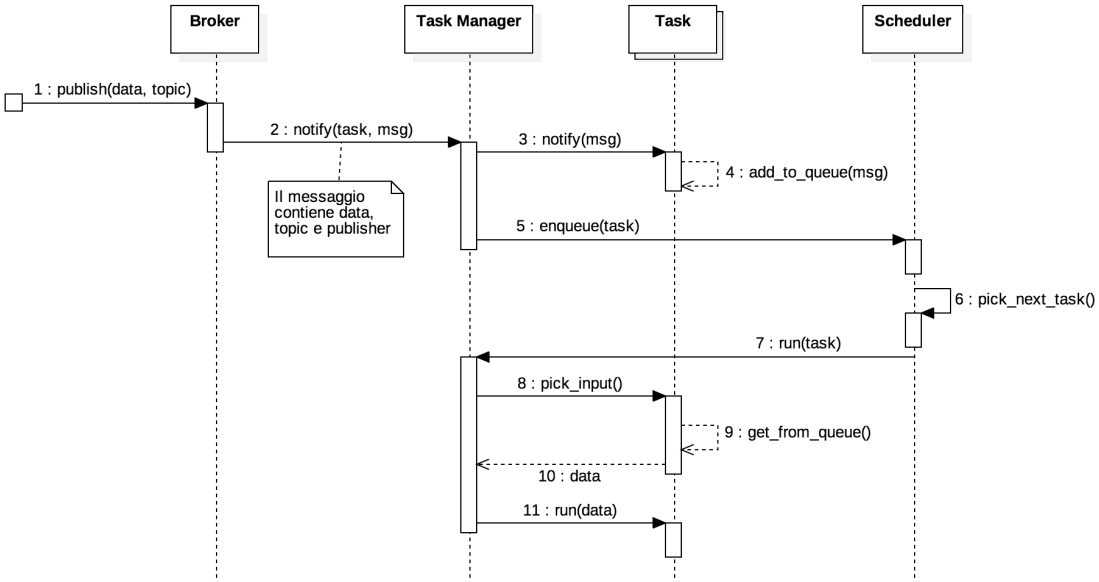
Le richieste giungono quindi alla risorsa sotto forma di messaggi inoltrati dal Broker. Il compito del sistema di arbitraggio è quello di valutarne il contenuto e, in base alla propria politica, decidere quali e quante richieste possano essere soddisfatte. Nella Figura 4.1a è schematizzato il processo di pubblicazione ed esecuzione di un dato in COIN: il Broker riceve il dato, lo inserisce nel data store del topic e accoda immediatamente i sottoscrittori allo Scheduler. Il data store è un buffer FIFO comune a tutti i task che conserva i dati pubblicati per un topic e da cui i task prelevano gli input per l'esecuzione. Nessuna informazione aggiuntiva è conservata sui dati, ad esempio su chi l'abbia pubblicato; inoltre, essendo il buffer comune, i task non posseggono una visione individuale e una gestione dei dati al proprio ingresso.

Queste condizioni non permettono ad un task, e dunque ad una risorsa, di poter operare delle decisioni sui dati da ricevere e quindi realizzare su di essi l'arbitraggio. Per queste ragioni, ho introdotto un concetto di messaggio più strutturato, il quale includa oltre al dato l'informazione su chi l'abbia pubblicato, che nel caso delle risorse è il richiedente. I messaggi vengono notificati ad un task dal Broker con un nuovo metodo `notify`, che si occupa, appoggiandosi al Task Manager, di inserire il messaggio in una coda di input del task. Per semplificare il ruolo del Broker, il Task Manager diventa inoltre responsabile di gestire l'accodamento dei task allo Scheduler.
Nella Figura 4.1b è schematizzato questo procedimento assieme al ruolo di un altro nuovo metodo del task, il `pick_input`. Il suo compito è quello di scegliere tra i messaggi in coda quale debba essere scelto per essere utilizzato come input dell'esecuzione.

Le politiche implementate da `notify` e `pick_input` sono fondamentali nel processo



(a) Prima



(b) Dopo

Figura 4.1: Sequence Diagram della pubblicazione ed esecuzione di un dato

decisionale di selezione e ordinamento dei messaggi che verranno elaborati da un task. E, nel momento che per messaggi intendiamo richieste per una risorsa, essi sono gli elementi caratterizzanti del sistema di arbitraggio. Ad esempio, implementando `notify` come inserimento in fondo alla coda e `pick_input` come prelievo dell'elemento in testa alla coda, la politica di arbitraggio realizzata sarà corrispondente ad una FIFO.

4.2 Allocazione

Il manifesto di ciascun task, tra le altre cose, riporta la sua domanda per una risorsa. Questa dichiarazione rappresenta la condizione ideale per il task, ma è interpretata dal sistema in un'ottica *best-effort*, piuttosto che come una promessa di allocazione. Infatti, per via della variabilità nella domanda, è ignoto a priori quale sarà la capacità di risorsa allocabile ad ogni task nei vari momenti della sua vita nel sistema, per cui non è possibile ipotizzare una negoziazione tra task e sistema sulla domanda.

Il sistema, col mutare del numero dei task e conseguentemente delle richieste, deve allocare ad ogni richiedente una quantità di risorsa senza che la somma degli assegnamenti ecceda la capacità totale. Ovvero, data una risorsa $\mathcal{R}$ dotata di capacità $C_{\mathcal{R}}$ e un set di N task $\mathcal{T} = \{t_1, t_2, ..., t_n\}$ ciascuno avente una domanda d_i $(1 \leq i \leq N)$ per la stessa, l'allocazione $\mathcal{A}_{\mathcal{R}} = \{a_1, a_2, ..., a_n\}$ deve essere tale per cui

$$\sum_{i=1}^{N} a_i \leq C_{\mathcal{R}}$$

È facilmente intuibile come il problema nel calcolare un'allocazione, così come la necessità di un arbitraggio, nasca nel momento in cui $\sum_{n=1}^{N} d_n > C_{\mathcal{R}}$, ossia quando la domanda supera la disponibilità. Il criterio con cui viene composto $\mathcal{A}_{\mathcal{R}}$ influenza il comportamento del sistema e deve rispondere ai requisiti di fairness ed efficienza.

Nella Sezione 2.3 è descritto come la letteratura su questo problema sia ampia ed evoluta nei più svariati contesti. Criteri basati su allocazioni eguali hanno un duplice svantaggio: riservano risorse in eccesso agli utilizzatori che richiedono meno dell'allo-

cazione $\frac{C_{\mathcal{R}}}{N}$, causandone il mancato utilizzo e quindo lo spreco, e conseguentemente limitano ingiustificatamente chi chiede di più. La condivisione puramente proporzionale, d'altro canto, nel caso di richieste fortemente disomogenee, risulterebbe troppo stringente per i task che richiedono meno risorse, rischiando di non concedergliene affatto (*starvation*). Si nota anche che, nonostante questo lavoro poggi sull'assunzione di base dell'onestà delle dichiarazioni fatte dai task, questa proporzionalità indurrebbe gli sviluppatori dei task a mentire al rialzo sulla propria domanda, per ottenere di più. Infine, la proportional fairness (Sezione 2.3.2) presenta un'eccessiva complessità nella definizione e nel calcolo (i.e. determinazione del set di allocazioni possibili) che non la rendono direttamente applicabile.

L'obiettivo di evitare che le risorse vengano monopolizzate da task con alte richieste è invece efficacemente raggiunto dalla *max-min fairness* descritto nella Sezione 2.3.3. Tale paradigma, a dispetto di altre soluzioni presentate, è un approccio alla condivisione equa facile e largamente riconosciuto.
Il Progressive Filling (vedi Algoritmo 1) permette di ottenere un'allocazione max-min algoritmicamente con una complessità temporale $O(^{C_{\mathcal{R}}}/_{pace})$, la quale è controllata dallo sviluppatore, che stabilisce la granularità dell'allocazione (*pace*), e non è dipendente dal numero o dalle domande dei task. Inoltre, l'allocazione è valida per tutto il periodo in cui le condizioni di domanda rimangono invariate, ovvero finché non vengono aggiunti o rimossi utilizzatori della risorsa, che è plausibile assumere essere operazioni con frequenza relativamente bassa.

4.3 Arbitraggio

Calcolate le allocazioni delle risorse, si pone il problema di individuare un meccanismo che si occupi di implementare le allocazioni, ovvero di imporre che ai task sia dato effettivamente accesso alla quantità assegnata a ciascuno.

Nella Sezione 2.4 abbiamo visto come sia diffusa l'idea di interpretare l'allocazione come un budget che il task ha a disposizione. Ad ogni richiesta, l'ammontare di risorsa utilizzata viene scalata da questo budget finché questo non esaurisce e all'utilizzatore viene impedito di utilizzare la risorsa. Il fatto di avere un budget non permette di regolare il rate a cui le richieste vengono soddisfatte, per cui un task potrebbe

Algorithm 1 Progressive Filling pseudocode

```
procedure FILLING(𝒯, C_ℛ, pace)
    𝒯 = {t_1, ..., t_n} ← set of N tasks
    d_i ∈ 𝒟 ← demand of the i-th task
    a_i ∈ 𝒜 ← resource allocated to the i-th task
    c ← allocated capacity

    for each a_i ∈ 𝒜 do
        a_i ← 0

    while c + pace ≤ C_ℛ  &&  size(𝒯) > 0 do
        for each t_i ∈ 𝒯 do
            if c + pace > C_ℛ then // break if another pace step would overcome capacity
                break

            if a_i + pace > d_i then // the task demand's been satisfied
                𝒯 ← 𝒯 − {t_i}
            else
                a_i ← a_i + pace
                c ← c + pace
    return 𝒜
```

spendere tutta la propria disponibilità in un unico burst e rimanere bloccato per tutto il resto della propria vita nel sistema. Inoltre, dal momento che in COIN il tempo di vita nel sistema di un task non è fissato e può variare significativamente tra l'uno e l'altro, task più longevi avrebbero, a parità delle altre condizioni, budget più alti per il semplice fatto che gli deve essere sufficiente per un tempo più lungo. Questo, assieme alla mancanza di regolazione del rate, significa che un task particolarmente longevo potrebbe disporre del budget per monopolizzare per un lungo periodo consecutivo la risorsa, eventualmente anche eccedente l'intera vita di altri task che non otterrebbero così accesso alla risorsa.

