Università degli Studi di Padova

Facoltà di Ingegneria

Corso di Laurea Specialistica in Ingegneria Informatica

Tesi di Laurea

Implementazione di sincronizzazione temporale distribuita in reti di sensori wireless

Relatore: Ch.mo Prof. Luca Schenato

Laureando: Federico Fiorentin

Anno Accademico 2007-2008

Ai miei nonni
Alla mia famiglia
A Costanza

Indice

Sommario

La sincronizzazione temporale è un elemento critico ed importante per ogni sistema distribuito. Il concetto di tempo deriva da quello più generale che riguarda l'ordine in cui avvengono gli eventi. Per questo, gran parte dei dati raccolti da una rete di sensori non porta alcuna informazione se non accompagnata da un riferimento temporale omogeneo. Molte applicazioni per reti di sensori richiedono che l'orologio locale di ciascun nodo sensore sia sincronizzato secondo vari gradi di precisione. Le proprietà intrinseche di una rete di sensori come risorse di energia, calcolo, memoria, banda limitata combinate con le alte densità dei nodi della rete rendono i metodi tradizionali di sincronizzazione inadeguati. Gli esistenti metodi di sincronizzazioni necessitano di essere estesi per soddisfare le nuove richieste. Tutto ciò porta ad una continua ricerca di algoritmi specifici di sincronizzazione per reti di sensori.

Inizialmente vengono affrontati lo studio, l'analisi e la descrizione delle soluzioni già presenti in letteratura nel campo della sincronizzazione temporale. Vengono introdotti alcuni parametri per confrontare i vari algoritmi proposti in modo indipendente dalle implementazioni e dal tipo di hardware utilizzato. Il contributo principale di questo lavoro è la presentazione di un nuovo algoritmo di sincronizzazione completamente distribuito, chiamato *Average TimeSync*, in cui tutti i nodi operano alla stessa maniera senza distinzione di compiti. La soluzione proposta raggiunge la sincronizzazione globale della rete attraverso lo scambio locale di informazioni tra nodi vicini. L'algoritmo è caratterizzato da un'elevata robustezza ai guasti in quanto l'informazione per la sincronizzazione è distribuita su tutta la rete. Perciò, risulta indipendente dalla topologia della rete ed è particolarmente indicato per reti che possono variare nel tempo. Ogni nodo della rete si accorda con gli altri su un tempo globale di riferimento sulla

base delle proprie informazioni e di quelle ricevute dai nodi vicini. Questo tipo di tecnica fa parte dei *problemi di consenso*. La nuova soluzione è implementata su sensori wireless (architettura Tmote/TinyOS-2.X) per verificarne il funzionamento e analizzarne le prestazioni. Il software prodotto e gli ottimi risultati ottenuti costituiscono il punto di partenza per lo sviluppo di applicazione che richiedono una rete di nodi sincronizzata come ad esempio l'esecuzione di azioni di gruppo o la multiplazione temporale del canale di trasmissione.

Capitolo 1

Introduzione

I recenti vantaggi portati nei campi della miniaturizzazione dal basso costo e basso consumo dei dispositivi elettronici hanno dato vita a sistemi di sensori e attuatori wireless. La prospettiva è quella di creare un "ambiente intelligente" in grado di percepire le condizioni ambientali come temperatura, movimento, suoni e luminosità attraverso il dispiegamento di migliaia di sensori, ciascuno dei quali ha la possibilità di elaborare e trasmettere dati via wireless con distanza massima limitata.

La sincronizzazione temporale costituisce un blocco fondamentale per l'infrastruttura di ogni sistema distribuito. Le reti di sensori wireless (WSN) fanno un uso particolarmente estensivo della sincronizzazione: per esempio integrare una serie di rilevazioni di prossimità per stimare il movimento di un oggetto o di un veicolo; per misurare il tempo di propagazione del suono per localizzarne la sorgente; per sopprimere messaggi ridondanti riconoscendo che questi messaggi descrivono duplicazioni dello stesso evento registrato da diversi sensori. Le reti di sensori hanno inoltre le stesse esigenze dei tradizionali sistemi distribuiti: accurati timestamps[1] sono infatti richiesti nella crittografia e per la coordinazione di eventi programmati secondo un certo ordine (scheduling).

La larga natura delle applicazioni riguardanti le reti di sensori porta ad esigenze di temporalizzazione che per portata, durata e precisione differiscono dai sistemi tradizionali. Inoltre la maggior parte dei nodi ha a disposizione una

[1] Con il termine timestamp ci si riferisce alla misurazione del tempo da parte di un dispositivo in un determinato istante in accordo con il riferimento di tempo locale.

riserva di energia limitata. Tutte le comunicazioni, pure l'ascolto passivo, ha significanti effetti su questa riserva. I metodi di sincronizzazione temporale per WSN in particolare devono tener conto di tempi ed energia che essi consumano.

1.1 Reti di Sensori Wireless

Una rete di sensori wireless (WSN è l'acronimo inglese per Wireless Sensor Network) consiste in una serie di dispositivi distribuiti nello spazio e che utilizzano i loro sensori per cooperare al monitoraggio di eventi o situazioni ambientali. Il termine *sensore* sta ad indicare un dispositivo che è in grado di tradurre una grandezza fisica in un segnale di natura diversa (tipicamente elettrico) più facilmente misurabile o memorizzabile. Solitamente il segnale originale viene tradotto in una grandezza di tensione o corrente che ne permette l'acquisizione e l'eventuale elaborazione in formato digitale.
Le reti di sensori sono costituite da un insieme di nodi. Ciascun nodo della rete è dotato di uno o più sensori, di un trasmettitore radio (o altro tipo di dispositivo di trasmissione wireless), di un piccolo microcontrollore (che permette l'elaborazione delle informazioni) e infine di una sorgente di energia (tipicamente una batteria). I nodi di una WSN sono spesso impropriamente chiamati sensori, ciò ne evidenzia il fine principale: la rilevazione delle condizioni ambientali dello spazio in cui operano.

Lo sviluppo delle WSN fu originariamente motivato dalle possibili applicazioni in campo militare, come ad esempio la sorveglianza di un campo di battaglia. Tuttavia le reti di sensori hanno trovato ampio spazio in molte applicazioni civili, come monitoraggio di ambienti e abitazioni, applicazioni healthcare, automazione e controllo del traffico. La vasta varietà di sistemi e applicazioni in cui trovano impiego le WSN rende difficoltoso discutere su applicazioni specifiche e direzione della ricerca in questo ambito[1].

Il costo di una rete di sensori è variabile. Dipende dal numero di nodi, dalle dimensioni di questi (generalmente più piccoli sono più costano) e dal tipo di sensori montati su ciascun nodo. Altri vincoli legati alla dimensione e al costo del

nodo corrispondono alle quantità di risorse che il nodo dispone, come ad esempio l'energia, la memoria, capacità computazionale e la banda di trasmissione.

1.2 Applicazioni

Le applicazioni per WSN sono molte e di varia tipologia. Essenzialmente vengono usate in ambito industriale e commerciale per operazioni di monitoraggio che risulterebbero impossibili o troppo costose utilizzando sensori cablati (con cavo). I nodi della rete possono essere disposti in qualsiasi tipo di area dove possono rimanere per alcuni anni (monitorando alcune variabili ambientali ad esempio) senza la necessità di ricaricare o rimpiazzare la loro fonte di energia (es. batteria). Tipicamente le WSN sono state utilizzate per il monitoraggio, tracking di oggetti in movimento, tuttavia con la recente riduzione dei costi dei nodi le WSN trovano applicazione in vari settori, come ad esempio:

Agroalimentare Monitoraggio del terreno, rilevazione muffe, batteri, deterioramenti...

Salute Parametri vitali in presenza di patologie o sforzo fisico (sport)...

Automazione Civile Ospedali, aereoporti... (qualità della rete elettrica, climatizzazione, presenze...).

Militare Esplosivi, vibrazioni in aria,terra/acqua, comunicazioni...

Casa Climatizzazione, illuminazione, monitoraggio, allarmi, automazione di porte, finestre, giardini,...

Automotive Interno autovettura (qualità dell'aria, clima, regolazioni), infrastrutture (traffico, nebbia, controllo velocità).

Impianti Industriali Regolazione, controllo di processo, database, interfaccia operatore, analisi statistica...

Ambiente Clima, livello acque, agenti inquinanti, incendi, sismi, smottamenti...

La necessità di “fondere” queste applicazioni nell’ambiente in considerazione ed il grande numero di nodi impiegato rendono impensabile effettuare la sostituzione delle batterie di alimentazione quando si sono esaurite. A seguito di un dispiegamento di massa di nodi si può agire solamente sulle applicazioni attuando una politica di riduzione del consumo di energia in modo da allungare la durata delle batterie e limitare il cambio delle batterie con scadenza almeno annuale.

Capitolo 2

La Sincronizzazione Temporale

Il concetto di "tempo" è fondamentale per il nostro modo di pensare. Esso è derivato dal concetto più generale riguardante l'ordine in cui avvengono gli eventi. In un sistema distribuito, come quello di una WSN, tale concetto acquista ancora maggior importanza. La raccolta di dati di misurazione effettuata da una rete di sensori non porta alcuna informazione se non accompagnata da un riferimento temporale omogeneo. Vediamo di seguito come caratterizzare la sincronizzazione temporale e di stabilire alcune metriche per valutare i diversi algoritmi di sincronizzazione. Tali metriche sono state proposte con lo scopo di poter valutare l'algoritmo di sincronizzazione in modo indipendente dalla implementazione e dalle caratteristiche del dispositivo in cui verrà installato.

2.1 Caratterizzare la Sincronizzazione Temporale

Vi sono molti metodi di sincronizzazione disponibili al giorno d'oggi. Basta pensare ai sistemi Global Positioning System (GPS); oppure all'algoritmo Network Time Protocol di Mills [2] che si occupa della sincronizzazione di una una rete di computer, distribuendo su di essa il tempo fornito da un insieme di sorgenti, o una variante, il Precision Time Protocol (PTP definito nello standard IEEE 1588) [5].

Nello studiare la sincronizzazione temporale per WSN è utile classificare i diversi aspetti di un metodo sotto diversi profili.
Ci sono alcune metriche da tener in considerazione, che sono:

Precisione calcolata come dispersione temporale tra un insieme di nodi.

Durata che può variare da una sincronizzazione persistente ad una sincronizzazione istantanea.

Portata la zona geografica a cui si riferisce la sincronizzazione.

Disponibilità la possibilità di un nodo di accedere ad una misura di tempo globale.

Efficienza il tempo e l'energia necessaria per raggiungere tale sincronizzazzione.

Costo che assume importanza quando la sincronizzazione riguarda migliaia di nodi wireless.

I classici metodi di sincronizzazione falliscono in alcuni dei precedenti punti; nessun metodo è l'ottimale lungo tutti i precedenti assi. Ad esempio la sincronizzazione attraverso GPS può sincronizzare un nodo della rete in modo persistente e con una precisione superiore ai 200ns. Tuttavia i sistemi GPS spesso non possono essere usati (es. all'interno di edifici, in particolari condizioni ambientali, sott'acqua), ed inoltre richiedono diversi minuti di inizializzazione per esser operativi. In alcuni casi le unità GPS possono essere di grandi dimensioni, ad alto consumo energetico e costosi se paragonati ai nodi di una WSN.

Scalando i tradizionali metodi di sincronizzazione come NTP e PTP alle WSN, la frequenza di inoltro dei dati necessari per raggiungere tale sincronizzazione porta a consumare più energia di quella a disposizione del singolo nodo.

2.2 Metodi di valutazione

Valutare le proprietà di un algoritmo di sincronizzazione per WSN è una cosa piuttosto complicata. Un esempio è rappresentato dai dati presentati dagli autori

di RBS [9] che riportano una precisione di 11 μs dell'algoritmo su piattaforma Mica. Tuttavia, lo stesso algoritmo implementato sulla stessa piattaforma dagli autori di TPSN [11] ha raggiunto un errore medio nel caso di single-hop di 29,1 μs. L'errore riportato per TPSN è di 16,9 μs vincendo il confronto con RBS. Avendo due valori di precisione di RBS a quale dobbiamo fare affidamento? Probabilmente entrambi i risultati di RBS sono corretti.

Il problema è stabilire una serie di metriche standard da utilizzare per confrontare gli algoritmi in modo da valutarli equamente. È inoltre da sottolineare come sia piattaforma che implementazione dell'algoritmo possono influire nella prestazione finale dello stesso.

Vediamo di seguito di illustrare le principali metriche che sono state utilizzate per confrontare le prestazioni degli algoritmi di sincronizzazione presi in considerazine:

- la precisione.
- la complessità.
- il numero di messaggi necessari.
- la quantità di memoria necessaria per mantenere il servizio di sincronizzazione.
- la scalabilità.
- la dipendenza dalla topologia della rete e la robustezza ai guasti.

Precisione Nell'ambito delle WSN la *precisione* di un algoritmo di sincronizzazione è un valore strettamente legato all'errore di sincronizzazione. Mentre l'errore di sincronizzazione è una funzione del tempo t e si riferisce a ciascun nodo, la *precisione* della sincronizzazione è un parametro globale riferito a tutta la rete di nodi definito come valore massimo oppure valore medio della precisione istantanea in un intervallo di tempo.

Possiamo definire la *precisione instantanea* $p(t)$, altrimenti nominata come *dispersione di gruppo* (o group dispersion) in [9], come l'errore massimo di fase tra una qualsiasi coppia di nodi.

Definendo $c^x(h^x(t))$ la costruzione del clock software con la funzione $c^x(.)$ a partire dal clock locale hardware $h^x(t)$ all'istante t per ogni nodo x, la *precisione instantanea* risulta:

$$p(t) = \max_{i,j} \left\{ c^i(h^i(t)) - c^j(h^j(t)) \right\} \tag{2.1}$$

per ciascun nodo N_i e N_j da considerare.

Se per la sincronizzazione ci si riferisce ad una sorgente esterna di tempo allora definiremo

$$p(t) = \max_{i} \left\{ c^i(h^i(t)) - t \right\} \tag{2.2}$$

per ciascun nodo N_i da considerare.

Complessità È una grandezza direttamente proporzionale al numero di messaggi trasmessi e ai messaggi elaborati dai nodi per mantenere il servizio di sincronizzazione. È un valore indicativo del lavoro che la rete di nodi deve svolgere per mantenere il servizio di sincronizzazione.

Occupazione del Canale Questa parametro indica il grado di utilizzo del canale di trasmissione per mantenere la sincronizzazione. Dà una stima della congestione del mezzo di comunicazione.

Memoria Occupata Ogni algoritmo implementato su un elaboratore utilizza una porzione di memoria dello stesso per essere eseguito e per memorizzare variabili, stati del sistema. Lo stesso vale per ogni algoritmo di sincronizzazione, sarà quindi necessario memorizzare nel singolo nodo alcune

informazioni per raggiungere tale sincronizzazione. Ovviamente la parte di memoria necessaria per la sincronizzazione dovrà essere limitata in modo da permettere al nodo l'esecuzione delle operazioni di monitoring e rilevazione di dati ambientali per il quale è stato impiegato.

Scalabilità È intesa come capacità dell'algoritmo di sopportare le variazioni del numero di nodi che richiedono il servizio di sincronizzazione.

Dipendenza da una topologia di rete e robustezza ai guasti È un valore che mette in evidenza la capacità dell'algoritmo di essere indipendente dalla disposizione che avranno i nodi nello spazio e da un eventuale guasto che i nodi potranno conseguire.

2.3 Computer Clock e Sincronizzazione

La maggior parte dei computer e dei dispositivi in grado di eseguire operazione di calcolo hanno a disposizione un proprio clock [1] basato su un oscillatore hardware. Per oscillatore si intende un dispositivo in grado di generare una forma d'onda periodica qualsiasi, senza alcun segnale applicato all'ingresso. Solitamente il tipo di oscillatore usato è di quelli al quarzo per l'alta stabilità e il costo modesto. Specifichiamo di seguito il modello del sistema seguito per la sincronizzazione temporale che abbiamo usato come punto di partenza della nostra analisi. Per prima cosa definiamo una notazione standard che verrà mantenuta nel seguito del documento. Ad esempio definiamo come τ_a l'istante di tempo in cui si verifica l'evento generico a. Ed il tempo locale del nodo N_i in quell'istante come τ_a^i.