In ECOSystem, descritto nella Sezione 2.4.1, viene introdotta l'idea di legare i budget ad un riferimento temporale comune, l'*epoch*. I budget vengono ricalcolati e ripristinati ad ogni nuova epoch, a prescindere dal loro utilizzo effettivo. La scelta della durata di ogni epoch diviene critica: infatti, epoch troppo lunghe riproporrebbero problematiche analoghe a quelle incontrate nei meccanismi che non le adottano. D'altro canto, nell'ipotesi che si cada su periodi brevi, alcuni task potrebbero non riuscire ad

usare la risorsa pur essendoci nel complesso la disponibilità.

Per comprendere quest'ultima considerazione, si consideri il caso di voler arbitrare con questo sistema la trasmissione di una ipotetica radio capace di inviare 50 Byte/ms. Nel sistema sono presenti due task: il primo (T_1) invia una richiesta da 75 Byte ogni $3ms$, mentre il secondo (T_2) inoltra una richiesta da 10 Byte ad intervalli di 1ms. Nell'ipotesi di impostare l'epoch ad $1ms$, il progressive filling soddisfa la richiesta di T_2, allocandogli 10 Byte, e assegna il resto della capacità per epoch a T_1, ossia $(50 - 10 =)$ 40 Byte.
Non potendo cumulare il budget attraverso le varie epoch (poiché facendolo si vanificherebbe l'uso delle stesse), T_1 si trova nella condizione di non aver mai un diritto all'uso della risorsa che gli permetta di inviare il proprio pacchetto. Questo pur non essendo la capacità della risorsa sfruttata a pieno: infatti, considerando un periodo di $3ms$, il suo utilizzo complessivo da parte dei due task è di $3 * 10Byte + 75Byte < 3 * 50Byte$. Naturalmente, non è desiderabile che il sistema di arbitraggio neghi ad un task l'accesso ad una risorsa esclusivamente per via delle sue stesse limitazione, piuttosto che per una necessità di gestione della capacità.

I meccanismi puramente basati su budget appaiono quindi sotto diversi aspetti eccessivamente rigidi e si è ritenuto non siano compatibili con i requisiti di dinamicità di questa tesi. Lo scheduling è da tempo oggetto di ricerca nell'informatica e innumerevoli risultati son stati prodotti negli anni, alcuni dei quali riportati nella Sezione 2.5, proprio per la capacità delle sue metodologie di gestire la condivisione tra utilizzatori con caratteristiche eterogenee e dinamiche. Una significativa porzione di questi è focalizzata sulla condivisione dell'uso dei processori e non facilmente applicabile a risorse generiche. Il Lottery Scheduling descritto nella Sezione 2.5.3 è un approccio alla gestione delle risorse che combina la flessibilità di uno scheduler con le capacità di astrazione della moneta dei sistemi a budget. L'allocazione delle risorse, rappresentata dai ticket, è utilizzata come distribuzione di probabilità e l'utilizzatore della risorsa è il vincitore di una lotteria tra i task che fa uso di tale distribuzione per l'estrazione. I ticket non vengono consumati ma permangono definiti attraverso le varie estrazioni, finché non è necessario effettuare una nuova allocazione (e.g. installazione o rimozione di un task).
Ogni ticket rappresenta una quantità atomica di risorsa secondo una equivalenza definita dallo sviluppatore e dipendente dalla natura della risorsa stessa. Ad esempio,

un ticket può equivalere al payload di un pacchetto BLE (20 Byte), a 10 secondi d'uso di un attuatore meccanico e così via.

La conversione da quantità di risorsa a ticket permette di avere un'astrazione facilmente applicabile alla disponibilità della stessa, nonché alle domande e richieste dei singoli task, che può essere usata per l'intero sistema di arbitraggio. Nell'interpretazione del Lottery in questa tesi, la capacità della risorsa e le domande di ogni task sono espresse rispetto ad un arco temporale comune, su ispirazione delle *epoch*, il quale svolge il compito di unità di riferimento. Questo permette al sistema di arbitraggio di essere indipendente dalla periodicità dei singoli task senza introdurre una rigidità temporale nel sistema. Infatti, essendo i ticket una forma di budget *leggero*, il caso descritto per le epoch in cui un task le cui domande possedevano una periodicità superiore all'epoch veniva bloccato non si presenta: possedendo dei ticket, il task viene ad un certo punto estratto e accontentato.
Una volta che le domande e capacità sono convertite in ticket, ad ogni task che desideri utilizzare la risorsa viene assegnato un determinato numero di ticket attraverso il meccanismo di allocazione descritto nella Sezione 4.2.

4.4 Queuettery

Il paradigma del Lottery, come presentato dall'autore in [54], non contempla la gestione di molteplici richieste da parte dei partecipanti alla lotteria, in quanto assume che gli utilizzatori rimangano bloccati in attesa di poter utilizzare la risorsa. Il nostro sistema, sulla base del paradigma actor-like, non può basarsi su questa assunzione.

Nella Figura 4.2 è rappresentato lo schema di gestione concepito per questa tesi, il *Queuettery*. Il nome riflette i due algoritmi che ne costituiscono la principale ispirazione: il Fair Queueing (vedi Sezione 2.5.1) e il Lottery Scheduling.
La risorsa possiede una coda in ingresso per ogni task utilizzatore. Il metodo `notify` (Algoritmo 2) inserisce il messaggio ricevuto tramite il Broker in fondo alla coda relativa al mittente. Tali code sono implementate come delle *circular queue*, le quali sono delle FIFO con lunghezza limitata che, quando arrivano a saturazione, fanno spazio a nuovi elementi che si aggiungono in fondo alla coda, eliminando l'elemento in testa.

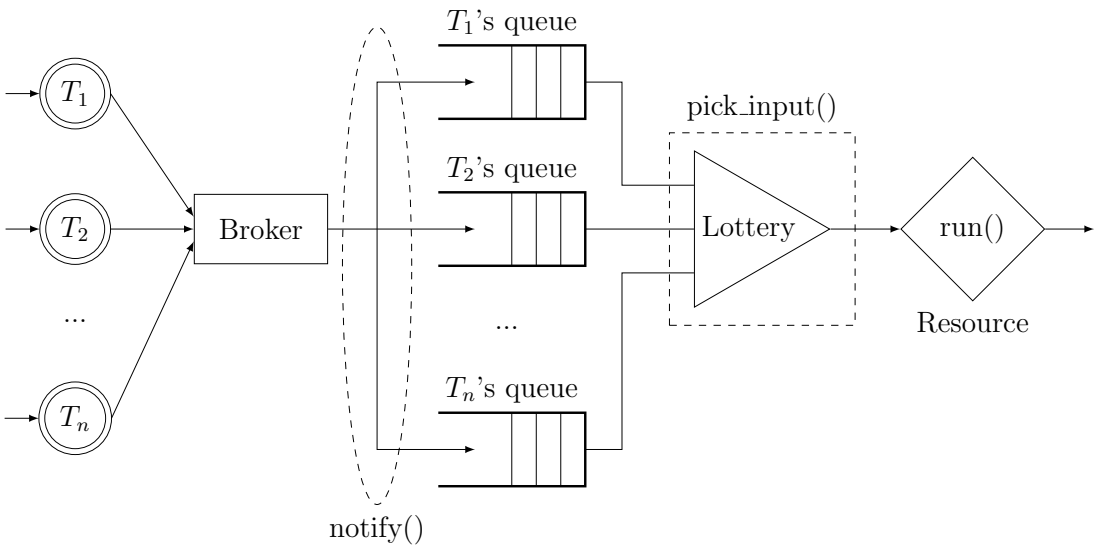


Figura 4.2: Schema di gestione delle risorse in Queuettery

Queste code si riempiono quando la risorsa preleva da esse meno richieste di quante ne vengano inserite; in altre parole, quando l'afflusso alla coda è superiore alla capacità della risorsa o all'allocazione relativa a tale coda. Le richieste che vengono eliminate per via della saturazione della coda, sono quindi quelle che la risorsa non è in grado di accontentare.

Algorithm 2 notify(message) for lottery pseudocode

```
procedure NOTIFY(MESSAGE)
    task_queues ← table of queues

    sender_q ← get_queue(task_queues, message.sender) // get sender's queue

    if sender_q.size ≥ MAX_TASK_Q_SIZE then
        remove_queue_el(sender_q, 0) // remove the head element

    append_queue(sender_q, message)
```

Il compito del metodo `pick_input`, delineato come pseudocodice nell'Algoritmo 3, è quindi quello di scegliere quale tra queste code servire, ovvero quale task andrà ad utilizzare la risorsa. Questa scelta, che nel *fair queueing* avviene con un criterio a turni (*round-robin*), avviene in Queuettery utilizzando una lotteria. Nella procedura `lottery_extract_winner` viene estratto, utilizzando un generatore di numeri pseudocasuali (PRNG), un numero compreso tra 1 e il numero di ticket in cui è stata

convertita la capacità $C_{\mathcal{R}}$. Il task a cui è stato assegnato nella fase di allocazione il ticket corrispondente al numero estratto è il vincitore della lotteria. La richiesta in testa alla coda del vincitore viene quindi inoltrata in input alla procedura `run` della risorsa.