2.4 Modello del Clock

Un clock digitale misura essenzialmente intervalli di tempo. Esso consiste in un contatore τ che si incrementa idealmente ogni fissato periodo di tempo. Spesso

[1] Con il termine clock si indica un segnale periodico utilizzato per sincronizzare il funzionamento di dispositivi elettronici digitali.

ci si riferisce a questo contatore con il termine di "clock locale" e ne definiamo la lettura al tempo reale t come $\tau(t)$. Il contatore è incrementato da un oscillatore di frequenza f. La frequenza f al tempo reale t è data dalla derivata prima di $\tau(t)$, cioè $\delta\tau(t)/\delta t$.

Avendo a disposizione un oscillatore ideale la lunghezza degli intervalli temporali scanditi dal clock sarebbero tutti uguali e risulterebbe:

$$\tau(t) - \tau(t-1) = costante \qquad \forall t \tag{2.3}$$

$$f = \frac{\delta\tau(t)}{\delta t} = 1 \tag{2.4}$$

Tuttavia tutti i clock reali sono soggetti a fluttuazioni della frequenza f di funzionamento. Queste variazioni della frequenza di oscillazione che non sono prevedibili a priori possono dipendere da molteplici fattori: temperatura, sbalzi di tensione, campi magnetici, voltaggio applicato, invecchiamento, usura.

Sebbene la frequenza del clock possa cambiare nel tempo essa rimane sempre compresa tra certi valori limite, altrimenti la lettura del clock non porterebbe nessuna informazione. Infatti, può essere approssimata con buona accuratezza da un oscillatore con frequenza fissa. Quindi per un nodo i nella rete possiamo approssimare il suo clock locale come:

$$\tau_i(t) = \alpha_i t + \beta_i \tag{2.5}$$

dove α_i è nominato *skew* o deriva, mentre β_i è l'*offset* del clock del nodo i. Skew denota la frequenza del clock, offset è la distanza temporale dal tempo reale t. Visto che il nodo non può accedere al tempo di riferimento t non è possibile trovare i valori di α_i e β_i. In realtà i due parametri α_i e β_i sono dipendenti dal tempo t, tuttavia $\alpha_i(t)$ e $\beta_i(t)$ variano lentamente nel breve periodo e perciò possono essere considerati costanti. Quindi $\alpha_i(t) \cong \alpha_i$ e $\beta_i(t) \cong \beta_i$.

Tuttavia usando l'equazione 2.5 possiamo confrontare il tempo locale di due nodi qualsiasi della rete (diciamo nodo i e nodo j). Infatti risolvendo 2.5 rispetto a t otteniamo $t = \frac{\tau_i - \beta_i}{\alpha_i}$ e sostituendo nella stessa equazione relativa al nodo j

risulta:

$$\begin{aligned} \tau_j &= \frac{\alpha_j}{\alpha_i}\tau_i + (\beta_j - \frac{\alpha_j}{\alpha_i}\beta_j) \\ &= \alpha_{ij}\tau_i + \beta_{ij} \end{aligned} \tag{2.6}$$

Dove α_{ij} è lo *skew* relativo, mentre β_{ij} l'*offset* relativo del clock locale del nodo j rispetto al nodo i. Se i clock dei due nodi sono perfettamente sincronizzati allora il loro skew relativo è 1, il ché significa che i due clock hanno la stessa frequenza di aggiornamento e il loro offset relativo è 0. Alcuni scritti in letteratura fanno uso del parametro *phase offset* anziché *offset* e lo *skew* viene anche chiamato *drift*.

I clock dei nodi di una rete di sensori possono essere inconsistenti per diverse ragioni. Il clock può andare alla deriva a seguito di cambiamenti ambientali, come temperatura, pressione, voltaggio della batteria. I nodi della rete possono essere stati inizializzati in istanti differenti, e i loro clock continueranno a differire di tale quantità durante il loro funzionamento.
Il clock di un nodo può essere influenzato da altri componenti interni al sistema. Ad esempio i comuni nodi possono mancare l'evento che implica l'incremento del clock quando sono occupati a gestire trasmissioni di messaggi.
Il problema della sincronizzazione in una rete di n nodi corrisponde al problema di uguagliare il clock dei diversi nodi. La sincronizzazione può essere *globale*, cioè tende a uguagliare $\tau_i(t)$ per $i = 1...n$; ma può essere anche *locale* tentando di uguagliare $\tau_i(t)$ per un sottoinsieme di nodi della rete, solitamente tra nodi vicini nello spazio. Equalizzare istantaneamente i clock correggendo i loro offset non è sufficiente visto che il clock andrà alla deriva quasi immediatamente, per cui è necessario provvedere alla compensazione della frequenza di aggiornamento del clock oppure si dovrebbe provvedere ad un continuo periodico riaggiustamento degli offset per tenere i clock sincronizzati entro un certo errore. La definizione di sincronizzazione è spesso rilassata a diversi gradi di precisione in base alle necessità delle applicazioni che la utilizzano e in particolare in base ai vincoli fisici e tecnici dei dispositivi da sincronizzare.

Il problema della sincronizzazione può essere classificato secondo tre modelli: la prima forma è quella più semplice che ha lo scopo di poter stabilire l'ordine con cui si sono manifestati una serie di eventi. Lo scopo è quello di poter stabilire se un evento a è accaduto prima o dopo un evento b.
La seconda forma permette ad ogni nodo di mantenere il proprio clock locale, ma allo stesso tempo necessita di stimare l'offset relativo e lo skew relativo con gli altri nodi della rete. Questo permette in ogni istante di convertire il tempo di un nodo nel tempo locale di un qualsiasi altro nodo della rete, come in [9].
Il terzo metodo permette di mantenere per ciascun nodo un clock che è sempre sincronizzato con un clock di riferimento della rete. Lo scopo è quello di mantenere una scala globale univoca in tutta la rete come in [13],[11] e come sarà proposto nell'algoritmo Average TimeSync (ATS).

2.5 Clock Software

Un algoritmo di sincronizzazione può modificare direttamente il clock (fisico) locale τ di un nodo oppure può, basandosi su di esso, limitarsi a costruire un clock software $\hat{\tau}$. Il clock software è semplicemente una funzione che trasforma il clock locale $\tau(t)$ in $\hat{\tau}(\tau(t))$ attraverso una formula matematica. La funzione $\hat{\tau}(\tau(t))$ è generalmente una funzione continua e strettamente monotona crescente come ad esempio $\hat{\tau}(\tau(t)) = a\tau(t) + b$ con a e b due generici parametri.

Capitolo 3

Stato dell'arte - Algoritmi di Sincronizzazione

La *Sincronizzazione Temporale* è un servizio chiave per molte applicazioni riguardanti il calcolo distribuito. Possiamo affermare che la sincronizzazione temporale è un'area di ricerca che possiede una lunga storia. Molti protocolli e algoritmi sono stati proposti finora per soddisfare la necessità di rendere una rete di dispositivi sincronizzata, sia che quest'ultima sia una wired o una wireless network. Uno dei primi e più diffusi e avanzati algoritmi in materia è Network Time Protocol (NTP) ideato da Mills[2] nei primi anni Ottanta del Novecento e utilizzato prettamente per sincronizzare un rete di computer. Nel 2002 venne introdotto il protocollo IEEE 1588 meglio conosciuto come Precision Time Protocol (PTP) [5] ideato per sistemi locali che richiedono un'accuratezza superiore a quella ottenibile con NTP. Tuttavia, come vedremo più avanti, molte delle caratteristiche di una WSN precludono l'utilizzo delle esistenti tecniche di sincronizzazione in questo dominio.

3.1 Network Time Protocol (NTP)

NTP [2, 3, 4] fu disegnato per soddisfare grandi network, con topologia piuttosto statica (come ad esempio Internet). In NTP i nodi della rete sono sincronizzati ad una sorgente di tempo globale, che è iniettata nella rete attraverso un sotto-

gruppo di nodi master (chiamato “stratum-1” server). Questi nodi master sono sincronizzati direttamente ad una sorgente esterna di tempo come ad esempio un dispositivo GPS (tali dispositivi formano lo “stratum-0”). Tutta la rete è strutturata gerarchicamente a livelli come si vede in figura 3.1, dove i nodi foglia sono chiamati clients, mentre i nodi interni sono chiamati stratum-L servers, dove L è il livello del nodo nella gerarchia. Ogni nodo deve specificare in un file di configurazione quali sono i suoi nodi padre. I nodi scambiano frequentemente messaggi di sincronizzazione con i loro padri ed usano le informazioni ottenute per aggiornare regolarmente il proprio clock.

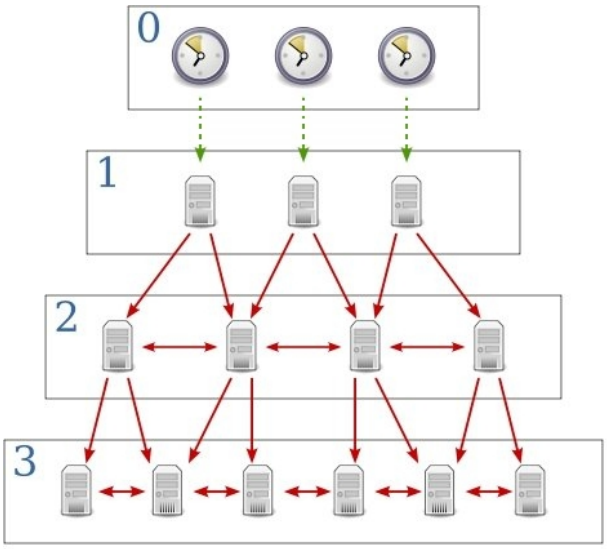

Figura 3.1. Le frecce tratteggiate indicano una connessione diretta, mentre le frecce continue indicano una connessione di rete.

Energia e le altre risorse Le applicazioni per reti di sensori spesso richiedono che il sensore stesso sia di piccole dimensioni e dal costo contenuto. Queste caratteristiche hanno diverse implicazioni. La principale è rappresentata dall’ammontare di energia disponibile, che per nodi di piccole dimensioni è molto limitata a causa della ridotta densità di energia disponibile attualmente (batterie). Per assicurare longevità al singolo nodo di una WSN sono stati adottati sia accorgimenti hardware che software per rendere il nodo energicamente efficiente. In aggiunta, le risorse di storage, computing e la comunicazione sono limitate da-

gli stessi vincoli di dimensione ed efficienza energetica. Inoltre si preclude l'uso di GPS come riferimento esterno di tempo necessario ai nodi stratum-1 sia per un fattore di efficienza energetica sia per il costo proibitivo che risulterebbe dotando i nodi di una rete di un rilevatore GPS per la sincronizzazione temporale. Tipicamente le WSN sono composte da nodi con capacità di batterie, di calcolo e di salvataggio dati limitate, quindi caratterizzati da risorse di costo ridotto. Effettivamente NTP non è ottimizzato per il risparmio energetico perché non è questo un problema per le network per cui era stato ideato. Infatti le normali reti di computer non necessitano di speciali politiche atte a ridurre i consumi energetici perché ne hanno una disponibilità pressoché illimitata. NTP utilizza il processore e la rete in modo da portare ad un dispendioso overhead se implementato su una WSN. Ad esempio i nodi del livello L devono essere sempre in ascolto di eventuali richieste provenienti dai nodi di livello L-1. In una WSN restare in ascolto è un'operazione dispendiosa tanto quanto la trasmissione di dati, per questo molti protocolli per WSN permettono di spegnere la radio quando possibile.

3.2 IEEE 1588 Precision Time Protocol (PTP)

IEEE 1588 è uno standard più comunemente conosciuto con il nome di *Precision Time Protocol* (PTP). Pubblicato nel Novembre 2002 e basato sul lavoro fatto da John Eidson presso Agilent Labs. IEEE 1588 specifica hardware e software per permettere a dispositivi connessi in rete (clients) di sincronizzare i loro clocks ad un clock di rete detto master. Lo standard è stato inizialmente sviluppato per l'ambiente dell'automazione industriale dove precedentemente non era possibile un preciso controllo utilizzando una rete Local Area Networks (LAN). In seguito è risultato interessante nell'ambito delle telecomunicazioni, nell'ambito energetico e militare. Lo standard è applicabile alle Local Area Networks che supportano comunicazioni di tipo multicast (includendo e non limitandosi a Ethernet). Da parte di IEEE sono in corso dei tentativi di estendere tale protocollo alle Wireless Local Area Network come presentato in [8]; è quindi possibile ipotizzare

un'implementazione di IEEE 1588 su reti di sensori wireless (anche se vedremo non risulterà adatto).

IEEE 1588 è un protocollo Master/Slaves che si basa sullo scambio di una serie di pacchetti tra un *master clock* e diversi *slave clocks*. IEEE 1588 è in grado di sincronizzare sistemi eterogenei (con clocks che variano per precisione, risoluzione e stabilità) con precisione superiore al microsecondo. Per raggiungere tale precisione esso richiede che il timestamp di spedizione/arrivo dei messaggi sia generato da uno specifico hardware o da una componente più vicina possibile al mezzo fisico (come avviene per i sensori wireless con il timestamp a livello MAC), al contrario di NTP che agisce solo a livello software e non richiede alcun particolare hardware. Informazioni più dettagliate su IEEE 1588 si possono trovare in [5], [6], [7].

Per sincronizzare dei dispositivi con IEEE 1588 sono necessari due operazioni:

- determinare quale dispositivo avrà la funzione di *master clock*.
- misurare e correggere la divergenza degli *slave clocks* dovute ai loro offset iniziali e ai ritardi della rete.

Quando il sistema è inizializzato il protocollo utilizza l'agoritmo denominato "Best Master Clock Algorithm" per determinare automaticamente quale clock nella rete è il più preciso, ed esso verrà nominato *master clock*. Tutti gli altri clock si sincronizzeranno con questo *master clock*.

Pincipio di Sincronizzazione IEEE 1588 fu sviluppato per sfruttare i sistemi Ethernet come mezzo per raggiungere la sincronizzazione. In questi sistemi a causa del protocollo di gestione dei frames (CSMA/CD)[1], può accadere che un pacchetto venga ritardato o persino eliminato. Per questo motivo lo standard descrive una procedura che tiene conto di queste problematiche.

Per prima cosa, un nodo (*master clock*) invia un messaggio detto "Sync message" registrando a parte l'esatto istante di tempo in cui tale invio viene effettuato, questo valore viene trasmesso in un secondo messaggio chiamato "Follow-Up

[1]È l'acronimo inglese per Carrier Sense Multiple Access with Collision Detection ovvero accesso multiplo tramite rilevamento della portante con rilevamento delle collisioni.

message". Il ricevitore utilizza il proprio clock locale per registrare l'istante di arrivo di "Sync message" e lo confronta con l'effettivo istante di trasmissione contenuto nel "Follow-Up message". La differenza tra i due valori di tempo rappresenta l'offset dello *slave clock* più il ritardo di propagazione del messaggio. Tale ritardo viene calcolato dal nodo ricevitore inviando al *master clock* un messaggio chiamato "Delay-Request message" registrando l'istante di invio di tale messaggio. Quando il *master clock* riceve un "Delay-Request message", ne registra l'istante di arrivo e lo salva su un messaggio chiamato "Delay-Response message" che viene inoltrato allo *slave clock*.

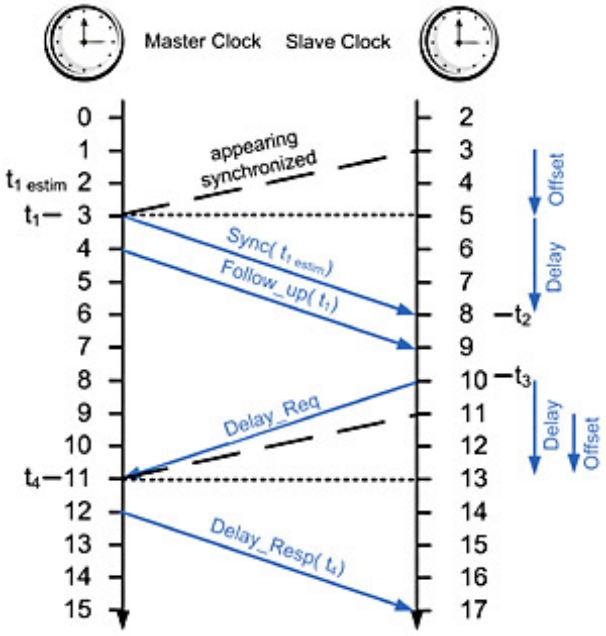

Figura 3.2. Scambi necessari per la sincronizzazione tra un *clock master* e un *clock slave* in PTP.