Algorithm 3 pick_input() for lottery pseudocode

```
procedure PICK_INPUT()
    task_queues ← table of queues

    winner ← lottery_extract_winner()
    winner_q ← get_queue(task_queues, winner) // get winner's queue

    return get_queue_el(winner_queue, 0) // return winner's queue head element
```

Nella versione descritta finora, il meccanismo sottintende l'assunzione che ogni richiesta equivalga precisamente ad un utilizzo della risorsa pari al valore di un ticket. Ipotizzando che una richiesta possa superare tale quantità, il task che l'ha realizzata otterrebbe una sopravvalutazione del proprio ticket, che gli permetterebbe di accedere con una vincita all'equivalente di più ticket. Il discorso può essere fatto analogamente per richieste che sottovalutano il ticket utilizzandone una frazione.

Utilizziamo come esempio di risorsa un attuatore meccanico il cui ticket equivalga a 10 secondi d'uso. Ad un task T_1 che effettua una singola richiesta per 20 secondi d'uso, viene garantito l'accesso a quei 20 secondi, equivalenti a due ticket, con una sola vincita. I ticket di T_1 acquistano così una valore maggiore (doppio) dei ticket di un secondo task T_2 che effettua nello stesso periodo due richieste da 10 secondi, ma per le quali deve vincere due volte.

Mentre possiamo assumere che la granularità dei ticket scelta dallo sviluppatore sia sufficiente da rendere l'effetto della sottovalutazione irrilevante, imporre una totale uniformità nell'entità delle richieste appare eccessivamente limitante per il sistema. Queuettery permette di creare richieste dal valore di molteplici ticket, semplicemente indicandone il numero nel messaggio di richiesta. Il metodo `pick_input` si occupa di selezionare la richiesta di un task solo quando questo sia stato estratto come vincitore un numero di volte pari al valore in ticket della richiesta. Riprendendo l'esempio dell'attuatore meccanico il cui ticket equivale a 10 secondi d'uso, il task T_1 effettua una richiesta r per 20 secondi d'uso indicando che essa vale 2 ticket. La prima volta

che T_1 viene estratto come vincitore, la richiesta r non viene inoltrata all'attuatore e `pick_input` provvede ad effettuare una nuova estrazione della lotteria. La seconda volta, non necessariamente consecutiva, che T_1 viene estratto come vincitore la richiesta r viene finalmente soddisfatta.

Capitolo 5

Prototipo

Il prototipo di Queuettery è stato sviluppato in C per microcontrollori a 32-bit di classe Cortex M, sulla base del preesistente prototipo di COIN.
Nella Sezione 4.1 si è detto che la modifica dell'architettura principale di COIN non fosse tra gli obiettivi di questa tesi. Questa ortogonalità si è naturalmente riflessa nell'implementazione del prototipo, il cui lavoro è consistito principalmente nell'introduzione di alcuni nuovi componenti e nell'adattamento degli esistenti. Di seguito sono descritte le modifiche ed aggiunte implementative che sono state eseguite.

5.1 Task

L'adattamento che ha avuto maggior impatto negli altri componenti riguarda la scelta di estendere l'astrazione dei task alle risorse (Sezione 4.1.1). La Virtual Machine (vedi Sezione 2.2.1 per l'architettura di COIN) è implementata da un interprete Python ridotto per microcontrollori, PyMite [68]. Nel prototipo originale, ogni task è mappato come un thread di questo interprete, per cui lo scheduler EDF e il task manager operano su questi thread PyMite.

È stata introdotta una struttura dati `task_struct` distinta che permettesse di spezzare la corrispondenza univoca task-thread. La struttura racchiude essenzialmente le stesse informazioni che prima erano contenute nel thread (tipo degli input/output, tempo di esecuzione, deadline, code, etc.). La differenza sostanziale risiede nel fatto

Listato 5.1: Frammento (semplificato) dell'implementazione del metodo run() per task Python

```
void py_run(task_struct *this_task) {
   // get the thread associated with this task
   pthread = py_task_get_thread(this_task);
   // call Pymite to interpret and execute the Python task
   py_interpret(pthread);
}
```

Listato 5.2: Frammento (semplificato) dell'implementazione del metodo run() per invio della radio

```
void bt_run(task_struct *this_task) {
   // get the message the message that has been picked as the current
      input by the policy (i.e FIFO, lottery)
   msg = this_task->current_input;
   // the message contains the data and the data type, update the BLE
      characteristic accordingly
   ble_update_char(msg);
}
```

che l'esecuzione (`run`) non è più fissata implicitamente nell'avvio dell'interprete ma, dato che il task può implementare operazioni differenti, astratto come chiamata ad una funzione implementata specificamente. I Listati 5.1 e 5.2 esemplificano due implementazioni differenti del metodo `run` per mostrare come l'astrazione del task possa essere implementata per funzionalità diverse: la prima esegue come in precedenza un thread PyMite; mentre la seconda si occupa invece dell'operazione di invio di una risorsa radio BLE.

5.2 Messaggi e code

L'altra modifica che ha richiesto un adattamento è l'introduzione delle code e dei messaggi descritto nella Sezione 4.1.2. Nel prototipo originale, il Broker riceve la pubblicazione di un dato e lo salva nel data store del topic in cui è stato pubblicato. A questo punto, cicla sui sottoscrittori del topic e li inserisce immediatamente nella coda dello Scheduler; nel momento in cui i task vengono eseguiti, questi possono accedere al data store e utilizzare il dato che desiderano. Il data store è comune a

Listato 5.3: Struttura dati di un messaggio

```
struct msg_struct {
   /** Name of the publish-subscriber topic */
   char topic[TOPIC_NAME_MAX_SIZE];
   /** Name of the publisher */
   char publisher[TASK_NAME_MAX_SIZE];
   /** Pointer to the data content of the message */
   uint8_t *data;
};
```

tutti i sottoscrittori e non presenta informazioni sull'ordine d'arrivo o su chi abbia pubblicato il dato.

Nel nuovo prototipo la `msg_struct` (Listato 5.3) svolge il ruolo di messaggio di notifica ai task della presenza di un nuovo dato per un topic, portando con sè anche l'informazione di chi l'abbia pubblicato. Il Broker continua ad utilizzare il data store, in quanto consente di evitare la duplicazione del dato tra i vari sottoscrittori (ad esempio, non è desiderabile che il dato "30°" sia salvato distintamente in memoria per ognuno dei sottoscrittori del topic "temperatura"), ma non si occupa più direttamente di inserire il task nello Scheduler. Invece, crea un messaggio contenente un puntatore all'area di memoria del data store e, tramite il Task Manager, lo notifica ai sottoscrittori, i quali lo inseriscono nella propria coda di ingresso.

Si noti che il messaggio inviato dal Broker ai sottoscrittori è identico per ciascuna pubblicazione. Anche in questo caso, per ottimizzare il consumo di memoria, si evita la duplicazione e ad essere messo nelle code non è il messaggio per intero ma un puntatore a questo. Il costo di questo risparmio è però il problema di determinare quando un messaggio possa essere cancellato dalla memoria. Ad esempio, nel momento in cui un messaggio è stato ricevuto e consumato da un sottoscrittore, lo stesso messaggio potrebbe ancora trovarsi nella coda di un altro. Non è possibile quindi decidere *localmente* di liberare la memoria, ovvero senza andare a verificare le code di tutti i sottoscrittori.

Questo genere di problema è molto comune ed è noto come *garbage collection* [69]. PyMite utilizzata nel prototipo implementa un garbage collector di tipo *mark-sweep*, il cui funzionamento è distinto dalle due fasi da cui prende il nome:

- **Mark:** gli oggetti in memoria sono collegati in un grafo che possiede una root (radice); quando un oggetto ne utilizza un altro, mantiene un puntatore ad esso. L'algoritmo, partendo dalla root, segue i puntatori e contrassegna (mark) tutti gli oggetti raggiungibili attraverso questi collegamenti.
- **Sweep:** in questa fase gli oggetti contrassegnati sono considerati *vivi*, ovvero utilizzati da qualcuno. La sweep elimina tutti gli oggetti che non sono stati contrassegnati, in quanto non sono più necessari a nessuno.

PyMite mette anche a disposizione due strutture dati: le liste e i dizionari. Il vantaggio nell'uso di queste due strutture, oltre ad essere già incluse nel codice della VM e quindi evitare nuove implementazioni, è che sono già predisposte per il garbage collector. Liste e dizionari sono oggetti raggiungibili dalla root, per cui ciò che contengono (sotto forma di puntatore) diviene raggiungibile a sua volta.

Il problema di memoria è stato quindi risolto grazie al garbage collector, utilizzando le liste di PyMite per le code. Un messaggio in coda per i task T_1 e T_2 che viene consumato (tolto dalla coda) da T_1, non viene più puntato dalla sua coda ma rimane raggiungibile attraverso la coda di T_2, per cui non viene rimosso. Quando anche T_2 consuma il messaggio, allora non verrà più contrassegnato dalla mark e verrà correttamente rimosso.

I dizionari (anche noti come *hash table*) sono delle tipologie particolari di lista che permettono di associare un valore ad una chiave. Queste strutture servono nel nostro caso ad associare un task (chiave) alla propria coda (valore) del Lottery. Quando un messaggio di un determinato task T_1 viene notificato, viene ricercata nel dizionario (lookup) la sua chiave, la quale punta direttamente alla coda ad esso associata e a cui viene aggiunto il nuovo messaggio. Analogo procedimento avviene quando T_1 è estratto dal Lottery e deve essere preso un messaggio dalla sua coda.