A questo punto lo *slave clock* può modificare il proprio clock in accordo con il *master clock* avendo tutte le informazioni necessarie di seguito riportate:

- offset= istante di ricezione - istante di trasmissione - ritardo di propagazione (relativo a "Sync message")
- ritardo di propagazione = (ritardo tra master e slave + ritardo tra slave e master)/2 (si assume un ritardo simmetrico)

- ritardo tra master e slave = istante di ricezione - istante di trasmissione (relativo a “Sync message”)
- ritardo tra slave e master = istante di ricezione - istante di trasmissione (relativo a “Delay-Request message”)

A causa dell’indipendente divergenza dei clock coinvolti nelle procedura, è necessaria una ripetizione periodica di correzione dell’offset e del ritardo di propagazione per mantenere i clock sincronizzati. In ambito di reti di sensori wireless, lo stesso discorso fatto per NTP vale anche per PTP, infatti l’algoritmo originale non sembra adatto ad un’implementazione diretta su una rete di sensori. La sincronizzazione tra *clock master* e *clock slaves* viene effettuata con la sola compensazione dell’offset e senza nessuna stima dello skew relativo, ciò comporta una resincronizzazione continua dei nodi della rete con periodo proporzionale alla precisione richiesta. Inoltre ogni nodo si sincronizza direttamente con il *clock master*, il quale deve gestire le richieste di tutti i nodi in maniera indipendete. Al crescere del numero di nodi della rete il lavoro eseguito dal *clock master* aumenta implicando problemi di scalabilità, riguardanti sia le operazioni di trasmissione/ricezione dei messaggi da parte del *clock master* sia dalla necessità della presenza di un collegamento diretto tra nodi master e slaves.

3.3 Algoritmi per WSN

Nell’ultimo decennio sono state molte le ricerche e le pubblicazioni sulla sincronizzazione temporale per le reti di sensori wireless. Di seguito saranno illustrati i più noti algoritmi di sincronizzazione conosciuti in letteratura. Lo scopo è quello di dare una panoramica delle tecniche utilizzate.

3.3.1 Reference Broadcast Synchronization (RBS)

Uno dei primi algoritmi ideati in materia è stato proposto da Elson, Girod e Estrin [9] e chiamato Reference Broadcast Synchronization Protocol (RBS). In RBS ogni nodo è normalmente non sincronizzato con il resto della rete. Un nodo segnalatore (beacon node) periodicamente invia in modalità broadcast dei

pacchetti di riferimento detti beacon ai nodi all'interno del suo raggio di trasmissione. L'evento di ricezione del pacchetto beacon è utilizzato come riferimento di tempo per la stima di offset e skew dei nodi vicini. Quando un nodo riceve un pacchetto beacon registra il tempo di arrivo del pacchetto effettuando il timestamp in accordo al proprio clock locale. In seguito scambia il proprio timestamp con il resto dei nodi. Ogni nodo utilizza i pacchetti contenenti i timestamps e la regressione lineare per stimare gli offset e drift relativi ai nodi vicini.

La cosa interessante di RBS è che registra il timestamp solo nei nodi ricevitori, infatti non interessa il tempo in cui il nodo di riferimento invia il pacchetto beacon. Per cui tutti gli errori di incertezza dal lato trasmettitore sono eliminati. Questa caratteristica rende l'algoritmo particolarmente adatto a tutti quei dispositivi hardware che non forniscono un timestamp a livello MAC. Il meccanismo proposto con RBS può essere facilmente esteso a multi-hop network. Lo svantaggio principale di RBS risulta nel fatto che non si verifica direttamente una sincronizzazione tra nodo trasmettitore e ricevitore dei pacchetti beacon. Inoltre un numero elevato di scambi di messaggi è richiesto per raggiungere la sincronizzazione.

3.3.2 Tiny-Sync e Mini-Sync (TS/MS)

Tiny-Sync e Mini-Sync [10] sono entrambi algoritmi di sincronizzazione che sfruttano la trasmissione di informazioni tra coppie di nodi (pairwise synchronization). Gli autori assumono che ciascun clock può essere approssimato con il seguente modello a frequenza costante:

$$t_i(t) = a_i t + b_i \tag{3.1}$$

dove a_i e b_i sono *drift* e *offset* del clock del nodo i. Considerando due nodi 1 e 2 con i loro rispettivi clock hardware $t_1(t)$ e $t_2(t)$ dalla 3.1 segue che t_1 e t_2 sono linearmente legati da:

$$t_1(t) = a_{12} t_2(t) + b_{12} \tag{3.2}$$

dove a_{12} e b_{12} rappresentano il *drift* relativo e l'*offset* relativo tra i due nodi.

Entrambi gli algoritmi usano valori di *round-trip time*[2] per ottenere stime di offset e drift tra coppie di nodi. La raccolta di tali informazioni (*data collection*)

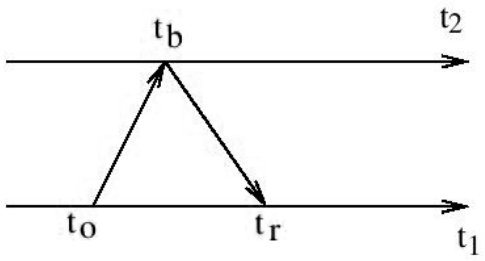

Figura 3.3. Un messaggio "probe" dal nodo 1 è immediatamente ritornato dal nodo 2. I timestamps sono registrati ad ogni ricezione/invio di messaggi ottenendo il data-point (t_0,t_b,t_r).

avviene con una semplice procedura visualizzata in figura 3.3 e di seguito descritta. Il nodo 1 manda al nodo 2 un messaggio "probe" e registra l'istante di tale invio corrispondente al tempo t_0. Quando il nodo 2 riceve il messaggio genera il timestamp t_b in accordo col proprio clock locale e immediatamente spedisce indietro al nodo 1 un messaggio di risposta contenente tale valore. Alla ricezione della risposta del nodo 2, il nodo 1 registra il timestamp t_r. Sfruttando l'ordine degli eventi descritti (corrispondenti alla ricezione/trasmissione di un messaggio), i relativi timestamps e la relazione 3.2 è possibile ricavare le seguenti sistema di disuguaglianze:

$$t_0 < a_{12} t_b + b_{12} \tag{3.3}$$

$$t_r > a_{12} t_b + b_{12} \tag{3.4}$$

I tre valori dei timestamps (t_0,t_b,t_r) formano un "data point". Entrambi gli algoritmi lavorano con un gruppo di data points collezionati con lo scambio precedentemente spiegato e permettono di fare un'analisi di drift e offset come rappresentato in figura 3.4. Le linee tratteggiate in figura 3.4 soddisfano l'equazione 3.2 e rappresentano i vincoli imposti dai data points. Una è la retta con pendenza maggiore e offset minore ($\overline{a_{12}}$ e $\underline{b_{12}}$) mentre l'altra è la retta con pendenza minore e offset maggiore ($\underline{a_{12}}$ e $\overline{b_{12}}$). Il drift relativo e l'offset relativo

[2]Tempo necessario ad un messaggio per andare e tornare da un nodo all'altro della rete.

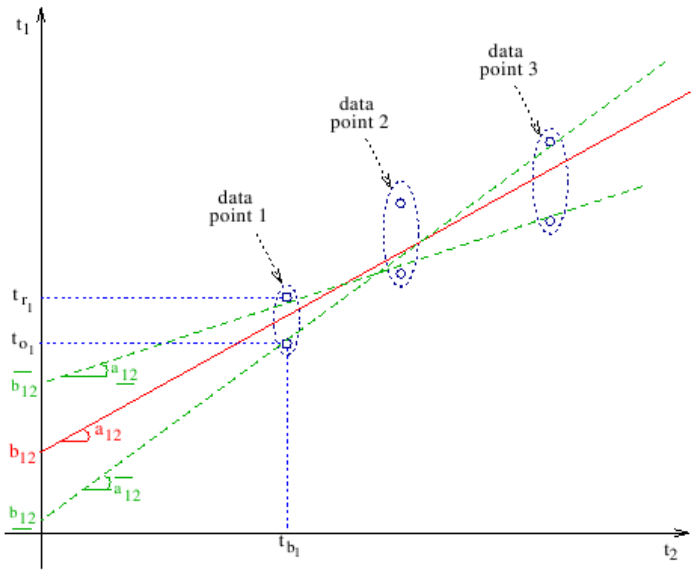

Figura 3.4. Dipendenze lineari e vincoli imposti ad a_{12} e b_{12} da tre data points

di una coppia di nodi sono limitati dalle seguenti disuguaglianze:

$$\underline{a_{12}} \leq a_{12} \leq \overline{a_{12}}$$
$$\underline{b_{12}} \leq b_{12} \leq \overline{b_{12}}$$

Possiamo dire che più vicini sono i limiti $([\overline{a_{12}}, \underline{a_{12}}][[\overline{b_{12}}, \underline{b_{12}}])$, più la stima di offset e drift è buona e la precisione della sincronizzazione è maggiore. In questo modo non possono essere determinati esattamente i valori di offset e drift, ma possono essere ben stimati, anzi più data points colleziona un nodo più la stima dei parametri può essere accurata. Ovviamente i limiti di memoria e di computazione dei dispositivi limitano il numero di data points disponibili per nodo. Per questo *TinySync* e *MinySync* sfruttano il fatto che non tutti i data points sono utili per la stima. Come possiamo vedere in figura 3.4 il data point 2 non ha nessuna influenza sulla stima dei parametri delle disuguaglianze precedenti. Questa analisi è sfruttata in *TinySync* il quale mantiene in memoria solamente i due data points che danno i migliori bounds, tuttavia la stima non è sempre la migliore possibile. *MiniSync* è l'estensione di *TinySync*, infatti esso cerca la

soluzione ottima con relativo aumento della complessità dell'algoritmo.

3.3.3 Time-Sync Protocol for Sensor Network (TPSN)

Ganeriwal *et al.* [11] proposero un loro protocollo di sincronizzazione per WSN chiamato Time-Sync Protocol for Sensor Network (TPSN) [11]. Il protocollo è suddiviso in 2 fasi: la prima chiamata level discovery phase e la seconda synchronization phase. Lo scopo della prima fase è quello di costituire una topologia gerarchica nella rete strutturata in livelli. In questa fase ad ogni nodo viene assegnato un livello. Solo ad un nodo viene assegnato il livello 0, questo nodo viene chiamato nodo radice (root node). Nella seconda fase avviene la sincronizzazione della rete secondo la seguente procedura: un nodo del livello i si sincronizza ad un nodo di livello $i-1$. Alla fine della fase di sincronizzazione, tutti i nodi sono sincronizzati al nodo radice, così si raggiunge la sincronizzazione globale della rete.

Level Discovery Phase Questa fase è eseguita una volta sola al momento del posizionamento dei nodi della rete (o al variare della topologia di rete). Per prima cosa deve essere stabilito un nodo radice, questo nodo può essere dotato di un ricevitore GPS in modo tale da sincronizzare la rete con una sorgente temporale esterna. Per l'elezione del nodo radice può venir utilizzato un algoritmo o può essere stabilito a priori. Al nodo radice viene assegnato il livello 0 ed inizia a trasmettere in modalità broadcast un pacchetto chiamato di *level-discovery* e contenente l'identità e il livello del nodo mittente. I nodi vicini al nodo radice che ricevono questo pacchetto si assegnano livello 1. In seguito ogni nodo di livello 1 inoltra un pacchetto *level-discovery* con il proprio livello e l'identificativo. Quando ad un nodo è assegnato il proprio livello, esso scarta ulteriori pacchetti *level-discovery* in modo da mantenere stabile la struttura della rete. Questa catena di trasmissione attraversa tutta la rete di nodi e la fase è completa quando a tutti i nodi è stato assegnato un livello.

Synchronization Phase La struttura base di questa fase è costituita da uno scambio di messaggi tra una coppia di nodi, come mostrato in figura 3.5. Gli autori assumono che la deriva del clock tra una coppia di nodi sia

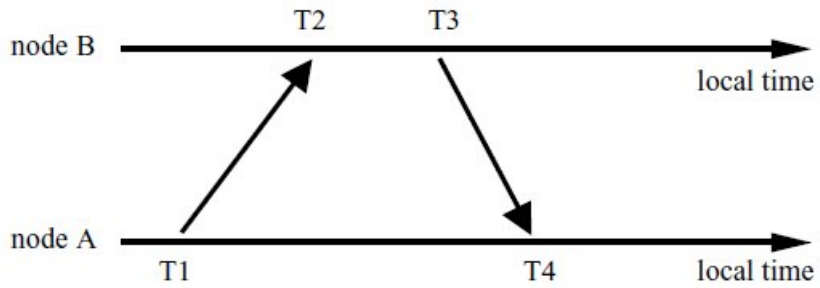

Figura 3.5. doppio scambio di messaggi tra due nodi

costante nell'intervallo di tempo necessario per un singolo scambio di messaggi. Inoltre si assume che il ritardo di propagazione sia costante e uguale in entrambe le direzioni. Considerando lo scambio di messaggi illustrato in figura 3.5, il nodo A inizia la sincronizzazione inviando un messaggio chiamato *synchronization-pulse* contenente il tempo di spedizione T1 in accordo con il proprio clock locale e il livello del nodo. Il nodo B riceve tale pacchetto in accordo al proprio clock locale al tempo T2 = T1 + Δ + δ. Dove Δ è la deriva relativa tra i clock dei due nodi, mentre δ è il ritardo di propagazione tra i nodi. Il nodo B risponde nell'istante T3 con un messaggio di acknowledgement che include il livello di B e i tempi T1, T2 e T3. Il nodo A è così in grado di calcolare la deriva relativa dei clock Δ e il ritardo di propagazione δ come dimostrato sotto, e in questo modo A può sincronizzarsi a B.

$$\Delta = \frac{(T2 - T1) - (T4 - T3)}{2}; \quad \delta = \frac{(T2 - T1) + (T4 - T3)}{2};$$

È il nodo radice che inizia la fase di sincronizzazione inviando un pacchetto *time-sync* ai nodi di livello 1. Alla ricezione di questo pacchetto i nodi di livello 1 iniziano lo scambio di messaggi con il nodo radice. Ogni nodo di livello 1 attende per un tempo casuale prima di iniziare lo scambio con la radice in modo da minimizzare eventuali collisioni di messaggi nel canale wireless. Una volta ricevuto l'acknowledge dal nodo radice, essi aggiustano il loro clock a quello del nodo radice. I nodi a livello 2, captando i messaggi

di sincronizzazione tra i nodi di livello 1 e il nodo radice, inizieranno un altro scambio di messaggi sempre dopo aver aspettato per un tempo casuale in modo da assicurare che i nodi di livello 1 si siano sincronizzati completamente. La procedura attraversa tutti i livelli della rete sincronizzandola con il nodo radice.

3.3.4 Lightweight Time Synchronization for Sensor Network (LTS)

LTS [12] è una variazione del protocollo TPSN [11] in cui viene sacrificata la precisione della sincronizzazione per guadagnare in minor costo di energia necessaria a tale sincronizzazione. Diversamente da molte altre tecniche che tendono alla massima precisione, in LTS le necessità di comunicazione e calcolo per la sincronizzazione del singolo nodo sono state significativamente ridotte traendo vantaggio dal rilassamento del vincolo di accuratezza. Lo schema proposto sacrifica l'accuratezza eseguendo le operazioni di sincronizzazione con minor frequenza e tra alcuni nodi. Sono proposti due algoritmi: entrambi assumono l'esistenza di

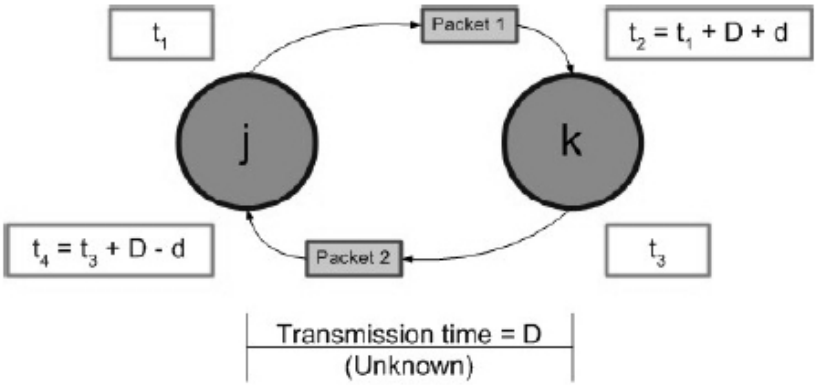

Figura 3.6. scambio di pacchetti per una sincronizzazione pair-wise

almeno un nodo che abbia accesso ad una sorgente di riferimento di tempo globale. Nel primo algoritmo si usa un approccio *centralizzato* dove gli aggiornamenti periodici e la sincronizzazione sono gestiti da un nodo radice. La base dell'algoritmo è la costruzione di un albero di coperture (spanning tree) T a profondità

minima che comprende i nodi della rete. Il nodo radice inizia la sincronizzazione sincronizzandosi con tutti i suoi figli in T (single-hop) sfruttando il round-trip time di un messaggio come mostrato in figura 3.6. Successivamente ogni figlio del nodo radice si sincronizza con i propri figli. Questo processo continua finché i nodi foglia di T sono stati sincronizzati. Il tempo necessario per un ciclo completo dell'algoritmo è proporzionale alla profondità di T. Un nuovo spanning-tree T è costruito ogni volta che l'algoritmo viene eseguito. La frequenza di sincronizzazione è calcolata in base alla precisione richiesta, alla profondità dell'albero T e al limite di deriva del clock ρ_{max}.
Nell'algoritmo *decentralizzato* si usa lo schema distribuito dove i singoli nodi sono responsabili per iniziare ad eseguire la resincronizzazione. Nessun tipo di spanning-tree è necessario a dirigere la comunicazioni tra nodi.
Quando un nodo necessita di esser sincronizzato, spedisce un messaggio di richiesta al nodo di riferimento più vicino utilizzando un qualsiasi algoritmo di routing. La frequenza di sincronizzazione è calcolata in base alla accuratezza richiesta, la distanza in numero di hop dal nodo di riferimento e in base al limite di deriva del clock ρ_{max}.