5.3 Ciclo di vita del task

Riepiloghiamo ora il nuovo ciclo di vita di un task per comprendere meglio il funzionamento di queste modifiche al sistema. Il task viene installato nel sistema e avviato (STARTED) tramite il Task Manager; il task rimane in questo stato finché il Broker non gli notifica un messaggio per un topic a cui è sottoscritto. Al ricevimento di un

messaggio, esso viene attivato (ENABLED) dal Task Manager e inserito nella coda dello Scheduler. Ad un certo punto, seguendo la politica EDF (Sezione 2.2.4), lo Scheduler lo selezionerà come il prossimo task che deve essere mandato in esecuzione. Questa operazione viene gestita dal Task Manager con la procedura `task_m_run_task` (vedi Listato 5.4). In essa viene chiamato il metodo `pick_input` introdotto nella Sezione 4.1.2, che ha il compito di decidere quale tra i messaggi in coda debba essere processato, e immediatamente dopo il `run`, che avvia l'esecuzione effettiva del task. Al termine, se ci sono altri messaggi in attesa, il task viene nuovamente messo nella coda dello Scheduler; diversamente, il task torna nello stato d'attesa (STARTED).

Listato 5.4: Frammento (semplificato) della procedura `task_m_run_task` del Task Manager

```
void task_m_run_task(task_struct *ptask) {
   // set the state state as RUNNING
   task_m_set_task_state(ptask, RUNNING);
   // select the message from the input queue(s) that should be processed
      next according to a policy (e.g FIFO, lottery)
   ptask->current_input = ptask->pick_input();
   // execute the task
   ptask->run(ptask);

   if (ptask->in_count <= 0) {
      // set the task state as STARTED if there are no more messages in
         the queue(s)
      task_m_set_task_state(ptask, STARTED);
   } else {
      // otherwise, set the task state as ENABLED
      task_m_set_task_state(ptask, ENABLED);
      // and enqueue it in the scheduler again
      sched_enqueue(ptask);
   }
}
```

5.4 Lottery

Per l'implementazione della Lottery è stato necessario creare due strutture dati: `lott_participant` e `lottery_struct`.
Il `lott_participant` rappresenta un partecipante alla lotteria ed è caratterizzato da:

- `task`: riferimento al task partecipante;
- `n_req_tickets`: la quantità di ticket richiesti dal partecipante;
- `n_tickets`: la quantità di ticket che son stati assegnati al partecipante;
- `first_ticket_num`: il numero del primo ticket che il partecipante possiede.

A sua volta, la `lottery_struct` è formata da

- `name`: nome della lotteria (tipicamente il nome della risorsa);
- `n_participants`: numero di partecipanti;
- `participants[]`: array di partecipanti;
- `n_tickets`: numero totale di ticket disponibili per la lotteria (assegnati e non);
- `n_assigned_tickets`: numero di ticket assegnati;

Oltre a queste strutture dati, la Lottery è costituita da alcuni metodi che implementano i meccanismi principali. Il `lott_prog_filling` si occupa di calcolare l'assegnamento dei ticket ai partecipanti utilizzando il Progressive Filling come visto nella Sezione 2.3.3 e nell'Algoritmo 1. Dopo che ogni partecipante ha allocato la propria quantità di ticket, `lott_adjust_ticket_numbers` aggiusta i valori dei `first_ticket_num` in modo tale che ognuno dei partecipanti abbia dei ticket con numerazione contigua (ad esempio, un partecipante con 4 ticket e primo ticket con numero 13, possiede i ticket dal 13 al 16).

Il cuore della lotteria è `lott_extract_ticket`, il quale utilizza un *pseudorandom number generator* (PRNG) per estrarre un numero casuale tra 1 e `n_assigned_tickets` che determinerà il ticket vincente. Data l'importanza di questo componente, nel prototipo son state incluse le implementazioni di due PRNG, il Lehmer (o Park-Miller) [70] e il PCG-basic [71], le cui prestazioni nell'estrazione dei ticket vengono esaminate assieme a quelle del generatore nativo `rand` nel Capitolo 6.

Capitolo 6

Evaluation

La valutazione di questa tesi si concentra nel verificare le capacità del sistema di arbitraggio realizzato nell'implementare l'allocazione ideale a scapito dell'overhead di sistema. Le prestazione dell'approccio Queuettery sono state confrontate, in setup determinati, con quella della politica *di default* di COIN nonché *common practice* su sistemi a risorse limitate [5, 72], ovvero FIFO.

Gli esperimenti condotti mostrano che Queuettery è in grado di ottenere un errore medio relativo all'allocazione ideale max-min inferiore al 7% in scenari tipici in cui lo stesso errore relativo registrato per FIFO supera mediamente il 76%, con punte del 95%. Inoltre, si constata che l'overhead computazionale e di memoria di Queuettery rispetto alla stessa FIFO risulta limitato per un numero di task tipico.

Setup Le misurazioni sono state condotte sul prototipo descritto nel Capitolo 5 utilizzando una evaluation board STM32F091RC (Figura 6.1), dotata di un microcontrollore Cortex-M0, 256KBytes di memoria flash e 32KB di memoria SRAM, rappresentativa dei dispositivi usati per la applicazioni di cui alla Sezione 2.1.
Per poterne controllare le caratteristiche, si è scelto di simulare una risorsa, rappresentativa di una radio, che fosse in grado di inviare in maniera costante 1 pacchetto ogni 100ms. Ogni pacchetto è rappresentato da un ticket e l'arco temporale di riferimento per la Lottery è pari 10 secondi (10000ms), cosicché sono disponibili 100 ticket.
Le valutazioni presentano diverse formazioni di task con specifiche configurazioni di domanda e frequenza di richiesta per tale risorsa. Tali task si occupano di fatto

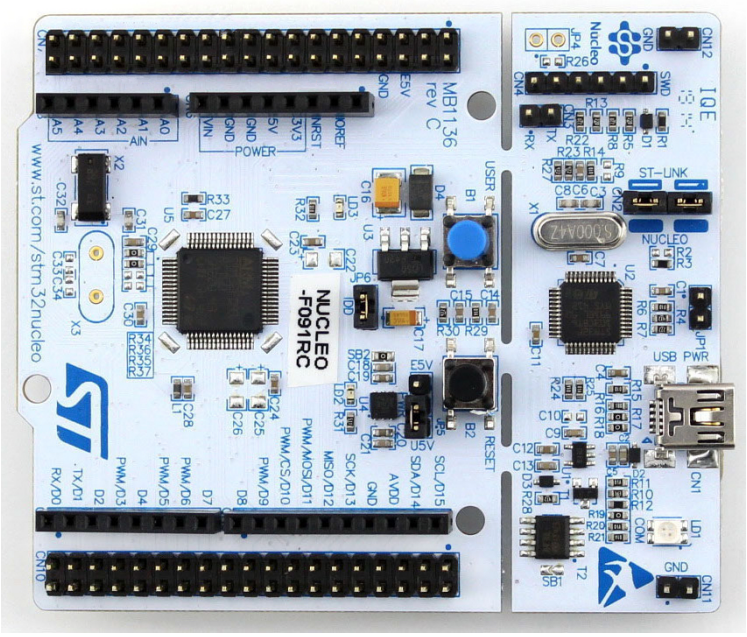


Figura 6.1: Evaluation board STM32F091RC

esclusivamente dell'invio di messaggi e hanno un tempo per la singola esecuzione minimo, mentre il tempo di esecuzione della risorsa viene adattato dinamicamente per mantenerne costante la capacità nel tempo. Questa costanza della risorsa è stata realizzata perché i risultati degli esperimenti non venissero influenzati e resi incomparabili da scostamenti temporali tra le richieste e il rate effettivo della risorsa. La lunghezza della coda l_{FIFO} per la politica FIFO è stata impostata a 20, mentre la lunghezza di ognuna delle code di Queuettery l_{qttry} utilizzata è 5.

Metriche Questa valutazione misura i benefici di Queuettery in termini di scostamento (errore) relativo dell'allocazione effettiva da quella ideale max-min, confrontati con l'overhead computazionale e di uso della memoria da questo introdotti. ll tempo del sistema nella board è misurato grazie ad un timer di sistema, il *SysTick*, che ha un periodo di circa 1ms. I microcontrollori di classe M0 non dispongono purtroppo del supporto pieno per l'unità *Data Watchpoint and Trace* (DWT). In particolare, non è supportato il registro DWT_CYCCNT che avrebbe permesso di ottenere una misurazione delle performance con precisione al livello di ciclo di clock. Le misurazioni temporali sono state eseguite mediando 1000 esecuzioni di operazioni con durata sufficiente da essere rilevante rispetto alla precisione del SysTick.

6.1 E1: Capacità di implementazione dell'allocazione e valutazione dei PRNG

L'obiettivo di questa valutazione è verificare l'allocazione delle richieste per task eterogenei in condizioni di saturazione della risorsa. Inoltre, si sono volute confrontare le prestazioni dei due PRNG, Lehmer e PCG-basic, assieme a quelle del generatore nativo `rand`.

A tal scopo sono stati eseguiti quattro esperimenti per misurare le prestazioni di sistemi basati su FIFO e su Queuettery con ciascuno dei tre PRNG. Gli esperimenti sono stati svolti per il tempo necessario alla risorsa per inviare un totale di 1000 pacchetti. Si è ipotizzato lo scenario in cui ad inviare i pacchetti siano 4 task eterogenei, i cui parametri son riportati nella Tabella 6.1: il $Task_A$ trasmette ad alta frequenza l'orientamento del dispositivo rilevato dal giroscopio; i $Task_B$ e $Task_C$ trasmettono la media della temperatura e dell'umidità dell'aria, rispettivamente; infine, il $Task_D$ tramette a bassa frequenza la localizzazione del dispositivo.