3.3.5 Flooding Time Syncronization Protocol (FTSP)

FTSP[13] può esser utilizzato per sincronizzare un'intera network e necessita che ogni nodo venga identificato univocamente con un codice ID. Il nodo con ID più basso è eletto nodo radice e serve come sorgente di riferimento del tempo globale della rete. Se questo nodo viene a mancare, il nodo con ID più basso nella rete rimanente viene eletto nuovo nodo radice. Il nodo radice periodicamente inoltra in modalità broadcast un messaggio di sincronizzazione contenente il relativo timestamp di invio. I nodi che non hanno ancora ricevuto questo messaggio ne registrano il timestamp contenuto ed il tempo di arrivo del messaggio in accordo con la loro stima del tempo globale. In seguito aggiornano la propria stima del tempo globale e inoltrano il messaggio ai nodi vicini. Il timestamp è realizzato a livello MAC per minimizzarne i ritardi variabili dovuti ai livelli superiori. Ciascun nodo colleziona otto coppie di (timestamp, tempo di arrivo) dei messaggi di sincronizzazione e utilizza la regressione lineare di questi punti

per stimare la differenza di offset e frequenza con il nodo radice.

3.3.6 Asynchronous Diffusion (AD)

In AD [14] viene proposta una tecnica di sincronizzazione per WSN che si basa sull'univocità del valore medio dei clock di una rete di nodi. Li e Rus propongono un semplice algoritmo nel quale ogni nodo, quando vuole sincronizzarsi, spedisce in broadcast messaggi di sincronizzazione ai propri vicini, i quali rispondono con un messaggio contenente il loro tempo locale. Il nodo che ha iniziato la procedura, dopo aver ricevuto i timestamps dei nodi vicini, ne esegue la media e rispedisce questo valore in modo che i vicini possano aggiornare il loro valore di clock. Si assume che la sequenza di operazioni sopra descritte sia atomica per permettere il corretto calcolo della media dei clock. Si assume il funzionamento dell'algoritmo in una rete di nodi fortemente connessa. Viene inoltre dimostrata la convergenza dell'algoritmo anche se nessuna implementazione è presentata.

3.3.7 Reachback Firefly Algorithm (RFA)

Werner Allen et al. propongono in [15] un innovativo algoritmo decentralizzato. Il modello matematico utilizzato è derivato dallo studio che descrive la sincronizzazione spontanea delle lucciole o dei neuroni. Partendo dal modello iniziale viene proposto un algoritmo modificato con una serie di accorgimenti per far fronte alle caratteristiche di una rete di sensori wireless. Viene definito il concetto di sincronia (synchronicity) come l'abilità di organizzare azioni simultanee e collettive in una rete di sensori. La sincronizzazione richiede che tutti i nodi condividano una nozione comune di tempo, mentre la sincronia richiede che i nodi concordino nel periodo di aggiornamento del clock e nella fase. Sincronia e sincronizzazione vengono descritte come caratteristiche complementari; infatti, i nodi che hanno accesso a un tempo comune, possono organizzare azioni collettive nel futuro, e viceversa, i nodi che possono eseguire azioni collettive possono stabilire una significativa base temporale per tutta la rete.

Nell'algoritmo ogni nodo agisce come un oscillatore con periodo fisso T. Ogni nodo ha un tempo interno t che comincia da zero e incrementa costantemente fino ad arrivare a $t = T$. A questo punto il nodo emette un segnale e resetta il

tempo, $t = 0$. I nodi possono avviarsi in istanti differenti, perciò il tempo interno differisce. In assenza di input dai nodi vicini, il nodo emette il segnale ogni volta che $t = T$. Se il nodo recepisce dei segnali emessi dai vicini, allora reagisce aggiustando la fase del proprio tempo interno. L'aggiustamento è determinato da una funzione $f(t)$ chiamata *firing function* e da un parametro costante $\epsilon < 1$. Se il nodo osserva un segnale del vicino al tempo $t = t'$, istantaneamente corregge il proprio tempo interno al tempo $t = t''$ dove:

$$t'' = f^{-1}(f(t') + \epsilon) \tag{3.5}$$

Nel caso $t'' > T$ allora viene impostato $t = T$ e immediatamente invia il suo segnale. L'algoritmo ha l'effetto di aumentare la fase del nodo di $\Delta(t') = t'' - t'$ quando il nodo osserva un segnale da un vicino all'istante $t = t'$. Se la funzione f è continua, monotonica crescente e concava allora l'insieme dei nodi della rete convergono alla stessa fase per qulasiasi istante iniziale di accensione e tale comportamento si ottiene anche in reti multihop.

3.3.8 Solis, Borkar, Kumar protocol

Solis et al. (2006) in [17] presentano un algoritmo distribuito per sincronizzare reti wireless multihop. Non necessita di nodi radice e sfrutta solamente le comunicazioni tra nodi vicini. L'idea centrale, che si differenzia dalle precedenti tecniche, è di migliorare le prestazioni della sincronizzazione sfruttando il gran numero di vincoli che devono essere soddisfatti dalla comune nozione di tempo in una WSN.

Essendo O_{ij} l'offset del clock del nodo i rispetto al clock del nodo j, allora la stima $\hat{O}_{ij}$ può essere calcolata tra i nodi i e j con uno scambio di pacchetti contenenti i timestamp d'invio dei messaggi. La stima $\hat{O}_{ij}$ è affetta da rumore ed è basata solo sui valori dei timestamp scambiati tra i due nodi. L'algoritmo propone di raffinare questa stima utilizzando dei vincoli globali; ad esempio, considerando qualsiasi ciclo tra i nodi della rete $i_1, i_2, i_3, i_4 ... i_n = i_1$ si ha che $\sum_{k=1}^{n} O_{i_k i_{k+1}} = 0$, cioè la somma degli offset lungo il ciclo è nulla. Ogni nodo stima con il solito scambio di pacchetti bilaterale l'offset e lo skew relativo ai propri nodi vicini. Queste informazioni sono necessarie per la successiva stima

v_i del'offset relativo tra il nodo i ed un nodo di riferimento della rete (che può cambiare saltuariamente). La stima ad ogni ciclo dell'algoritmo andrà aggiornata aggiungendo un coefficiente δ_i che dipenderà dalla quantità $v_j + \hat{O}_{ji}$ del nodo j, per ogni nodo j vicino del nodo i. Lo stesso concetto è utilizzato per la stima dello skew $\hat{\alpha}_{ij}$; visto che il prodotto dello skew attorno ad un ciclo deve essere uguale a 1, basta utilizzare il precedente procedimento avendo cura di sostituire $\log(\hat{\alpha}_{ij})$ ad $\hat{O}_{ji}$

3.3.9 Simeone and Spagnoli protocol

Simeone et al. (2007) [16] propongono un protocollo distribuito e in grado di portare alla convergenza globale dei tempi la rete di K nodi. Ogni nodo ha un clock discreto con periodo T_k dove l'evoluzione è descritta da $t_k(n) = nT_k + \tau_k(0)$, dove $0 \leq \tau_k(0) < T_k$ e $\tau_k(0)$ è la fase iniziale del nodo mentre $n = 1, 2, 3...$.

Per raggiungere la sincronizzazione un segnale è trasmesso all'istante $t_k(n)$ dal nodo k che viene ricevuto dagli altri nodi. Si assume che tutti i nodi trasmettano con la stessa potenza e che per ogni segnale ricevuto da nodo n il nodo i sia in grado di calcolare la differenza di tempo $t_i(n) - t_k(n)$ e la potenza del segnale $P_{ki}(n)$. Al n-esimo periodo il k-esimo nodo aggiorna il proprio clock $t_k(n)$ come segue:

$$\begin{aligned} t_k(n+1) &= t_k(n) + \epsilon \Delta t_k(n+1) + T_k \\ \Delta t_k(n+1) &= \sum_{i=1, i \neq k}^{K} \alpha_{ki}(t_i(n) - t_k(n)) \end{aligned}$$

la scelta interessante fatta dagli autori è di considerare maggiormente i segnali provenienti da nodi con potenza più forte, quindi da nodi con canale più affidabile. Ciò può essere fatto utilizzando

$$\alpha_{ij}(n) = \frac{P_{ki}(n)}{\sum_{j=1, j \neq k}^{K} P_{kj}(n)}$$

Si dimostra che la convergenza dell'algoritmo è soddisfatta se e solo se la rete di sensori è fortemente connessa, cioè se esiste un cammino che collega ciascuna coppia di nodi. Partendo dall'assunzione $t_k(n+1) - t_k(n) = T$ per ciascun

nodo k della rete (equivale a dire che la frequenza dei clock è la stessa per tutti i nodi) l'algoritmo permette di far convergere i clock per qualsiasi valore dello sfasamento iniziale $t_k(0)$. Se si assume invece che la velocità dei clock dei nodi sia diversa, allora l'algoritmo porta alla convergenza di tutti gli skew dei nodi lasciando i clock ad un asintotico errore di fase (le velocità dei clock diventano uguali, ma non si compensa la diversa fase iniziale).

Capitolo 4

Algoritmo Proposto (ATS)

Verrà di seguito descritto e testato il protocollo denominato *Average TimeSync* (ATS) [18] che ha come obiettivo la sincronizzazione di una rete di sensori wireless utilizzando la teoria dell'*average consensus* [1]. L'algoritmo fa parte della classe degli algoritmi distribuiti, conosciuti con i nomi di *consensus, agreement, gossip o rendez-vous* che si basano sull'idea di utilizzare la media delle loro informazioni. L'algoritmo utilizza un metodo di *diffusione* dell'informazione che è completamente distribuito e localizzato. La sincronizzazione è fatta localmente senza il bisogno di nodi globali che la inizializzino. Perciò ATS risulta molto robusto in caso di guasto di alcuni nodi. Non vi è distinzione tra i nodi (nessun nodo *master* o *slave*). Tutti i nodi operano nella stessa maniera secondo l'architettura peer-to-peer; mediante questa configurazione qualsiasi nodo è in grado di avviare o completare una sessione di sincronizzazione.

A differenza di FTSP [13] non si sfrutta il flusso (*flooding*) d'informazione da un nodo radice ai nodi foglia, anzi l'informazione è contenuta in tutti i nodi, che tramite scambi di pacchetti tra vicini ne mediano i valori. Il metodo di *diffusione* permette di raggiungere una sincronizzazione globale tramite una distribuzione della sincronizzazione locale all'intero sistema. L'algoritmo è in grado di sincronizzare la rete facendo in modo che ogni nodo della rete modifichi il proprio clock software al valore di *consensus* stabilito. La possibilità più semplice consiste nel

[1] I problemi di "consenso" si occupano di far convergere delle variabili di stato ad un parametro comune.

scegliere il più basso o il più alto valore di clock in tutta la rete. Questo tipo di sincronizzazione necessita di un semplice algoritmo. Tuttavia questa soluzione non è robusta, perché a seguito di un guasto o dell'aggiunta di nuovi nodi nella rete si possono introdurre valori anomali di clock (o troppo bassi o troppo alti) che probabilmente viziano la sincronizzazione raggiunta dalla rete sino a quell'istante. Per rendere il sistema più robusto e resistente ai guasti l'algoritmo proposto utilizza la media globale delle letture dei clock come valore di *consensus*.

L'idea principale dell'algoritmo è di livellare tutti i valori dei clock software di ciascun nodo alla loro media globale. Un nodo con valore di clock basso assorbe alcune quantità di valori dai nodi vicini ed incrementa il proprio valore. Dopo alcuni cicli di diffusione il clock di ciascun nodo avrà lo stesso valore.

4.1 Descrizione dell'algoritmo

Nel protocollo ATS la dinamica del clock di ogni nodo è modellizzata linearmente rispetto al tempo. L'orologio locale del nodo i-esimo risponde alla legge

$$\tau_i(t) = \alpha_i t + \beta_i \tag{4.1}$$

in cui τ_i è il valore del clock, α_i il coefficiente di deriva, β_i quello di offset. Dato che il valore del tempo t non è accessibile al nodo, esso non è in grado di fornire i parametri α_i e β_i. Per ovviare a questo problema si è scelto di ottenere l'informazione indiretta su questi due parametri misurando il clock locale del nodo i-esimo rispetto a un altro nodo j-esimo. Dall'equazione 4.1 otteniamo $t = \frac{\tau_i - \beta_i}{\alpha_i}$; sostituendo questo risultato al nodo j-esimo, si ottiene

$$\begin{aligned} \tau_j &= \frac{\alpha_j}{\alpha_i}\tau_i + \left(\beta_j - \frac{\alpha_j}{\alpha_i}\beta_i\right) \\ &= \alpha_{ij}\tau_i + \beta_{ij} \end{aligned} \tag{4.2}$$

Graficamente l'equazione precedente e la 4.1 sono riportate in Figura 4.1.
Come premesso, lo scopo dell'algoritmo è sincronizzare tutti i nodi rispetto a un

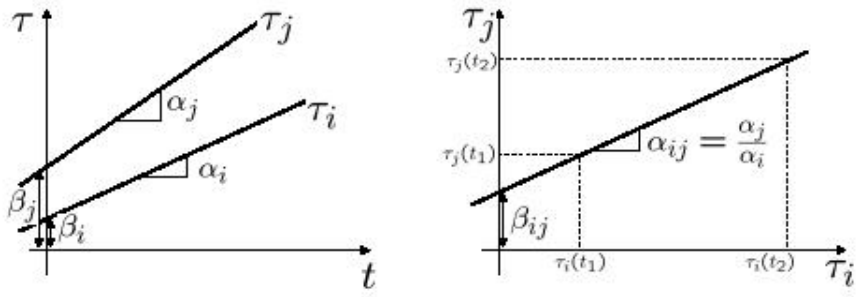

Figura 4.1. Dinamica dei clock rispetto al tempo t (a sinistra) e rispetto ad altri clock (destra)

clock virtuale di riferimento:

$$\tau_v(t) = \alpha_v t + \beta_v \tag{4.3}$$

Ad ogni nodo si cerca di dare una stima della (4.3) usando, ancora una volta, una funzione lineare del proprio clock locale:

$$\hat{\tau}_i = \hat{\alpha}_i \tau_i + \hat{o}_i \tag{4.4}$$

dove $\hat{\alpha}_i$ è la stima della deriva relativa e $\hat{o}_i$ è la stima dell'offset. Ovviamente lo scopo è trovare per ogni nodo i, $\hat{\alpha}_i$ e $\hat{o}_i$ tali che

$$\lim_{t\to\infty} \hat{\tau}_i(t) = \tau_v(t), \quad i = 1, \ldots, N \tag{4.5}$$

Una volta raggiunto tale obiettivo tutti gli N nodi della rete avranno un tempo di riferimento comune dato dal clock virtuale. L'espressione precedente può essere semplicemente riscritta sostituendo 4.1 nella 4.4 ottenendo:

$$\hat{\tau}_i = \hat{\alpha}_i \alpha_i t + \hat{\alpha}_i \beta_i + \hat{o}_i \tag{4.6}$$

perciò il limite 4.5 equivale ad :

$$\lim_{t\to\infty} \hat{\alpha}_i(t) = \frac{\alpha_v}{\alpha_i}, \tag{4.7}$$

$$\lim_{t\to\infty} \hat{o}_i(t) = \beta_v - \hat{\alpha}_i(t)\beta_i = \beta_v - \frac{\alpha_v}{\alpha_i}\beta_i, \quad i = 1, \ldots, N \tag{4.8}$$

In realtà i parametri α_i e β_i non sono costanti, ma variano in base alle condizioni ambientali e all'età del sensore in questione.

Da sottolineare è il fatto che il clock virtuale è un tempo di riferimento fittizio non fissato a priori. I valori assunti dai parametri α_v e β_v non sono rilevanti, l'importante è che tutti gli orologi convergano ad un valore di riferimento comune. In realtà α_v e β_v dipendono dalle condizioni iniziali e dalla topologia della rete.

Il protocollo ATS si basa su timestamps forniti dal livello MAC: con questo tipo di approccio si può inviare la lettura del clock del nodo i acquisita all'istante stesso di inizio della trasmissione. Lo stesso vale per il nodo ricevitore j e la corrispondente lettura del clock di j che può essere acquisita all'arrivo del primo bit inviato da i. Poiché il tempo di propagazione dell'informazione tra i vari nodi è trascurabile (inferiore al microsecondo per distanze sotto i 300 metri), la lettura del clock locale $\tau_i(t_1)$, la trasmissione di pacchetto e la lettura del clock locale $\tau_j(t_2)$ possono essere considerate istantanee, cioè $t_1 = t_2$.

4.2 Stima della deriva relativa

Come illustrato nella modellizzazione del problema in esame, i coefficienti α e β non sono disponibili in maniera assoluta per nessun nodo, quindi nemmeno per alcune coppie (i, j) sono disponibili α_{ij} e β_{ij}.

Ogni nodo i prova a stimare le derive relative α_{ij} rispetto a tutti i nodi j a lui vicini. A tale scopo il nodo j invia un messaggio contenente il tempo locale $\tau_j(t_1)$ di invio, quindi il nodo i che riceve questo pacchetto istantaneamente memorizza il proprio clock locale $\tau_i(t_1)$. Il nodo i registra nella propria memoria la coppia $(\tau_i(t_1), \tau_j(t_1))$. Lo stesso procedimento è applicato quando il nodo j trasmette il nuovo pacchetto a i, che memorizza la nuova coppia $(\tau_i(t_2), \tau_j(t_2))$. La stima della deriva relativa è eseguita come di seguito:

$$\eta_{ij}^{+} = \rho_\eta \eta_{ij} + (1 - \rho_\eta) \frac{\tau_j(t_2) - \tau_j(t_1)}{\tau_i(t_2) - \tau_i(t_1)} \tag{4.9}$$

dove ρ_η è un parametro di tuning $\in [0, 1]$.
Se non si verificano errori di misura e la deriva è costante, si ha

$$\lim_{t\to\infty} \eta_{ij}(t) = \alpha_{ij} = \frac{\alpha_j}{\alpha_i} = \frac{\tau_j(t_2) - \tau_j(t_1)}{\tau_i(t_2) - \tau_i(t_1)} \tag{4.10}$$

per ogni condizione iniziale e per ogni $\rho_\eta \in (0, 1)$.

4.3 Compensazione della deriva

L'idea che sta alla base della compensazione della deriva è quella di modificare le derive di tutti i nodi in modo da farle convergere ad un unico valore α_v. Per far questo si utilizza un algoritmo di consensus distribuito, basato solamente su scambio di informazioni locali.
In tale algoritmo ogni nodo prende la propria stima della variabile globale (in questo caso la stima della deriva del clock virtuale) e la aggiorna mediandola con le stime dei suoi vicini. La stima della deriva del clock virtuale al nodo i-esimo definita in (4.4) si aggiorna, non appena il nodo i riceve il pacchetto di informazione dal nodo j, secondo la seguente formula:

$$\hat{\alpha}_i^+ = \rho_v \hat{\alpha}_i + (1 - \rho_v)\, \eta_{ij} \hat{\alpha}_j \tag{4.11}$$

$\hat{\alpha}_i$ è ovviamente la stima della deriva del clock virtuale dal nodo j. Come condizione iniziale si considera $\hat{\alpha}_i(0) = 1$.
Si deve dimostrare la (4.5) per quel che riguarda la parte relativa alla deriva, che equivale a provare che

$$\lim_{t\to\infty} \hat{\alpha}_i(t) = \frac{\alpha_v}{\alpha_i} \Leftrightarrow \lim_{t\to\infty} \alpha_i \hat{\alpha}_i = \alpha_v \tag{4.12}$$

Definendo $x_i = \hat{\alpha}_i \alpha_i$ e tenendo conto della validità di (4.10), moltiplicando per α_i la (4.11) si ottiene:

$$x_i^+ = \rho_v x_i + (1 - \rho_v)\, x_j \tag{4.13}$$

Considerando il vettore $\mathbf{x} = (x_1, x_2, \ldots, x_N)^T$ e $\mathbf{1} = (1, 1, \ldots, 1)^T$, la (4.13) in forma matriciale diventa:

$$\mathbf{x}^+ = A\mathbf{x} \tag{4.14}$$

con $A \in \Re^{N \times N}$ (con N numero dei nodi) matrice stocastica: infatti essa ha tutti 1 sulla diagonale e 0 altrove, eccezion fatta per la riga i-esima in cui $A_{ii} = \rho_v$ e $A_{ij} = 1 - \rho_v$:

$$A = \begin{bmatrix} 1 & 0 & & \cdots & & 0 \\ 0 & 1 & 0 & & \cdots & 0 \\ & 0 & \rho_v & 0 & 1-\rho_v & \vdots \\ \vdots & & \ddots & 1 & 0 & \\ & & & & \ddots & 0 \\ 0 & & \cdots & & 0 & 1 \end{bmatrix} \tag{4.15}$$

quindi $A_{ij} \geq 0 \; \forall \; i, j$ e $A\mathbf{1} = \mathbf{1}$.

Grazie alle proprietà di A derivanti dalla stocasticità e per il grafo in questione fortemente connesso, ovvero è sempre possibile trovare un cammino di comunicazione tra ogni coppia di nodi (condizione di normale funzionamento per una rete wireless):

$$\lim_{t \to \infty} \mathbf{x} = \alpha_v \mathbf{1} \Leftrightarrow \lim_{t \to \infty} \hat{\alpha}_i \alpha_i = \alpha_v \tag{4.16}$$

$\forall$ nodo i e ciò è proprio quello che si voleva dimostrare.

Come precedentemente anticipato l'esatto valore di consensus α_v dipende dalle condizioni iniziali e dalla topologia della rete.

4.4 Compensazione dell'offset

Grazie a quanto dimostrato nella sezione precedente e sostituendo in 4.6 i risultati ottenuti in 4.12, si ha:

$$\hat{\tau}_i(t) = \alpha_v t + \frac{\alpha_v}{\alpha_i}\beta_i + \hat{o}_i \tag{4.17}$$

Per raggiungere il risultato globale determinato dalla (4.5), a questo punto è necessario solamente compensare possibili errori di offset. Si utilizza ancora

l'algoritmo di consensus per l'aggiornamento della stima dell'offset del clock virtuale:

$$\hat{o}_i^+ = o_i + (1 - \rho_o)\,(\hat{\tau}_j - \hat{\tau}_i) \tag{4.18}$$

con $\hat{\tau}_i$ e $\hat{\tau}_j$ calcolati al medesimo istante.

Con riferimento alla (4.17), affinché la (4.5) sia soddisfatta, bisogna dimostrare che

$$\lim_{t\to\infty} \hat{o}_i + \frac{\alpha_v}{\alpha_i}\beta_i = \beta_v \tag{4.19}$$

$\forall$ nodo i. Definendo $y_i = \hat{o}_i + \frac{\alpha_v}{\alpha_i}\beta_i$ la (4.19) diventa

$$y_i^+ = \rho_o y_i + (1 - \rho_o)\, y_j \tag{4.20}$$

identica nella struttura alla (4.13); come nel caso precedente si dimostra che

$$\lim_{t\to\infty} y_i = \lim_{t\to\infty} \hat{o}_i + \frac{\alpha_v}{\alpha_i}\beta_i = \beta_v \tag{4.21}$$

quindi tutti gli y_i convergono al valore di consensus β_v; anche questo dipende dalle condizioni iniziali e dalla topologia della rete.

Capitolo 5

Analisi della piattaforma architetturale Telosb-TinyOS

5.1 Piattaforma Hardware

Attualmente possono essere reperiti sul mercato svariati tipi di nodi sensore. I più comuni sono:

1. Mica Family (Mica Mica2 Mica2dot MicaZ) da Crossbow Technology Inc [19].

2. Telos Family (Telos, Telosa, Telosb, TmoteSky) da Moteiv Corporation, ora diventata Sentilla [20].

3. Intelmote2 da Intel [21].

4. Eyes Family (EyesIFX EyesIFXv1 EyesIFXv2) da Ember Corporation [22].

L'architettura di questi nodi è molto simile, essenzialmente sono tutti l'evoluzione del nodo nominato *Mica* progettato presso l'Università di Berkeley in California. Il nodo *Mica2* è stato per gran tempo la tecnologia di riferimento del mondo della ricerca e non, anche se attualmente è da considerarsi sorpassato. Verrà di seguito analizzata la famiglia *telos* e in particolare la piattaforma hardware Tmote Sky messa a disposizione dall'Università degli Studi di Padova.

5.1.1 Famiglia di nodi Telos

I nodi della famiglia *telos* hanno un'architettura molto simile tra loro. Sono stati progettati dall'Università di Berkeley (California) e prodotti dalla Moteiv Corporation. MSP430 della Texas Instruments è il microcontrollore montato che ha un'architettura a 16 bit di parallelismo. È dotato di 10kB di memoria RAM, 48kB di memoria flash, di un convertitore ADC a 12-bit e di un controllore DMA (Direct Memory Access). Il ricevitore è il CC2420 della Chipcon. I nodi della famiglia *telos* sono pertanto conformi alle specifiche dello standard IEEE 802.15.4 [23] e compatibili con lo stack radio Zigbee [24]. Il nodo presenta un'antenna a 2.4GHz integrata sullo stampato che permette di trasmettere sino a 125m in campo aperto e circa 50m negli edifici. Ogni nodo è inoltre dotato di 1Mbyte di memoria flash esterna e di alcuni sensori (umidità, temperatura, intensità luminosa). Ciascun nodo è identificato tramite un indirizzo programmabile a 16 bit che può essere facilmente definito durante l'operazione di installazione del software. A tale proposito il nodo è dotato di un convertitore seriale-USB (visibile in figura 5.1) che facilita le operazioni di installazione data l'enorme diffusione di questo standard. Non si necessita di alcuna "programming board" per la programmazione del nodo (come avveniva per i motes[1] della famiglia "Mica"), l'unica operazione necessaria per accedere al modulo consiste nel collegarlo alla presa USB di un computer. Inoltre sono presenti nel nodo alcuni leds per facilitarne la programmazione e il debug da work-station. Vediamo in figura 5.2 le due componenti necessarie per lavorare con i nodi della famiglia "Mica".

5.2 Piattaforma Software

5.2.1 TinyOS-2.x

TinyOS-2.x (TOS2) è la naturale evoluzione di TinyOS-1.x (TOS1) il più noto sistema operativo per reti di sensori wireless. Il nome deriva dall'abbreviazione di Tiny Operating System, infatti TinyOS è un sistema operativo open source sviluppato per sistemi "embedded" dall'Università californiana di Berkeley

[1] letteralmente il termine "mote" significa granello, in questo contesto è sinonimo di nodo sensore.

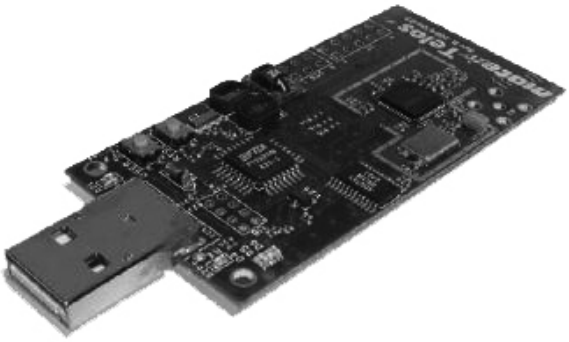

Figura 5.1. Nodo Tmote Sky della famiglia Telos.

Figura 5.2. Nodo Mica 2 e a fianco la "programming board" necessaria.

in cooperazione con Intel Research. Alla data di stesura di questo documento l'ultima versione disponibile del sistema operativo è la 2.0.2 che non è retro-compatibile con le versioni 1.x. Questo principalmente perché vi è stata una completa riscrittura del sistema operativo sia sotto il profilo organizzativo che di ottimizzazione delle risorse. Le applicazioni TinyOS sono scritte in *nesC*, un dialetto del più noto *linguaggio di programmazione C*, ottimizzato per la ridotta quantità di risorse disponibili per un sensore wireless.

Le astrazioni hardware in TOS2 seguono una gerarchia di tre livelli chiamati HAA (Hardware Abstraction Architecture) [25]:

- HPL (Hardware Presentation Layer): è un livello di astrazione collocato immediatamente sopra la piattaforma hardware che permette di avere il pieno controllo dell'hardware sottostante come I/O pins e registri di sistema. Questo livello non astrae nessuna delle funzionalità della piattaforma, ma ne maschera solo il codice di controllo ed è dipendente dall'hardware in uso nella piattaforma ("hardware-dependent").

- HAL (Hardware Adaptation Layer): questo livello si colloca sopra all'HPL e fornisce un livello più astratto delle funzionalità dell'hardware sottostante fornendo più facilità di utilizzo ed efficienza. HAL fornisce ancora la completa funzionalità del sottostante hardware e ne è ancora dipendente ("hardware-dependent").

- HIL (Hardware Interface Layer): è collocato sopra all'HAL; con questo livello viene fornito un controllo dell'hardware sottostante indipendente dalla piattaforma anche se non fornisce tutte le funzionalità possibili in HAL. Le interfacce tra componenti sono standardizzate in modo da rendere il sistema modulare e compilabile per un numero differente di piattaforme. A questo livello si rinuncia ad ogni possibilità di ottimizzazione del codice.

Le principali funzionalità fornite da TOS2 sono di seguito descritte:

1. Scheduler
   Lo Scheduler implementa una politica non-preemptive[2] FIFO (*first-in first-out*, il primo arrivato è il primo ad uscire) dove ogni task ha il suo riservato spazio nella coda ed ogni task può essere accodato solo una volta. L'accodamento fallisce se e solo se il task era già stato accodato in precedenza. Per riaccodare uno stesso task più volte bisogna richiamare l'accodamento all'interno dell'esecuzione del task stesso utilizzando una variabile ausiliaria che ne memorizzi lo stato. Essendo lo scheduler stesso un componente, esso può essere sostituito da un'altra versione dello stesso; questo permette di poter implementare nuove politiche di scheduling, ad esempio in [27] si fornisce un *Priority Based Scheduler* che gestisce 5 livelli di priorità dei tasks. Tale implementazione permette uno scheduling dei tasks con preemption; in questo modo è possibile interrompere tasks non critici in favore dell'esecuzione immediata di tasks con priorità più alta. Ciò rende possibile e semplifica lo sviluppo di softwares che soddisfano precisi vincoli temporali come le applicazioni real-time.

2. Virtualization
   In TOS2 si è introdotto per molti compondenti il concetto di *virtualizzazione* della risorsa. Questo permette di creare un'istanza di un qualsiasi oggetto che fornisce l'interfaccia richiesta. Ad esempio si possono creare tante istanze di componenti Timers, senza preoccuparsi di fornire alcun collegamento con la risorsa che fornisce tale servizio (Clock locale) in modo da ridurre possibili errori e semplificare la gestione delle risorse. Vediamo qui sotto come risulta semplice la creazione dell'istanza di un componente Timer.

   ```
   components App, new TimerMilliC();
   App.Timer -> TimerMilliC;
   ```

[2] il processo in esecuzione non può essere interrotto forzatamente dall'esterno, ma bisogna attenderne la sospensione spontanea.