	$Task_A$	$Task_B$	$Task_C$	$Task_D$
Periodo(ms)	100	500	250	1000
Domanda (ticket)	100	20	40	10

Tabella 6.1: [E1] Configurazione dei task

Nella Figura 6.2 sono mostrati i risultati degli esperimenti in termini di distribuzione quantitativa delle richieste. Per ciascuno dei task sono visibili:

- *Domanda:* la quantità di richieste inviate.
- *Max-min:* l'allocazione ideale max-min fairness.
- *FIFO:* il numero di richieste soddisfatte utilizzando la politica di default.
- *rand*, *Lehmer* e *PCG-basic:* il numero di richieste soddisfatte utilizzando Queuettery e ciascuno dei tre PRNG.

La prima evidenza che emerge dalla Figura 6.2 è come la politica FIFO prediliga i task a più alta frequenza di invio. Specialmente $Task_A$, che ha un volume di richieste superiore a quello degli altri tre combinati, arriva quasi a monopolizzare la risorsa. Agli altri task rimane una quota minoritaria che si distribuisce in maniera

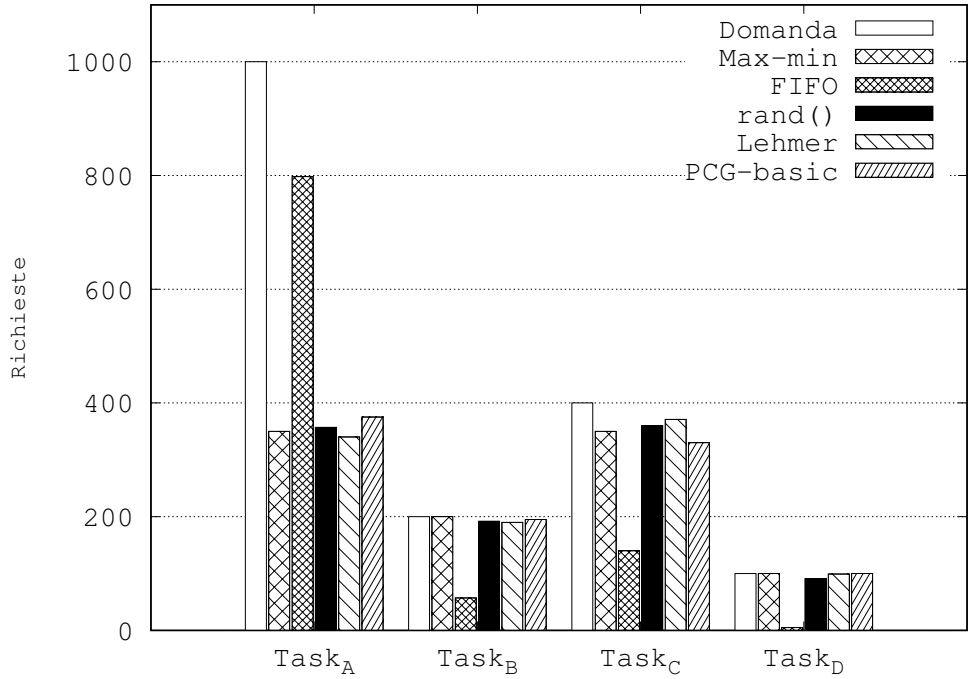
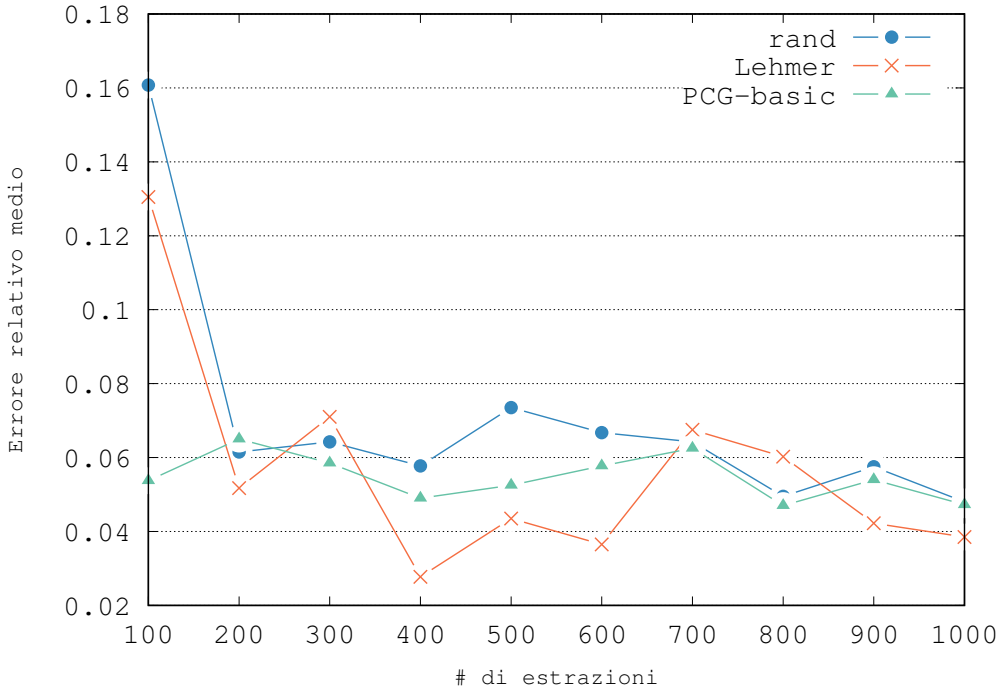


Figura 6.2: [E1] Distribuzione della risorsa tra i quattro task

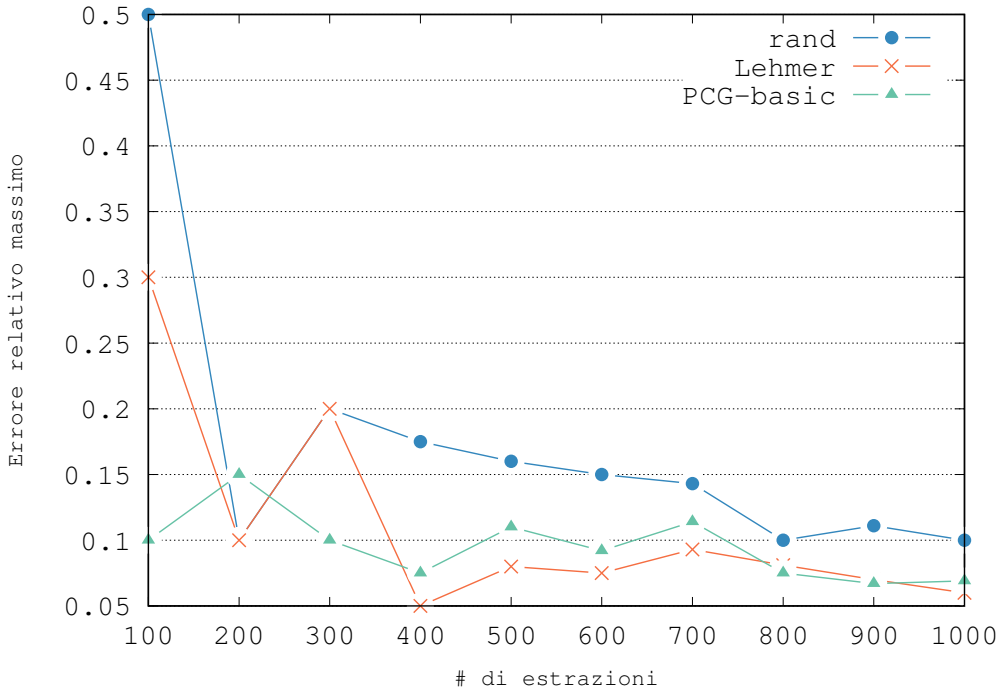
approssimativamente proporzionale alla loro domanda. Si noti come $Task_D$ venga quasi strozzato, ricevendo all'incirca il 5% della propria domanda. La sua presenza nella coda viene cancellata dall'alta frequenza di $Task_A$, il quale continuando ad aggiungere richieste in coda causa l'eliminazione delle poche richieste di $Task_D$ prima che queste possano essere soddisfatte.

I risultati degli altri tre esperimenti mostrano come, a prescindere dalle performance dei tre PRNG, l'approccio Queuettery sia in grado di avvicinarsi all'allocazione ideale indipendentemente dal flusso di richieste. Il discostamento relativo peggiore è compiuto dalla `rand` ai danni di $Task_D$, che riceve circa il 10% in meno della propria allocazione.

Queste informazioni non sono sufficienti per trarre considerazioni conclusive sui PRNG, le cui prestazioni appaiono relativamente simili. Per analizzare meglio il loro comportamento, nella Figura 6.3 sono riportati le evoluzioni di media e massimo dell'errore relativo rispetto all'allocazione max-min dei tre PRNG al crescere del numero di estrazioni.
Generalmente, come atteso, sia la media che il massimo tendono a diminuire con l'aumentare delle estrazioni, tuttavia i comportamenti appaiono significativamente



(a) Medio



(b) Massimo

Figura 6.3: [E1] Comparazione degli errori relativi dei tre PRNG rispetto all'allocazione ideale.

diversi con un numero basso di estrazioni. In particolare, Lehmer presenta una notevole varianza nelle prime 400 estrazioni, mentre la `rand` appare avere quasi sempre un comportamento peggiore rispetto alle altre due, in maniera speciale con un numero ridotto di estrazioni. PCG-basic, per quanto non risulti avere prestazioni nettamente migliori, conserva una dinamica più uniforme, significativamente con poche estrazioni.