3. Sensors
   Sono fornite le interfacce standard che permettono l'utilizzo dei sensori di un nodo indipendentemente dalla piattaforma hardware. Tramite semplici comandi come Read, ReadStream e Get è possibile ricevere informazioni dal sensore interessato.

4. Power Management
   Tutte le risorse del nodo (compreso il Microcontrollore e il chip Radio) forniscono interfacce per gestirne lo stato (acceso/spento). In particolare TOS2 distingue due classi di *power-management*: microcontrollori e periferiche. I microcontrollori hanno diversi stati di consumo di energia, mentre le periferiche hanno solo due stati che sono acceso o spento.

Cosa è TinyOS

È il più noto sistema operativo per WSN, open source e specifico per sistemi embedded sviluppato dall'Università californiana di Berkeley e dal gruppo Intel. Tale sistema operativo fornisce un ambiente di sviluppo di applicazioni per WSN e comprende i driver per le periferiche e i middleware di comunicazione e di sensing. Con TinyOS è compresa la libreria contenente le principali componenti necessarie per lo sviluppo di un'applicazione. Nei casi più semplici per sviluppare un'applicazione sarà necessario solamente collegare i componenti già forniti nella libreria.

Cosa non è TinyOS

Le caratteristiche non offerte dal sistema operativo devono esser necessariamente conosciute per riuscire a sviluppare un'applicazione efficiente, perciò sono di seguito elencate:

- non c'è la gestione della memoria MMU (Memory Management Unit).
- qualsiasi processo sarà pertanto in grado di interagire direttamente con gli altri senza nessun tipo di modalità protetta. Ciò è dovuto al fatto che su di un nodo sensore ci sarà un'unica applicazione in esecuzione.

- non vi è il concetto di concorrenza tra processi ma tutto è perfettamente sequenziale.

- non implementa una astrazione totale delle periferiche hardware. Questo perché il loro controllo diretto da parte dell'applicazione permette efficienza ed ottimizzabilità del codice.

- non vi è il concetto di allocazione dinamica della memoria.

5.2.2 Il linguaggio di programmazione *nesC*

NesC [26] è un linguaggio di programmazione a basso livello derivato dal *C* e utilizzato per sviluppare TOS stesso. Fu appositamente disegnato per ottimizzare i diversi vincoli che un'applicazione per *reti di sensori wireless* richiede. Per mantenere il sistema ad un livello semplice, *nesC* fu disegnato come un linguaggio statico, non c'è allocazione dinamica della memoria. I concetti base inclusi che caratterizzano *nesC* sono l'esecuzione del codice guidata dagli eventi, un modello flessibile di concorrenza e una strutturazione orientata alle componenti.

Struttura a componenti

Il modello di programmazione di TinyOS si basa su una particolare architettura a componenti. Ciascun componente costituisce una parte indipendente del codice finale. Ogni componente è definito da due parti, una prima parte che specifica le interfacce fornite ed utilizzate (attraverso le parole speciali *uses* e *provides*) e la seconda parte rappresenta la sua implementazione interna.

Le interfacce sono delle strutture bidirezionali utilizzate dai componenti per comunicare tra loro. Ogni interfaccia specifica un insieme di funzioni divisibili in due categorie *commands* ed *events*.

- *commands* sono funzioni che devono essere implementate dal componente che fornisce tale interfaccia.

- *events* sono funzioni che devono essere implementate dal componente che utilizza l'interfaccia.

Quindi un componente che implementa un'iterfaccia deve fornire anche un insieme di funzioni da lui implementate (*commands*) e necessita che il componente che utilizza tale interfaccia implementi delle funzioni (*events*) che vengono richiamate a seguito di alcuni eventi. Tale strutturazione è fissa per ogni componente e ne evidenzia la relazione con le funzionalità delle componenti fisiche del nodo sensore; ogni componente infatti ha delle funzionalità e può generare eventi che dovranno esser gestiti.

Le parole chiavi per eseguire un comando sono:

```
call nome_interfaccia.nome_comando();
```

dove nome_interfaccia è il nome dell'interfaccia utilizzata e nome_comando indica il comando da richiamare. L'esecuzione del comando è immediata e il controllo ritorna quando il codice implementato dal comando è stato eseguito. Per richiamare un evento si usa la seguente sintassi:

```
signal nome_interfaccia.nome_evento();
```

Come per i comandi l'esecuzione della routine associata all'evento è immediata e la chiamata ritorna quando termina il codice relativo all'evento. Esistono due tipi di componenti in *nesC*.

- moduli che implementano le funzionalità di una o più interfacce.
- configurazioni sono utilizzate per assemblare una o più componenti assieme, connettendo le interfacce fornite da un componente con quelle utilizzate da un altro. Ogni applicazione in *nesC* è caratterizzata dall'avere sempre una configurazione che funge da nodo radice della struttura del programma.

Il modello di concorrenza

Il codice di un'applicazione *nesC* viene eseguito in maniera sincrona se questo appartiene ad un *task* e in maniera asincrona se è codice appartenente a routine di gestione degli interrupt (eventi hardware esterni). Il task è un meccanismo per eseguire delle operazioni in un instante di tempo successivo alla sua chiamata. Un task deve essere definito come una funzione che ritorna un elemento *void*:

```
task void mio_task(){... }
```

I task vengono richiamati attraverso la parola chiave *post*, es. `post mio_task();`. L'operazione *post* programma l'esecuzione del task inserendolo in una coda FIFO e poi ritorna immediatamente. Ogni task verrà eseguito dallo scheduler tenendo conto della politica FIFO della coda e non potrà essere interrorro da un nessun altro task. I tasks possono essere considerati atomici tra loro, ma non sono atomici rispetto alle routine di gestione degli interrupt. I componenti possono usare i tasks quando le tempistiche d'esecuzione non devono essere immediate.

La divisione del codice, in *nesC*, segue due classificazioni:

- Codice Sincrono, che è il codice inteso come funzioni, comandi, eventi o task che è raggiungibile solo da task.
- Codice Asincrono, che è codice raggiungibile da almeno un gestore di interrupt (interrupt handler).

Il codice sincrono viene eseguito dallo scheduler con politica FIFO fino a completamento. Quando invece il sistema segnala un interrupt, il codice relativo viene immediatamente eseguito sospendendo qualsiasi codice sincrono che era in esecuzione in quell'istante. Per evitare che alcuni pezzi di codice vengano eseguiti parzialmente si può ricorrere all'uso del blocco *atomic*{... }. Viene infatti assicurata l'esecuzione atomica delle operazioni contenute nel blocco. Il possibile insorgere di condizioni di *race*[3] viene rilevato e segnalato durante la compilazione del programma.

Operazioni split-phase

Siccome i tasks vengono eseguiti sino a compimento, e non possono sottrarsi il processore, TinyOS non possiede operazioni che permettono di rimanere bloccati in attesa di una risorsa. Tutte le operazioni con una lunga latenza sono ottimizzate con la tecnica *split-phase* che si basa sulla divisione tra comando di richiesta e comando di completamento di tale richiesta. I comandi (*commands*) di un'interfaccia sono solitamente richieste di esecuzione di un'operazione, se il comando

[3]Condizione che si verifica quando due o più processi tentano di usare la stessa risorsa portandola ad uno stato inconsistente.

è *split-phase* esso ritorna subito nel programma chiamante e il completamento della richiesta di tale comando verrà segnalato in un secondo momento con un evento.
Ad esempio l'invio di un messaggio viene effettuato attraverso la chiamata
`call SendMsg.send(uint16_t address, uint8_t length, message_t* msg)`
che si preoccuperà di prendere in carico l'invio del messaggio puntato da `msg`.
Quando il messaggio sarà effettivamente spedito dal nodo, verrà segnalato
`event void SendMsg.sendDone(message_t* msg, error_t error){... };`
che ci confermerà l'effetivo inoltro nel canale di comunicazione del messaggio puntato dall'argomento `msg`. Ciascun componente implementa una metà dell'operazione *split-phase* e richiama l'altra. Le operazioni che non fanno parte della categoria *split-phase* semplicemente non hanno alcun evento di completamento della richiesta, come può essere il caso dell'accensione di un Led del nodo.

Capitolo 6

Il Software

La parte di software che implementa ATS è costituita da un unico componente in *nesC*, ma è stato necessario costruire un ambiente che permettesse sia l'esecuzione del protocollo ATS sia la raccolta di informazioni di funzionamento necessarie per avere un controllo e una verifica del funzionamento dell'algoritmo. Il software totale è costituito da un pacchetto di cinque programmi correlati. I dispositivi

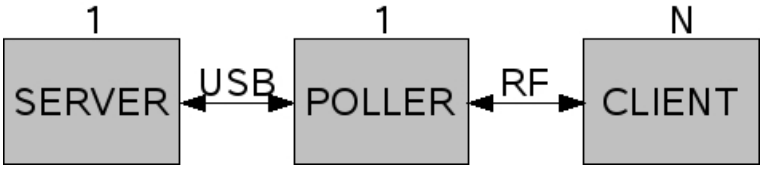

Figura 6.1. Componenti con cardinalità che compongono il software utilizzato. Si evidenzia inoltre il tipo di comunicazione tra le componenti, la prima seriale attraverso il connettore USB mentre la seconda via Radio Frequenze.

utilizzati sono organizzati secondo la gerarchia di figura 6.1 e di seguito descritta:

- Un nodo chiamato Poller (spesso definito anche con il nome di Base-Station) che eseguirà il codice in *nesC* che avrà il compito di generare via radio richieste periodiche dei parametri necessari al controllo dell'andamneto del protocollo di sincronizzazione in esecuzione nei vari Clients. Inoltre dovrà ricevere le risposte fornite da ciascun Client e convogliarle verso un Server che ne permetterà il salvataggio e l'elaborazione.

- Un numero N di nodi detti Clients in cui verrà installato il protocollo di sincronizzazione. Il protocollo ATS è stato sviluppato in *nesC* come componente indipendente che fornisce attraverso un'opportuna interfaccia il servizio di sincronizzazione. Il protocollo è utilizzato da un software di gestione del nodo che permette: (1) la ricezione delle richieste generate dal nodo Poller, (2) il recupero delle informazioni di sincronizzazione attraverso l'utilizzo dell'interfaccia fornita dal modulo di sincronizzazione, (3) la trasmissione al nodo Poller di tali informazioni raccolte.

- Un computer con funzionalità di Server che attraverso un'applicazione *Java* ha il compito di acquisire via USB tutte le informazione che il nodo Poller collezionerà in seguito alle richieste fatte ai Clients.

Il codice in *nesC* scritto per l'applicazione è destinato a nodi di tipo *telos* anche se con alcune leggere modifiche si può adattare alle altre famiglie di nodi. Sarà di seguito descritto come è stato strutturato.

6.1 Formato dei pacchetti

Tutte le componenti della struttura descritta precedentemente comunicano tra di loro attraverso uno scambio di pacchetti. Ogni componente deve definire all'interno del proprio codice la struttura dei pacchetti che può gestire. Questo essenzialmente per due motivi: per riconoscere pacchetti in entrata come messaggi propri e per accedere attraverso la definizione ai campi del pacchetto stesso. Sono state implementate quattro strutture di pacchetto: TimeSyncStart, PollReqMsg, PollRespMsg, SyncMsg. TinyOS fornisce *Active Message* (AM) layer per il multiplexing[1] dell'accesso alla radio attraverso un campo "AM Type" nel pacchetto. Il funzionamento è simile al campo "type" del frame in Ethernet o all'utilizzo delle porte in UDP e TCP che svolgono la funzione di multiplexing per servizi di comunicazione.

[1] In italiano multiplazione si riferisce alla condivisione tra più utenti di una risorsa di sistema.

Viene data la possibilità di utilizzare interfacce parametriche, come riportato di seguito, dove il parametro tra parentesi è il valore del campi "AM Type" relativo ad un certo pacchetto.

```
components new SerialAMSenderC(AM_POLLRESP);
components new SerialAMSenderC(AM_SYNCMSG);
```

Questo permette di scrivere un codice più semplice e leggibile in quanto si evita di utilizzare un componente unico per la ricezione/trasmissione di diversi messaggi.

I pacchetti "AM" hanno un ulteriore campo "AM address" che permette di indirizzare il pacchetto ad un nodo particolare. Il valore "AM address" di ciascun nodo può essere impostato durante l'installazione dell'applicazione TinyOS sul nodo, e può anche essere cambiato a runtime, cioè quando l'applicazione è in esecuzione sul nodo. La definizione di un indirizzo, come sarà in seguito spiegato, sarà utile per forzare la topologia della rete nei nostri tests.

TinyOS2 ha introdotto per i messaggi una nuova astrazione chiamata `message_t` che definisce il buffer standard di un messaggio. La struttura di `message_t` è definita come segue:

```
typedef nx_struct message_t {
  nx_uint8_t header[sizeof(message_header_t)];
  nx_uint8_t data[TOSH_DATA_LENGTH];
  nx_uint8_t footer[sizeof(message_header_t)];
  nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

Figura 6.2. Struttura standard del buffer contenente il generico pacchetto da trasmettere.

dove i campi `header, footer, metadata` non sono accessibili direttamente, ma attraverso le opportune interfacce. I pacchetti utilizzati tra Server/Poller e Poller/Client sono inseriti nel campo `data` che è composto da `TOS_DATA_LENGTH` bytes. La grandezza di default è 28 bytes, può essere modificata ridefinendo la costante `TOS_DATA_LENGTH` durante la fase di compilazione. Ad esempio per aumentare la capacità del campo `data` a 40 bytes basterà inserire la riga `CFLAGS += -DTOSH_DATA_LENGTH=40` nel Makefile.

TimeSyncStart

Questo pacchetto viene spedito dall'applicativo *Java* al nodo *poller* e permette la gestione della rete di *clients* dalla postazione dell'operatore. Il campo `flag` è utilizzato per comandare il nodo secondo le seguenti operazioni:

- inizia ad inviare le richieste periodiche ai nodi *clients* con intervallo `poller_rate`.
- sospendi richieste ai nodi *clients*.
- inizializza la sincronizzazione nei nodi *clients* con periodo `sync_rate`.
- blocca la sincronizzazione tra i *clients*.
- reset totale dello stato del nodo *client*

I campi `poller_rate, sync_rate`, al di là del nome, possono essere utilizzati come campi ausiliari del `flag`.

```
typedef nx_struct TimeSyncStart{
 nx_uint8_t flag;
 nx_uint8_t poller_rate;
 nx_uint8_t sync_rate;
} TimeSyncStart;
```

Figura 6.3. Struttura pacchetto *TimeSyncStart*.

PollReqMsg

Allo scadere del periodo di polling il nodo *poller* incrementa il valore del campo `pollNum` ed invia in modalità broadcast questo pacchetto. Ogni nodo *client* che lo riceve invia indietro al nodo *poller* un pacchetto denominato PollRespMsg contenente i propri parametri che descrivono lo stato del protocollo ATS e un ulteriore campo che fa riferimento all'identificativo della richiesta ricevuta `pollNum`. Gli ulteriori campi `flag, field` sono utilizzati per impartire comandi ai singoli nodi *clients* come reset di sistema, o inizio/fine sincronizzazione.

```
typedef nx_struct PollReqMsg{
 nx_uint16_t pollNum;
 nx_uint8_t flag;
 nx_uint8_t field;
} PollReqMsg;
```

Figura 6.4. Struttura pacchetto *PollReqMsg*.

PollRespMsg

Questo è il pacchetto di risposta che il *client* confeziona e spedisce quando ha ricevuto una richiesta "PollReqMsg" dal *poller*. Nel pacchetto sono contenute un insieme di informazioni per identificare il nodo (nel campo `nodeId`), l'identificativo del ciclo di polling (nel campo `pollNum`) e l'insieme dei parametri all'istante di ricezione di "PollReqMsg" che definiscono lo stato del protocollo ATS (definiti nel capitolo 4) :

- `globalTime` è la stima del tempo globale del nodo.
- `tau` è il timestamp in accordo con il clock locale all'istante di ricezione di "PollReqMsg".
- `tau_star` è l'istante di tempo in accordo con il clock locale relativo all'ultima ricezione di un "SyncMsg".
- `offset` stima dell'offset del clock virtuale.
- `eta[n]` parametro utilizzato per la stima dello skew. n è il numero massimo di nodi considerati, di solito 4.
- `skew` stima dello skew del clock virtuale.