Un buon comportamento fin dal principio è una caratteristica desiderabile per un PRNG, in quanto è sintomo di una buona uniformità nella distribuzione dei numeri. Dal momento che non son state fatte assunzioni sul numero di estrazioni a cui un task plausibilmente partecipa per una risorsa e che è preferibile per la valutazione che l'errore introdotto dal PRNG abbia un'entità stimabile, il PCG-basic è stato scelto come generatore per gli esperimenti seguenti.

6.2 E2: Comportamento dell'errore al crescere del numero di task

In questo esperimento si è voluto misurare l'andamento dell'errore con l'aumentare del numero dei task nel sistema. Sono stati utilizzati dei task uguali tra loro nelle caratteristiche (periodo 125ms e domanda 80 ticket) e sono state realizzate due esperimenti di 1000 richieste per ciascun incremento nel numero dei task: una utilizzando FIFO e una Lottery.

Le limitate capacità del microcontrollore e della SRAM hanno posto un limite al numero di task per cui questo test poteva essere effettuato. Superati i 5 task, sia per la FIFO che per Queuettery, la board non è più in grado di eseguire il prototipo senza incorrere in errori di memoria o sacrificando nello scheduling l'esecuzione di alcuni task. Non potendo disporre di maggiore risorse e non desiderando introdurre fenomeni d'errore esterni ai meccanismi analizzati e difficilmente prevedibili, si sono condotte gli esperimenti fino al raggiungimento di questo limite.

Nella Figura 6.4 viene riportato l'errore relativo rispetto all'allocazione ideale nel caso peggiore tra i task presenti. Si vede come entrambe le politiche partano da un errore nullo, in quanto la capacità della risorsa non è raggiunta da un singolo task. Una volta raggiunta la saturazione, FIFO incrementa il proprio errore in quanto non ha

alcun modo per equalizzare l'effetto sui task del turn-over della coda, la cui periodicità tende a penalizzare e/o premiare alcuni task piuttosto che altri. Di contro, invece, Queuettery, visto che i task hanno le stesse caratteristiche e dunque uguali allocazioni, si comporta in maniera simile ad una Fair Queue ed è in grado di mantenere l'errore basso e costante.

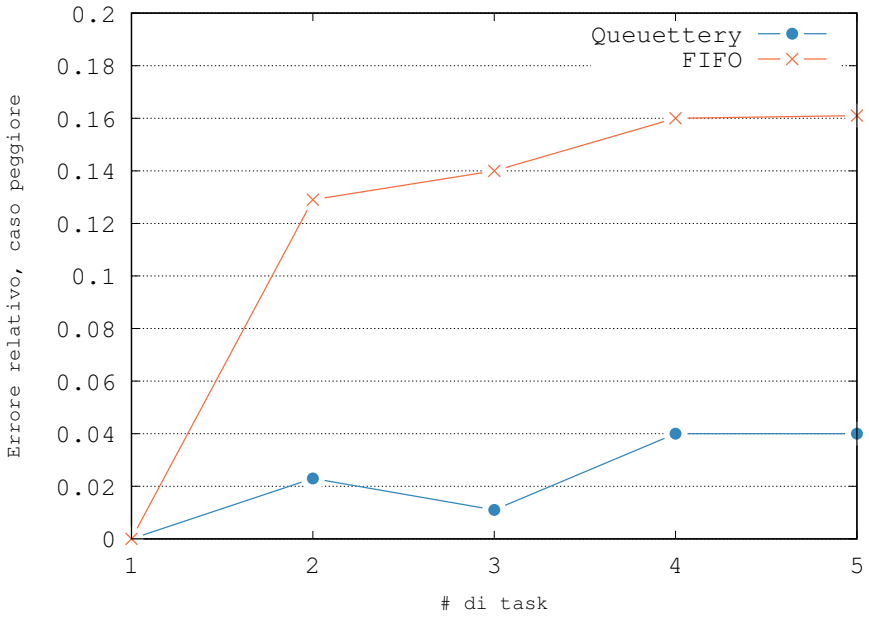


Figura 6.4: [E2] Comparazione dell'errore relativo nel caso peggiore tra Queuettery e FIFO

6.3 E3: Costo computazionale al crescere del numero dei task

Dopo aver valutato le capacità di Queuettery nel seguire l'allocazione ideale, questa valutazione mira ad analizzarne il costo computazionale al crescere del numero dei task. Analogamente agli esperimenti E2 (Sezione 6.2), i task hanno uguali caratteristiche e per ogni incremento nel numero dei task sono state fatte due esperimenti (uno per la FIFO, uno per Queuettery) fino al raggiungimento delle 1000 richieste esaudite. Anche in questo caso, il numero di task massimo negli esperimenti è stato pari a 5.

L'ideale per una valutazione puntuale dell'overhead di esecuzione sarebbe stato di misurare l'esecuzione per le singole operazioni in cui le due policy differiscono, ovvero

l'inserimento in coda e il prelievo dalla stessa. Tuttavia, singolarmente queste sono al di sotto di quanto può essere misurato con il SysTick e, come discusso nella premessa di questo capitolo, non è possibile utilizzare il numero di clock per una misura più precisa.

Per aggirare l'ostacolo e poter misurare l'overhead della gestione delle richieste, si è misurato il tempo tra una esecuzione della risorsa e la seguente. Affinché la differenza misurata fosse effettivamente quella tra le due policy, è stato necessario fissare le altre variabili:

- Il tempo di esecuzione della risorsa simulata, invece di adattarsi per garantire che le richieste soddisfatte rispettino il rate di 1 ogni 100ms, è stato congelato ad 80ms.
- Il prototipo, come discusso nella Sezione 5.2, utilizza un garbage collector per liberare la memoria dai messaggi eliminati. Per rendere deterministico ed uguale il suo contributo, le chiamate al garbage collector sono state effettuate ogni 50 esecuzioni della risorsa. Si noti che il ritmo a cui i messaggi sono prodotti dai task è uguale tra FIFO e Queuettery, per cui la quantità di memoria da processare è equivalente tra le due policy.

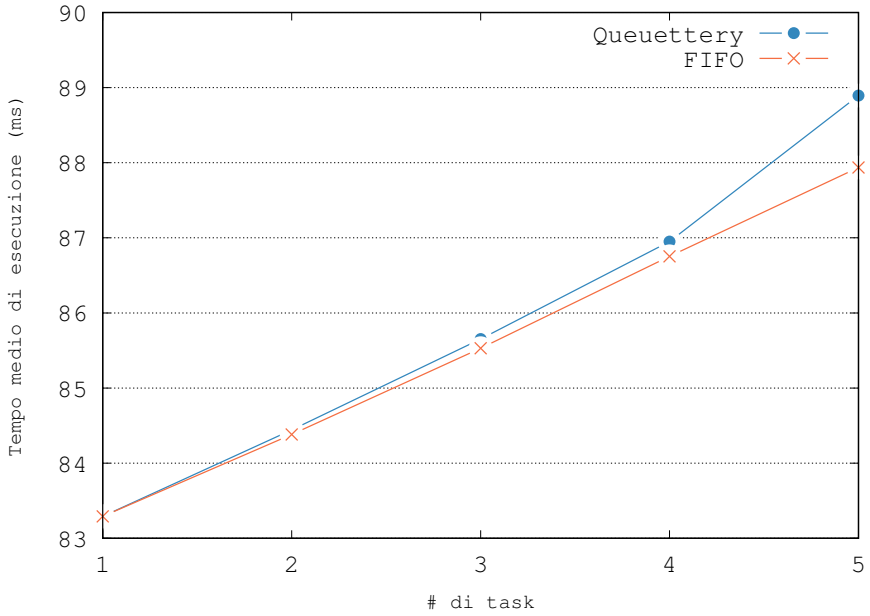


Figura 6.5: [E3] Tempo medio di esecuzione di Queuettery e FIFO al crescere del numero dei task

Nella Figura 6.5 sono riportate le medie delle 1000 misurazioni effettuate per ciascun esperimento. Entrambe le politiche pagano un overhead crescente generato dalla creazione, invio e rimozione di una quantità di messaggi che incrementa assieme al numero dei task. In questo, la politica FIFO mantiene un andamento lineare, mentre Queuettery mostra un andamento più che lineare, che è giustificato dal costo addizionale dell'estrazione e del lookup necessario a trovare la coda del vincitore, il quale aumenta con il numero dei task. Si nota però come l'andamento dell'overhead di Queuettery sia dominato dal costo base, comune alle due policy, la cui causa principale è identificabile nel carico crescente del garbage collector.

Immaginando di proiettare questi andamenti su un numero molto maggiore di task, Queuettery pagherebbe un overhead non indifferente, quasi esponenziale, rispetto alla FIFO. Va però considerato che l'intera architettura di COIN è realizzata sotto l'assunzione di un numero ridotto di task presenti nel sistema. In questo contesto, come si vede dal grafico e dalle valutazioni precedenti, Queuettery introduce un overhead ridotto a fronte di un'efficace controllo sulla distribuzione delle risorse.

6.4 E4: Overhead in memoria al crescere del numero dei task

L'overhead in memoria per task causato da Queuettery dipende principalmente dalle nuove strutture dati necessarie a rappresentare la lotteria e dalle code.

Struttura dati	Dimensione (Byte)
`task_struct`	128
`lottery_struct`	24
`lott_participant`	16
coda	$20 + 40 * \lceil \frac{length}{8} \rceil$
dizionario	$24 + 40 * \lceil \frac{length}{4} \rceil$

Tabella 6.2: Dimensione delle principali strutture dati del prototipo

Nella Tabella 6.2 sono riportate le dimensioni delle strutture dati principali. Utilizzando questi valori si ricavano le seguenti formule per il calcolo dell'overhead in memoria di una risorsa e di n task utilizzatori:

$$
\begin{aligned}
C_{FIFO} &= \underbrace{128 * (n+1)}_{\text{n task + 1 risorsa}} + \underbrace{20 + 40 * \left\lceil \frac{l_{FIFO}}{8} \right\rceil}_{\text{coda}} = \\
&= 268 + 128 * n \\
C_{Queuettery} &= \underbrace{128 * (n+1)}_{\text{n task + 1 risorsa}} + \underbrace{n * (20 + 40 * \left\lceil \frac{l_{qttry}}{8} \right\rceil)}_{\text{n code}} + \underbrace{24 + 40 * \left\lceil \frac{n}{4} \right\rceil}_{\substack{\text{dizionario di} \\ \text{n elementi}}} + \underbrace{24 + 16 * n}_{\substack{\text{lottery +} \\ \text{n partecipanti}}} = \\
&= 216 + 164 * n + 40 * \left\lceil \frac{n}{4} \right\rceil
\end{aligned}
$$

L'andamento di queste funzioni (Figura 6.6) riflette anche in questo caso il costo addizionale di Queuettery dovuto sia al dizionario, che consente di indicizzare le code, sia al fatto che ad ogni task utilizzatore venga assegnata una propria coda. La FIFO mantiene invece, fatto salvo il costo dei singoli task, un andamento indipendente dal numero di utilizzatori.

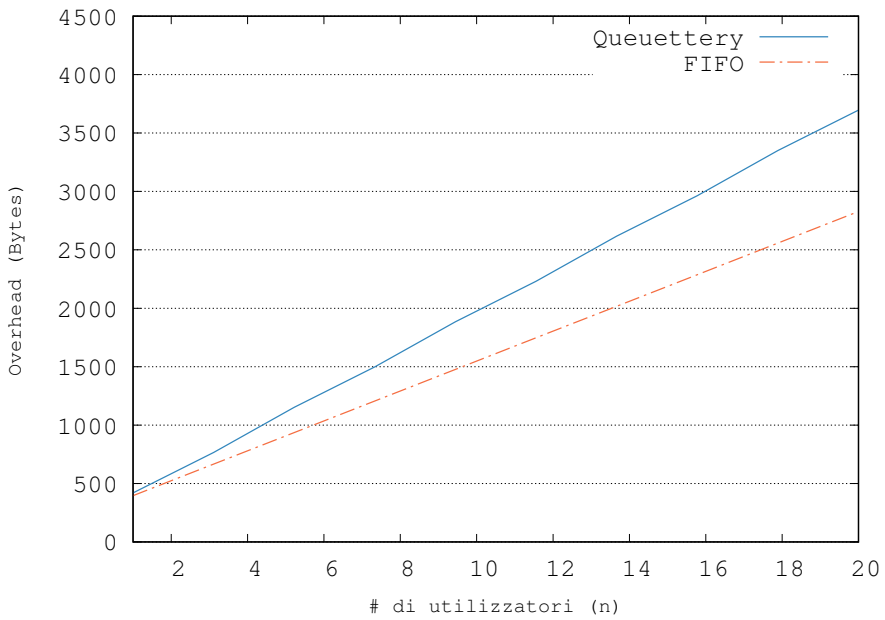


Figura 6.6: [E4] Andamento del costo in memoria in funzione del numero di utilizzatori di Queuettery e FIFO per una risorsa

Osserviamo che l'overhead in memoria di Queuettery risulta inferiore ai 500 Byte ogni 10 task. Questa entità va nuovamente messa in relazione col fatto che l'uso tipico

di COIN prevede un numero limitato di task presenti nel sistema contemporaneamente e che rappresenta un frazione minima della memoria disponibile.

Capitolo 7

Conclusioni

In questa tesi si è affrontato il problema dell'arbitraggio di risorse generiche nel contesto dei sistemi embedded e, nello specifico, del architettura COIN. Tale tematica è relativamente nuova in questi ambiti, in quanto tipicamente non mettono a disposizione le risorse del sistema nel modo in cui COIN lo permette. L'analisi è stata quindi estesa alle soluzioni esistenti per sistemi di diverso tipo e ai vari approcci alla condivisione.

In seguito, si è individuata un'architettura per il sistema di arbitraggio con un approccio modulare, che permette a COIN flessibilità e adattabilità a politiche di gestione diverse preservandone il modello concettuale. Infine, questo lavoro ha proposto l'idea di Queuettery, un meccanismo originale di gestione delle richieste che consente l'implementazione di allocazioni teoriche della risorsa con un approccio probabilistico.

Queuettery si è dimostrato, nella valutazione sperimentale, in grado di ottenere questo risultato con un discostamento relativo rispetto all'allocazione max-min mediamente inferiore al 7%. In particolare, Queuettery nel confronto con la politica FIFO, la più comune per questo genere di sistemi, introduce, per un numero di task tipico, overhead computazionali e di memoria esigui. A fronte di questo, il sistema acquisisce un controllo sulla distribuzione della risorsa che, in scenari tipici caratterizzati da volumi di richiesta non uniformi, permette un guadagno nella precisione dell'allocazione effettiva fino al 90% rispetto a FIFO, prevenendo la *starvation* degli utilizzatori.

In conclusione, Queuettery è un approccio alla condivisione delle risorse che può essere utilizzato laddove la necessità di una gestione dinamica e flessibile deve incontrare le capacità limitate del sistema. La sua applicazione in COIN è un passo importante verso l'utilizzo in contesti reali, dove il sistema possa essere esposto agli utenti e ad esigenze applicative diverse.

Queuettery è di per sé un sistema completo nel raggiungimento dei suoi obiettivi, per cui i suoi sviluppi prevedibili sono principalmente focalizzati nell'ottimizzazione delle sue strutture dati e conseguente ulteriore riduzione dell'overhead per estenderne l'applicabilità ad un numero maggiore di task. Ad esempio, si potrebbe valutare la realizzazione di strutture ad albero specializzate per la ricerca della coda dei task o anche per ottimizzare l'associazione tra ticket vincente e possessore.

Gli sviluppi più interessanti riguardano l'intero sistema COIN, la cui adozione verrebbe notevolmente facilitata dalla realizzazione di API e SDK per le principali piattaforme mobile. Attualmente la comunicazione tra il dispositivo e lo smartphone avviene attraverso tecnologia BLE utilizzando un protocollo primitivo che non facilita lo sviluppo di applicazioni compatibili con COIN e limita anche le possibilità di evoluzione delle sue funzionalità. Un altro aspetto da affrontare in un sistema che permette di eseguire codice arbitrario come questo è la sicurezza, ad esempio con tecniche di sandboxing dei task o della Virtual Machine.
O, ancora, può essere esplorata l'applicazione di COIN su sistemi embedded diversi da quelli attualmente in oggetto (e.g. smartwatch, droni, robot).

Bibliografia

[1] Maria Laura Stefanizzi, Luca Mottola, Luca Mainetti, and Luigi Patrono. Coin: Opening the internet of things to people's mobile devices. *IEEE Communications Magazine*, 55(2):20–26, February 2017.

[2] Carsten Magerkurth, Adrian David Cheok, Regan L. Mandryk, and Trond Nilsen. Pervasive games: Bringing computer entertainment back to the real world. *Comput. Entertain.*, 3(3):4–4, July 2005.

[3] Luca Mottola, Amy L. Murphy, and Gian Pietro Picco. Pervasive games in a mote-enabled virtual world using tuple space middleware. In *Proceedings of 5th ACM SIGCOMM Workshop on Network and System Support for Games*, NetGames '06, New York, NY, USA, 2006. ACM.

[4] Dimitri P Bertsekas, Robert G Gallager, and Pierre Humblet. *Data networks*, volume 2. Prentice-Hall International New Jersey, 1992.

[5] Luca Mottola, Animesh Pathak, Amol Bakshi, Viktor K Prasanna, and Gian Pietro Picco. Enabling scope-based interactions in sensor network macroprogramming. In *Mobile Adhoc and Sensor Systems, 2007. MASS 2007. IEEE International Conference on*, pages 1–9. IEEE, 2007.

[6] Smart Card Alliance Mobile and NFC Council. Bluetooth low energy (ble) 101: A technology primer with example use cases. Technical report, 2014.

[7] Peter Bauer, Mihail Sichitiu, Robert Istepanian, and Kamal Premaratne. The mobile patient: wireless distributed sensor networks for patient monitoring and care. In *Information Technology Applications in Biomedicine, 2000. Proceedings. 2000 IEEE EMBS International Conference on*, pages 17–21. IEEE, 2000.

[8] Rune Fensli, Einar Gunnarson, and Torstein Gundersen. A wearable ecg-recording system for continuous arrhythmia monitoring in a wireless tele-home-care situation. In *Computer-Based Medical Systems, 2005. Proceedings. 18th IEEE Symposium on*, pages 407–412. IEEE, 2005.

[9] M Masilela, EA Hughes, C Boanca, R Merrell, and A Rafiq. Vmote-ii, a biowearable health monitoring system. In *Information Technology Applications in Biomedicine, 2007. ITAB 2007. 6th International Special Topic Conference on*, pages 169–173. IEEE, 2007.

[10] Fazlay Rabbi, Taiwoo Park, Biyi Fang, Mi Zhang, Youngki Lee, and Rajiv Ranganathan. Demo: Towards immersive and interactive gym exercises. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services Companion*, MobiSys '16 Companion, pages 114–114, New York, NY, USA, 2016. ACM.