```
typedef nx_struct PollRespMsg{
 nx_uint16_t nodeId;
 nx_uint16_t pollNum;
 nx_uint32_t globalTime;
 nx_uint32_t tau;
 nx_uint32_t tau_star;
 nnx_int32_t offset;
 nx_uint32_t eta[n];
 nx_uint32_t skew;
} PollRespMsg;
```

Figura 6.5. Struttura pacchetto *PollRespMsg*.

SyncMsg

Il pacchetto di sincronizzazione scambiato tra i nodi *clients*. Ogni nodo ne spedisce uno allo scadere del periodo di sincronizzazione. Anche se il periodo dovrebbe esser lo stesso per tutti i nodi, l'invio di tale messaggio non deve essere sincrono, anzi per evitare possibili collisioni di messaggi è opportuno che la scadenza di tale periodo avvenga in instanti diversi per ciascun nodo. Nel pacchetto vengono inseriti:

- `nodeId` identificativo del nodo.
- `seqNum` numero sequenziale che viene incrementato ogni volta venga inviato un "SyncMsg".
- `timestamp` effettivo istante (16 bit) di invio del messaggio.
- `tau` tempo locale (32 bit) quando il messaggio è pronto per essere spedito.
- `tau_star` istante di tempo in accordo con il clock locale relativo all'ultima ricezione di un "SyncMsg".
- `alpha_hat` stima dello skew del clock virtuale.
- `o_hat` stima dell'offset del clock virtuale.
- `isSync` valore booleano che indica se il nodo è sincronizzato o meno.

```
typedef nx_struct SyncMsg{
 nx_uint16_t nodeId;
 nx_uint16_t seqNum;
 nx_uint16_t timestamp;
 nx_uint32_t tau;
 nx_uint32_t tau_star;
 nx_uint32_t alpha_hat;
 nx_int32_t o_hat;
 nx_bool isSync;
} SyncMsg;
```

Figura 6.6. Struttura pacchetto *SyncMsg*.

6.2 Codice nodo *Poller*

Il codice del nodo *poller* è scritto in *nesC*, quindi sviluppato secondo una struttura a componenti. Il componente di configurazione dove si definiscono le connessioni con le interfacce definite da TOS2 è chiamato `PollerAppC` mentre il modulo dove è definito il programma di gestione del nodo è `PollerC`. I componenti principali dichiarati in `PollerAppC` sono:

- `MainC` è il componente di controllo principale, necessario in ogni programma TOS. Esso permette di interagire con la sequenza di Boot di TinyOS.
- `LedsC` fornisce l'interfeccia di controllo dei leds.
- `TimerMilliC` permette di creare delle istanze di timer per la gestione del nodo, per temporizzare il ciclo di polling e l'accensione dei vari leds.
- `ActiveMessageC` componente che controlla la radio del nodo.
- `SerialActiveMessageC` componente che controlla la comunicazione seriale su USB.
- `PollerC` componente che gestisce la comunicazione con i nodi *clients*.

6.2.1 Compiti del nodo *Poller*

Il compito principale dell'applicazione è quello di fungere da fulcro tra i nodi che eseguono la sincronizzazione e il Server che ne registra i dati. Di seguito sono elencale le operazioni eseguite dal nodo Poller.

Richiesta raccolta dati

Il modulo `PollerC` prevede l'utilizzo di un Timer periodico per la gestione delle richieste di raccolta dati da inoltrare ai nodi *clients*. Il Timer è definito da un periodo di tempo Δt allo scadere del quale segnala l'evento "Timer.fired()" che invia una richiesta di raccolta dati alla rete di nodi con un pacchetto *PollReqMsg* (come nel primo punto di figura 6.7). In seguito viene incrementato il valore del campo `pollNum` come identificativo della sessione di polling. Il Timer è

attivabile/disattivabile dall'interfaccia *Java* del nostro computer attraverso dei semplici pulsanti.

Gestione del protocollo

Attraverso l'assegnazioni di opportuni valori al campo `flag` contenuto nel pacchetto *PollReqMsg* in figura 6.4 è possibile gestire il protocollo di sincronizzazione nei nodi *clients*. Ad esempio se all'inizio di una sessione di test si desidera che tutti i nodi della rete partano contemporaneamente ad eseguire l'algoritmo di sincronizzazione, sarà dunque possibile forzare una reset dello stato del protocollo ATS.

Raccolta dati ed inoltro

In seguito alla ricezione di un pacchetto *PollReqMsg* ogni nodo *client* attende un certo periodo di tempo e in seguito invia al nodo *poller* un pacchetto *PollRespMsg* visibile in figura 6.5 e descrivente lo stato del protocollo ATS. Il nodo *poller* è in grado di ricevere anche i pacchetti di sincronizzazione *SyncMsg* che i *clients* si scambiano. Tutti i pacchetti ricevuti dal nodo *poller* vengono in seguito inoltrati al Server attraverso la porta USB del nodo. Per il numero considerevole di pacchetti da registrare ed inoltrare, il nodo *poller* implementa una coda dove vengono registrati i messaggi ricevuti dal chip radio. È stato implementato un apposito task chiamato `uartSendTask` che ha il compito di svuotare la coda spedendo tali messaggi attraverso la porta USB.

6.3 Codice nodo *Client*

Il nodo *client* è programmato diversamente dal nodo *poller*. Il software del nodo *client* è costituito da un componente di configurazione chiamato `ClientAppC` mentre il modulo dove è definito il programma di gestione del nodo è `ClientC`. Oltre alle componenti standard, come `MainC,Leds,TimerMilliC,ActiveMessageC`, sono dichiarate anche le componenti `CC2420TransmitC`, che permette di gestire gli eventi corrispondenti agli istanti di ricezione/trasmissione dei messaggi via

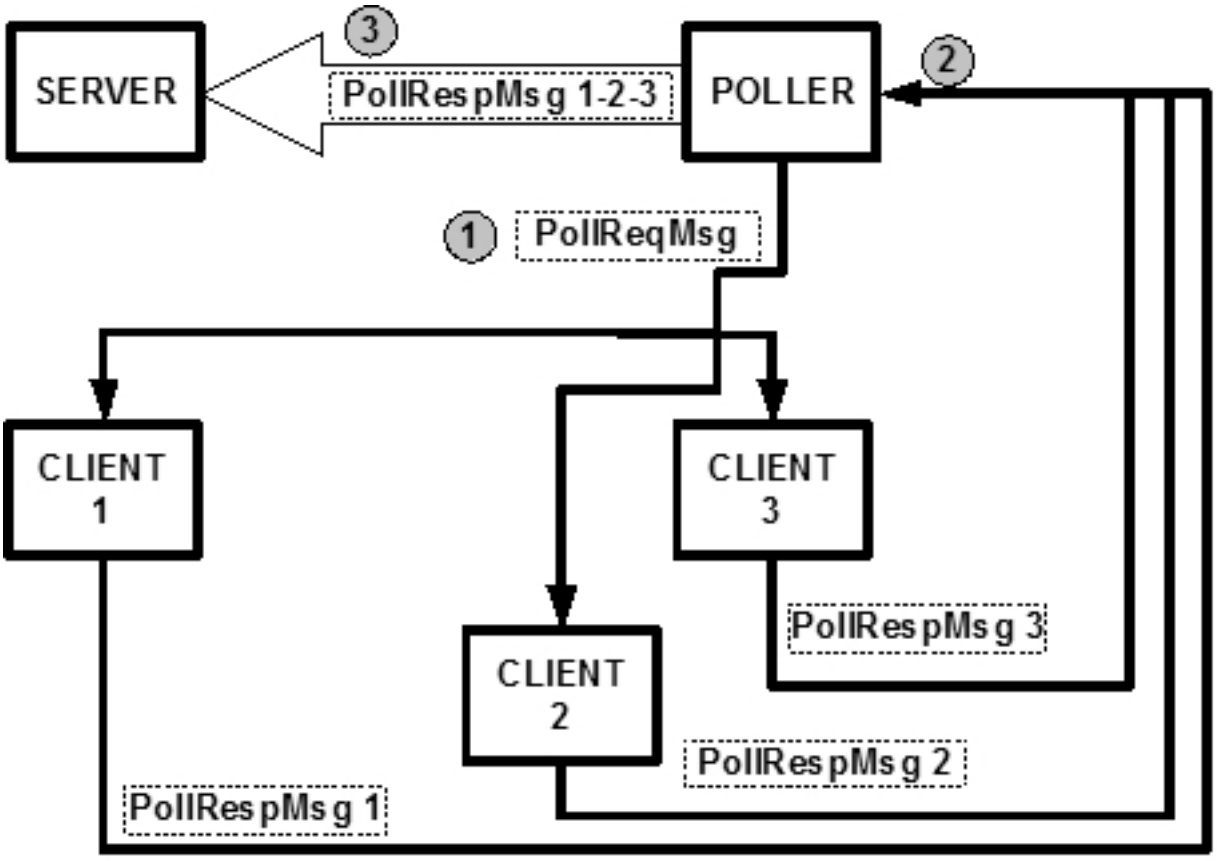

Figura 6.7. Principio di funzionamento della raccolta dati. (1) il nodo *poller* invia in modalità broadcast una richiesta di raccolta dati (pacchetto "PollReqMsg"). (2) ciascun *client* che riceve tale richiesta risponde al *poller* con un "PolRespMsg" contenente i dati da raccogliere. (3) I dati raccolti nel poller vengono reinoltrati al server da un opportuno task che svuota la coda dei messaggi ricevuti.

radio, `ClientC`, che è la componente di gestione del nodo e `AverageTimeSyncC` che costituisce il cuore del protocollo di sincronizzazione ATS.

6.3.1 Compiti del nodo *Client*

Il nodo *client* ha il compito di inizializzare la componente `AverageTimeSyncC` che si occupa della sincronizzazione e gestire la comunicazione con essa attraverso le apposite interfacce fornite. Elenchiamo di seguito le principali funzionalità fornite dal nodo *client*.

Gestione delle richieste provenienti dal nodo *poller*

In caso di arrivo di un pacchetto "PollReqMsg" ogni nodo *client* esegue una serie di operazioni. Per prima cosa il nodo calcola l'esatto istante di tempo d'arrivo del messaggio in accordo con il clock locale, poi controlla il campo `flag` del messaggio ricevuto ed in base al valore può decidere se sospendere la sincronizzazione, riattivarla, resettarla o non compiere nessuna delle precedenti azioni. In seguito il nodo inizia la costruzione del pacchetto "PollRespMsg" compilandone i campi in base ai valori ritornati dalle interfacce fornite dalla componente che si occupa della sicronizzazione.

Componente di Sincronizzazione

Tutto il lavoro relativo alla sincronizzazione è assolto dalla componente indipendente `AverageTimeSyncC`, che si occupa dello scambio dei pacchetti di sincronizzazione tra nodi *clients*, della loro elaborazione, e di fornire una insieme di interfacce per il servizio di sincronizzazione.

Il funzionamento dell componente di sincronizzazione è legato agli eventi di ricezione e trasmissione di un pacchetto di sincronizzazione. Ogni nodo fa affidamento ad un timer periodico interno che allo scadere di un intervallo temporale Δt invia un pacchetto di sincronizzazione ("SyncMsg" di figura 6.6) ai propri vicini come visualizzato in figura 6.8.

Alla ricezione di un pacchetto di sincronizzazione viene eseguito un filtraggio basato sull'identificativo "AM" del nodo in modo da poter forzare la rete di nodi ad una specifica topologia, anche se tutti i nodi della rete possono comunicare

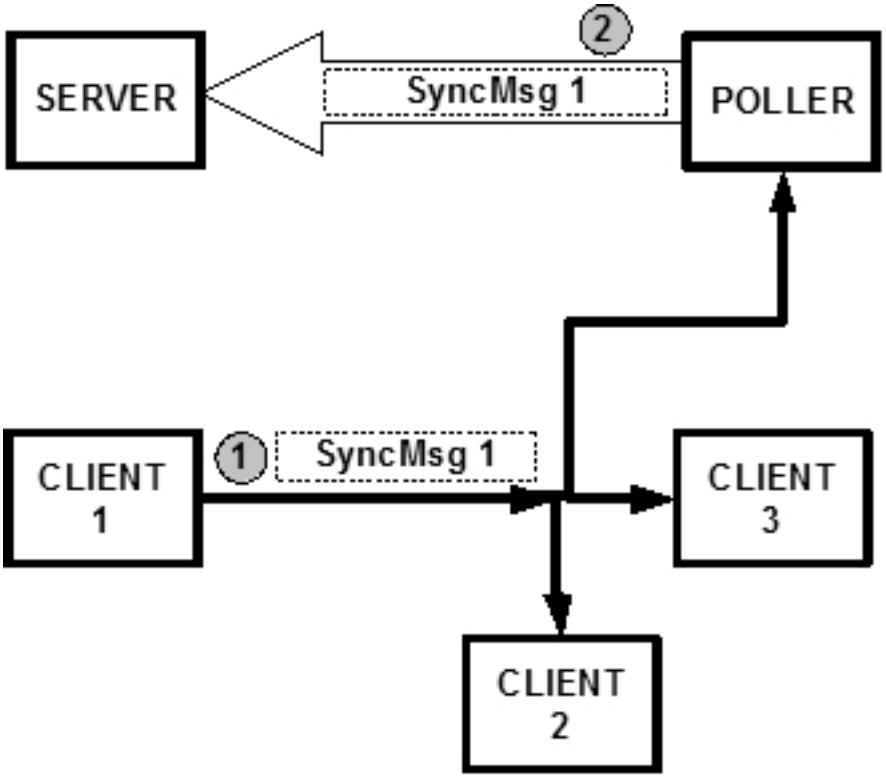

Figura 6.8. Principio d'inoltro dei messaggi di sincronizzazione. (1) il nodo *client* allo scadere del timer che regola l'invio dei messaggi di sincronizzazione invia in modalità broadcast il proprio pacchetto "SyncMsg", che viene ricevuto dagli altri nodi vicini e dal nodo *poller*. (2) in un secondo momento il nodo *poller* inoltra il "SyncMsg" al server per via seriale.

direttamente. In sequenza dopo il primo filtro ne è stato inserito un secondo che permette di stabilire se il timestamp di ricezione del messaggio sia corretto. Durante la fase di implementazione è infatti emerso che saltuariamente il timestamp dei messaggi ricevuti non risulta corretto; infatti quando ci sono molti pacchetti nella coda FIFO di ricezione può accadere che gli accoppiamenti fatti con i timestamp generati dall'evento SFD siano incorretti.

Il metodo utilizzato per controllare se il timestamp ottenuto è corretto è:

- quando viene segnalato l'evento `receivedSFD(uint16_t time)` viene salvato su una variabile `SFDtime` il valore dell'argomento `time`.
- più tardi, quando si ottiene il pacchetto, viene confrontato `SFDtime` con il timestamp del pacchetto `msg->time`.

Se i due valori non coincidono è probabile che qualcosa sia "andata storta" ed è preferibile scartare il pacchetto.

Se i filtri permettono la ricezione di tale pacchetto il nodo prosegue con il salvataggio all'interno di una tabella degli istanti di spedizione e ricezione del messaggio e l'aggiornamento dei parametri di stato di ATS quali : $\eta_{ij}, \hat{\alpha}_i, \hat{o}_i^*$ e τ_i^*.

Le interfacce fornite da `AverageTimeSyncC` sono in tutto cinque.

Init è un'interfaccia standard usata per tutti i componenti di TOS, e permette di inizializzare il componente attraverso un comando `command error_t init()`. L'inizializzazione è effettuata prima che il componente sia funzionante e prevede di impostare variabili e condizioni necessarie ad un corretto funzionamento del componente. Nel componente in discussione l'inizializzazione prevede semplicemente di impostare il valore di tutte le variabili di sistema a 0.

StdControl è l'interfaccia standard utilizzata per accendere o spegnere il componente che la fornisce. Tali operazioni sono possibili attraverso le chiamate ai comandi `command error_t start()` e `command error_t stop()`.

GlobalTime è tratta dall'implementazione di FTSP del 2002-2003, Vanderbilt University definita come segue:

```
interface GlobalTime
{
 async command uint32_t getLocalTime();
 async command uint32_t getGlobalTime(uint32_t time);
}
```

Dove `getLocalTime()` ritorna il tempo locale del nodo,
invece `getGlobalTime(uint32_t time)` restituisce la stima del tempo globale convertendo il tempo locale fornito come parametro `time`.

TimeSyncInfo è un'interfaccia di controllo che fornisce tutti i valori delle variabili di funzionamento dell'algoritmo ATS.