[11] Xia Kun, Xu Xinyue, and Wang Nan. Design of vehicle control system based on bluetooth low energy smartphone platform. In *Proc. Int. Conf. Electrical Machines and Systems (ICEMS)*, pages 1498–1501, October 2013.

[12] R. Frank, W. Bronzi, G. Castignani, and T. Engel. Bluetooth low energy: An alternative technology for vanet applications. In *Proc. 11th Annual Conf. Wireless On-demand Network Systems and Services (WONS)*, pages 104–107, April 2014.

[13] Samreen N Shaikh and SR Patil. Vehicle to vehicle communication system for smart cities.

[14] Alessio Agneessens, Francesco Buemi, Stefano Delucchi, Massimo Massa, Giovanni Agosta, Alessandro Barenghi, Carlo Brandolese, William Fornaciari, Gerardo Pelosi, Enrico Ferrari, et al. Safe cooperative cps: A v2i traffic management scenario in the safecop project. In *Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS), 2016 International Conference on*, pages 320–327. IEEE, 2016.

[15] Philips Hue. http://www.meethue.com/.

[16] Nest Thermostat. https://nest.com/thermostat/meet-nest-thermostat/.

[17] Augusta Smart Lock. http://august.com/products/august-smart-lock/.

[18] MA Prada-Delgado, A Vázquez-Reyes, and I Baturone. Physical unclonable keys for smart lock systems using bluetooth low energy. In *Industrial Electronics Society, IECON 2016-42nd Annual Conference of the IEEE*, pages 4808–4813. IEEE, 2016.

[19] Edyn. https://www.edyn.com/.

[20] GreenIQ. http://greeniq.co/.

[21] Nest Protect. https://nest.com/smoke-co-alarm/meet-nest-protect/.

[22] TrackR. https://www.thetrackr.com/.

[23] G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT AI-TR. MIT Press, 1986.

[24] Vijayshree Shinde and Seema C. Comparison of real time task scheduling algorithms. *International Journal of Computer Applications*, 158(6):37–41, jan 2017.

[25] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, January 1973.

[26] Mehdi Kargahi and Ali Movaghar. A method for performance analysis of earliest-deadline-first scheduling policy. *The Journal of Supercomputing*, 37(2):197–222, aug 2006.

[27] Anton-Hermann Chroust. Aristotle's conception of equity (epieikeia). *Notre Dame Law.*, 18:119, 1942.

[28] Menahem E Yaari and Maya Bar-Hillel. On dividing justly. *Social choice and welfare*, 1(1):1–24, 1984.

[29] H Peyton Young. *Equity: in theory and practice*. Princeton University Press, 1995.

[30] Guglielmo Lulli and Amedeo Odoni. The european air traffic flow management problem. *Transportation Science*, 41(4):431–443, 2007.

[31] Steven J Brams and Alan D Taylor. *Fair Division: From cake-cutting to dispute resolution*. Cambridge University Press, 1996.

[32] Steven J Brams and Alan D Taylor. *The win-win solution: guaranteeing fair shares to everybody.* WW Norton & Company, 2000.

[33] Jeonghoon Mo and Jean Walrand. Fair end-to-end window-based congestion control. *IEEE/ACM Trans. Netw.*, 8(5):556–567, October 2000.

[34] Laurent Massoulié and James Roberts. Bandwidth sharing: Objectives and algorithms. *IEEE/ACM Trans. Netw.*, 10(3):320–328, June 2002.

[35] D. Julian, Mung Chiang, D. O'Neill, and S. Boyd. QoS and fairness constrained convex optimization of resource allocation for wireless cellular and ad hoc networks. In *Proc. Twenty-First Annual Joint Conf. of the IEEE Computer and Communications Societies*, volume 2, pages 477–486, June 2002.

[36] H. SHI, R. V. Prasad, E. Onur, and I. G. M. M. Niemegeers. Fairness in wireless networks:issues, measures and challenges. *IEEE Communications Surveys Tutorials*, 16(1):5–24, 2014.

[37] J. Kay and P. Lauder. A fair share scheduler. *Commun. ACM*, 31(1):44–55, January 1988.

[38] J. L. Hellerstein. Achieving service rate objectives with decay usage scheduling. *IEEE Transactions on Software Engineering*, 19(8):813–825, August 1993.

[39] Chee Siang Wong, Ian Tan, Rosalind Deena Kumari, and Fun Wey. Towards achieving fairness in the linux scheduler. *SIGOPS Oper. Syst. Rev.*, 42(5):34–43, July 2008.

[40] Jean-Yves Le Boudec. Rate adaptation, congestion control and fairness: A tutorial. *Web page, November*, 2005.

[41] A. Malla, M. El-Kadi, S. Olariu, and P. Todorova. A fair resource allocation protocol for multimedia wireless networks. *IEEE Transactions on Parallel and Distributed Systems*, 14(1):63–71, January 2003.

[42] Angelo Coluccia, Alessandro D'Alconzo, and Fabio Ricciato. On the optimality of max–min fairness in resource allocation. *annals of telecommunications-annales des télécommunications*, 67(1-2):15–26, 2012.

[43] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Ecosystem: Managing energy as a first class operating system resource. *SIGPLAN Not.*, 37(10):123–132, October 2002.

[44] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association.

[45] Konrad Lorincz, Bor-rong Chen, Jason Waterman, Geoff Werner-Allen, and Matt Welsh. Resource aware programming in the pixie os. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 211–224, New York, NY, USA, 2008. ACM.

[46] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, March 2004.

[47] Arjun Roy, Stephen M. Rumble, Ryan Stutsman, Philip Levis, David Mazières, and Nickolai Zeldovich. Energy management in mobile devices with the cinder operating system. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 139–152, New York, NY, USA, 2011. ACM.

[48] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. *Commun. ACM*, 54(11):93–101, November 2011.

[49] M. B. Jones, P. J. Leach, R. P. Draves, and J. S. Barrera. Modular real-time resource management in the rialto operating system. In *Proc. 5th Workshop Hot Topics in Operating Systems (HotOS-V)*, pages 12–17, May 1995.

[50] Leonard Kleinrock and Richard R Muntz. Processor sharing queueing models of mixed scheduling disciplines for time shared system. *Journal of the ACM (JACM)*, 19(3):464–482, 1972.

[51] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 1. ACM, 2016.

[52] J. Nagle. On packet switches with infinite storage. *IEEE Transactions on Communications*, 35(4):435–438, April 1987.

[53] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Symposium Proceedings on Communications Architectures &Amp; Protocols*, SIGCOMM '89, pages 1–12, New York, NY, USA, 1989. ACM.

[54] Carl A. Waldspurger and William E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association.

[55] Carl A Waldspurger. *Lottery and Stride Scheduling: Proportional Share Resource Management*. PhD thesis, PhD thesis, MIT, 1995.

[56] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet. *SIGARCH Comput. Archit. News*, 30(5):96–107, October 2002.

[57] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, WSNA '02, pages 88–97, New York, NY, USA, 2002. ACM.

[58] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 381–396, Berkeley, CA, USA, 2006. USENIX Association.

[59] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steven Glaser, and Martin Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, IPSN '07, pages 254–263, New York, NY, USA, 2007. ACM.

[60] Shashank Priya, Chih-Ta Chen, Darren Fye, and Jeff Zahnd. Piezoelectric windmill: a novel solution to remote sensing. *Japanese journal of applied physics*, 44(1L):L104, 2004.

[61] Kris Lin, Jennifer Yu, Jason Hsu, Sadaf Zahedi, David Lee, Jonathan Friedman, Aman Kansal, Vijay Raghunathan, and Mani Srivastava. Heliomote: Enabling long-lived sensor networks through solar energy harvesting. In *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, SenSys '05, pages 309–309, New York, NY, USA, 2005. ACM.

[62] Naveed Anwar Bhatti, Muhammad Hamad Alizai, Affan A. Syed, and Luca Mottola. Energy harvesting and wireless transfer in sensor network applications: Concepts and experiences. *ACM Trans. Sen. Netw.*, 12(3):24:1–24:40, August 2016.

[63] Nicolas Tsiftes and Thiemo Voigt. Velox: A virtual machine for iot software security and resource protection. 2015.

[64] Q. Cao, D. Fesehaye, N. Pham, Y. Sarwar, and T. Abdelzaher. Virtual battery: An energy reserve abstraction for embedded sensor networks. In *Proc. Real-Time Systems Symp*, pages 123–133, November 2008.

[65] A. Eswaran, A. Rowe, and R. Rajkumar. Nano-rk: an energy-aware resource-centric RTOS for sensor networks. In *Proc. 26th IEEE Int. Real-Time Systems Symp. (RTSS'05)*, pages 10 pp.–265, December 2005.

[66] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: A language and runtime system for perpetual systems. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 161–174, New York, NY, USA, 2007. ACM.

[67] Andreas Lachenmann, Pedro José Marrón, Daniel Minder, and Kurt Rothermel. Meeting lifetime goals with energy levels. In *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, SenSys '07, pages 131–144, New York, NY, USA, 2007. ACM.

[68] PyMite. https://code.google.com/archive/p/python-on-a-chip/.

[69] Paul R Wilson. Uniprocessor garbage collection techniques. In *Memory Management*, pages 1–42. Springer, 1992.

[70] Stephen K. Park and Keith W. Miller. Random number generators: good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.

[71] MELISSA E O'Neill. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. *ACM Trans. Math. Softw.(submitted)*, pages 1–46, 1988.

[72] Marco Zimmerling, Luca Mottola, Pratyush Kumar, Federico Ferrari, and Lothar Thiele. Adaptive real-time communication for wireless cyber-physical systems. *ACM Transactions on Cyber-Physical Systems*, 1(2):8, 2017.