```
interface TimeSyncInfo
{
 async command bool getStatus();
 async command int32_t getOffset();
 async command float getEta(uint8_t idx );
 async command float getSkew();
 async command uint32_t getGTime(uint32_t);
 async command uint32_t getLTime();
 async command uint8_t getSeqNum();
 async command uint8_t getNumEntries();
 async command uint32_t getTauStar();
}
```

Descriviamo brevemente il significato dei comandi:

- `getStatus()` restituisce un valore booleano in accordo con lo stato del nodo (sincronizzato o meno).
- `getOffset()` restituisce la stima dell'offset del nodo.
- `getEta(uint8_t idx )` restituisce il valore del parametro η_{ij} dove l'indice `idx` identifica il nodo j relativo.
- `getSkew()` restituisce la stima dello skew con il clock virtuale.

- `getGTime(uint32_t)` restituisce la stima del tempo globale partendo dal tempo locale fornito come parametro.
- `getLTime()` restituisce il tempo locale del nodo.
- `getSeqNum()` restituisce il progressivo identificativo della sessione di sincronizzazione del nodo.
- `getNumEntries()` restituisce il numero di nodi con cui il nodo si sta sincronizzando (solitamente si va da un minimo di 0 nodi ad un massimo di 4).
- `getTauStar()` restituisce il valore di τ^* al momento della chiamata.

Syncer è l'interfaccia che permette la gestione della sincronizzazione.

```
interface Syncer
{
command bool start();
command bool stop();
command bool isRunning();
command error_t reset();
command error_t restore();
 }
```

- `start()` riavvia il timer responsabile dell'invio dei pacchetti di sincronizzazione e permette la ricezione dei messaggi di sincronizzazione.
- `stop()` ferma il timer responsabile dell'invio dei pacchetti di sincronizzazione e fa scartare tutti i messaggi di sincronizzazione futuri.
- `isRunning()` restituisce lo stato del timer responsabile dell'invio dei pacchetti di sincronizzazione. Se attivo restituisce *vero*, altrimenti restituisce *falso*.
- `reset()` riavvia il timer responsabile dell'invio dei pacchetti di sincronizzazione, inoltre resetta le variabili di stato del protocollo.

6.4 Codice *Server*

Si tratta di codice *Java* eseguito dal computer collegato via USB al nodo *poller*. TinyOs fornisce una serie di applicazioni per l'elaborazione dei dati ricevuti da un nodo attraverso comunicazione seriale. Il più semplice di essi è "net.tinyos.tools.Listen" che costituisce uno sniffer di pacchetti di basso livello. Esso stampa a video il contenuto binario dei pacchetti che cattura mettendosi in ascolto di una porta specifica. Vediamo di seguito un esempio dell'output generato dalla classe Listen.

```
00 FF FF 00 00 04 22 06 00 02 00 01
00 FF FF 00 00 04 22 06 00 02 00 02
00 FF FF 00 00 04 22 06 00 02 00 03
00 FF FF 00 00 04 22 06 00 02 00 04
00 FF FF 00 00 04 22 06 00 02 00 05
00 FF FF 00 00 04 22 06 00 02 00 06
00 FF FF 00 00 04 22 06 00 02 00 07
00 FF FF 00 00 04 22 06 00 02 00 08
00 FF FF 00 00 04 22 06 00 02 00 09
00 FF FF 00 00 04 22 06 00 02 00 0A
00 FF FF 00 00 04 22 06 00 02 00 0B
```

Anche se interessante per una prima forma di debugging, la classe Listen non sembra fare al caso nostro, non potendo fare alcuna analisi dei parametri contenuti generati da ATS.

Per una gestione dei singoli campi di un pacchetto si è utilizzata l'utilità "MIG" (Message Interface Generator) che può costruire un'intefaccia *Java, Python* o *C* della struttura del messaggio definita nel codice *nesC*. Data una sequenza di bytes l'interfaccia generata da mig è in grado di risalire a ciascun campo del pacchetto. Si sono utilizzate le classi *Java* generate a partire dalla struttura dei paccheti definita in *nesC*. Con TOS viene, inoltre, fornita un'applicazione *Java* chiamata oscilliscope che permette la raccolta di alcune letture fatte dai sensori wireless; una modifica di tale applicazione è stata utilizzata e in seguito adattata alle necessità della gestione dei dati raccolti dal nodo *poller*. Il software si occupa di reperire i dati forniti sulla porta USB e di riassemblarli secondo la struttura

originaria. Per ogni nodo della rete viene creato un file di testo dove ogni record corrisponde ad un pacchetto PollRespMsg. Ogni campo del pacchetto è separato da uno carattere separatore. Oltre al file di testo viene salvato anche un file con estensione *mat* che permette di elaborare i dati di tutti i nodi direttamente con MATLAB. L'applicazione *Java* fornisce una GUI[2] dotata di una serie di comandi, per la gestione del comportamento dei nodi, e un grafico dell'andamento dei parametri principali dell'algoritmo ATS che ne permette una prima valutazione del funzionamento.

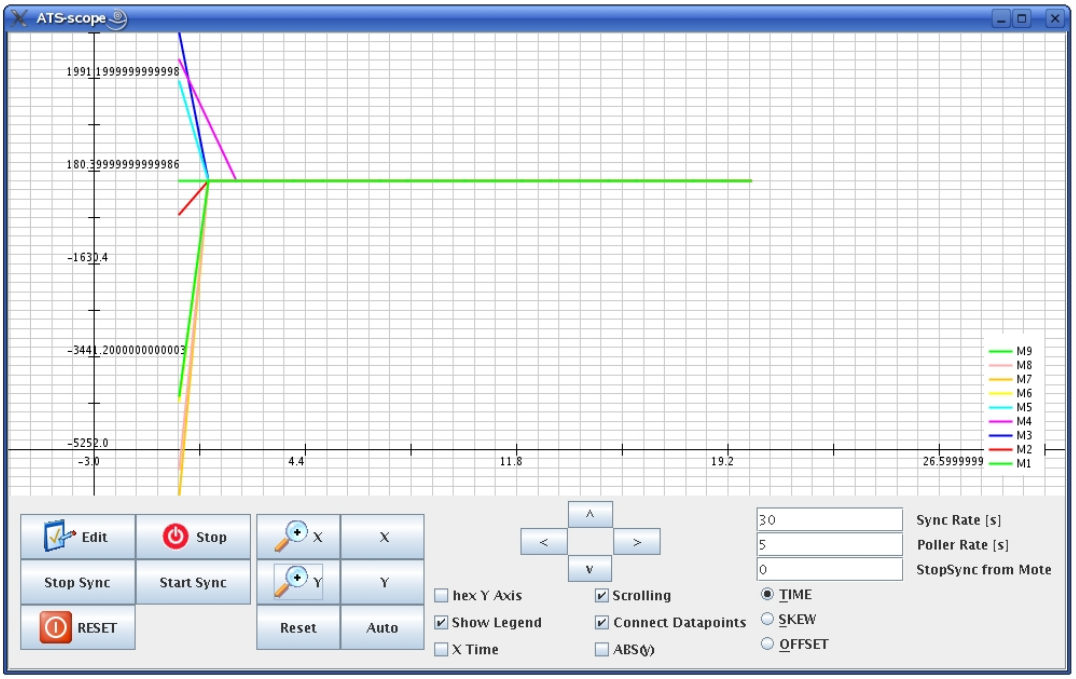

Figura 6.9. Interfaccia GUI utilizzata per il controllo della rete di nodi.

[2]GUI è l'acronimo di Graphical User Interface, cioè interfaccia grafica per l'utente

Capitolo 7

Esperimenti effettuati e valutazioni

La parte sperimentale di questa tesi riguarda lo studio dell'architettura hardware e software in dotazione per poter implementare in maniera corretta ed efficace l'algoritmo di sincronizzazione Average TimeSync precedentemente esposto.

Tutti i tests effettuati sono stati eseguiti su nodi di tipo Tmote Sky di Moteiv equipaggiati con il microcontrollore a 16 bit Texas Instrument MSP430.

Come sorgente di clock è stato sfruttato l'oscillatore al quarzo esterno, in dotazione al nodo, che ha una frequenza di 32 Khz e ha la caratteristica di funzionare anche quando il microcontrollore viene spento.

Il sistema operativo TinyOS2 fornisce le apposite routine per sfruttare il timestamp a livello MAC per gli eventi di ricezione ed invio di messaggi. Il sistema possiede un meccanismo che è in grado di notificare l'evento SFD (Start Frame Delimiter) e registrarne il tempo locale relativo. Tale evento corrisponde alla trasmissione/ricezione del primo bit del pacchetto in uscita/entrata.

Ciò significa che per ogni messaggio M ricevuto da un nodo, se ne può rilevare il tempo locale in cui il primo bit del messaggio viene ricevuto. Questo valore di tempo viene detto appunto timestamp e viene memorizzato automaticamente in un campo a 16 bit del messaggio arrivato.

Per quanto riguarda i messaggi trasmessi da un nodo, si sfrutta la possibilità di far eseguire un pezzo di codice quando si verifica l'evento SFD. Nel nostro caso il pezzo di codice andrà a modificare un campo del messaggio che si sta per

trasmettere, e in particolare si andrà ad inserire nel messaggio in trasmissione il valore di tempo a 16 bit corrispondete alla generazione dell'evento SFD. Abbiamo così a disposizione i timestamp di messaggi in entrata e uscita, ma non basta visto che la sorgente di clock utilizzata è a 32 Khz e manderebbe in overflow il contatore a 16 bit circa ogni 2 secondi.

7.1 Gestione dei Timestamps

Per far fronte alla ridotta risoluzione del nostro timestamp a 16 bit che chiameremo $t16$ si è adottato l'utilizzo di un riferimento temporale a 32 bit fornito dalla routine di sistema.

TinyOS2 permette di rilevare il valore del timer locale tramite una semplice chiamata al comando dell'apposita interfaccia:

```
call LocalTime.get();
```

che restituisce un valore a 32 bit che chiameremo $t32$.

Questa chiamata può essere effettuata solamente quando verrà notificata la ricezione di un messaggio e non all'arrivo del primo bit SFD. In figura 7.1 è visualizzato l'ordine di notifica degli eventi che seguono la ricezione di un messaggio. Sarà dunque necessario sfruttare entrambi i riferimenti di tempo ($t16$ e $t32$) per ottenere un timestamp valido a 32 bit.

Infatti, all'istante relativo l'evento (2) di figura 7.1 avremo disponibile il timestamp a 16 bit $t16$, mentre alla segnalazione dell'evento (3) che notifica la completa ricezione del messaggio possiamo registrare $t32$ con l'apposita routine.

L'istante corrispondente la ricezione di un messaggio sarà dato dalla formula

$$t32 - (uint16)(t32 - t16)$$

Per quanto riguarda il rilevamento dell'istante esatto di invio di un messaggio useremo la stessa tecnica, quindi per ogni messaggio inviato ci sarà un campo a 32 bit e uno a 16 bit. Nel primo andrà inserito il tempo locale a 32 bit rilevato durante la routine di invio del messaggio. Mentre nel secondo, alla generazione

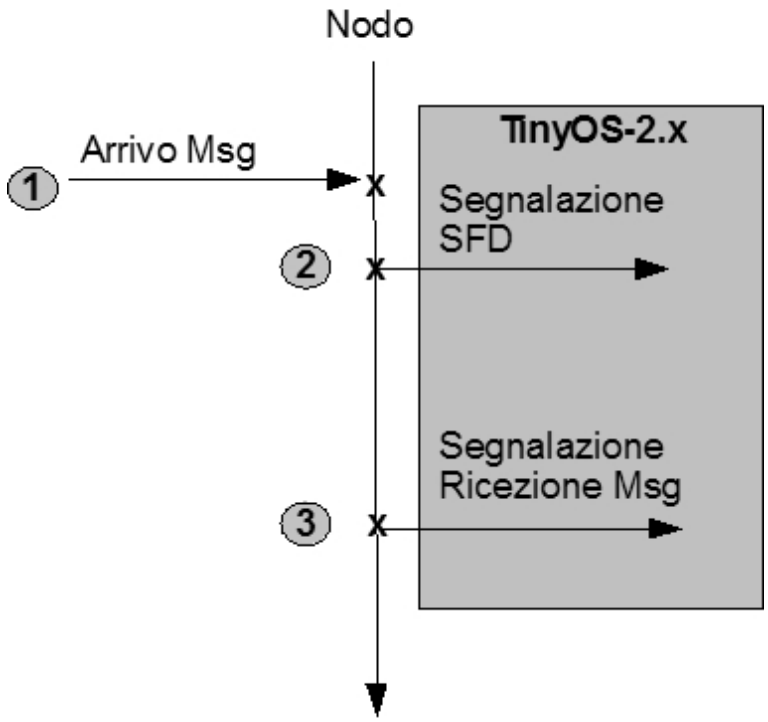

Figura 7.1. (1) Ricezione di un messaggio, segue (2) segnalazione evento SFD ricevuto e (3) segnalazione messaggio arrivato.

dell'evento SFD verrà inserito l'istante a 16 bit in cui sarà appunto generato l'evento SFD stesso.

Un primo esperimento per verificare il funzionamento del sistema di gestione dei vari timestamps è stato condotto su un numero ristretto di nodi. È stato programmato un nodo chiamato Base Station per spedire pacchetti beacon ad intervalli regolari di 5 secondi. Altri 8 nodi sono stati programmati in modo da rilevare il tempo di arrivo a 32 bit del messaggio di beacon secondo le regole esposte sopra e rispondere alla Base Station con un messaggio contente il valore del tempo calcolato.

Il test è stato condotto per un tempo prolungato di circa 4 ore e mezzo in modo da rilevare le variazioni a cui possono essere soggetti gli oscillatori nel tempo. L'andamento dei clock dei nodi è visibile in figura 7.2. Dopo una fase di assestamento iniziale si nota che lo skew risulta piuttosto costante sul lungo periodo. Ciò è molto importante in quanto ci permette di effettuare una stima della frequenza del clock di un nodo sul breve periodo. La figura 7.3 mostra l'andamento iniziale dei clock.

7.2 Average Time Sync

Descrizione dell'esperimento

È stato testato l'algoritmo su una griglia composta da 35 nodi disposti secondo una matrice di 5 righe e 7 colonne come mostrato in figura 7.4 .

Ogni nodo della griglia era all'interno del campo radio di qualsiasi altro, per questo l'algoritmo è stato modificato in modo che ogni nodo consideri i messaggi provenienti dai nodi adiacenti (orizzontali e verticali) nella struttura matriciale, e invece scarti tutti gli altri.

Ad esempio il nodo 1 poteva comunicare solamente con i nodi 2 e 8, il nodo 2 comunicava solo con i nodi 1, 3, 9 e così via. Ogni nodo trasmetteva pacchetti di sincronizzazione con una frequenza di 30 secondi in modo indipendente e asincrono dagli altri.

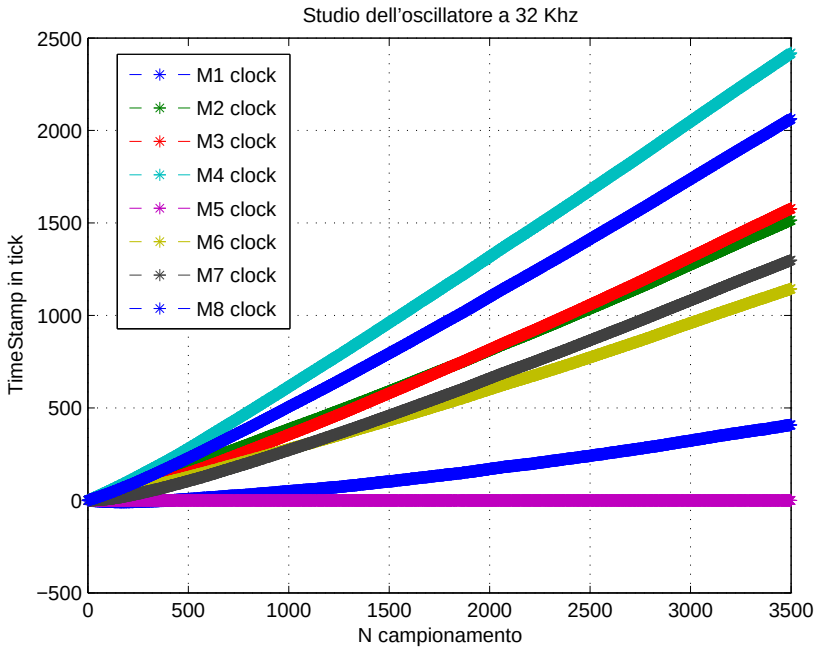

Figura 7.2. Andamento dei clock a 32 Khz riferiti al clock del mote M1

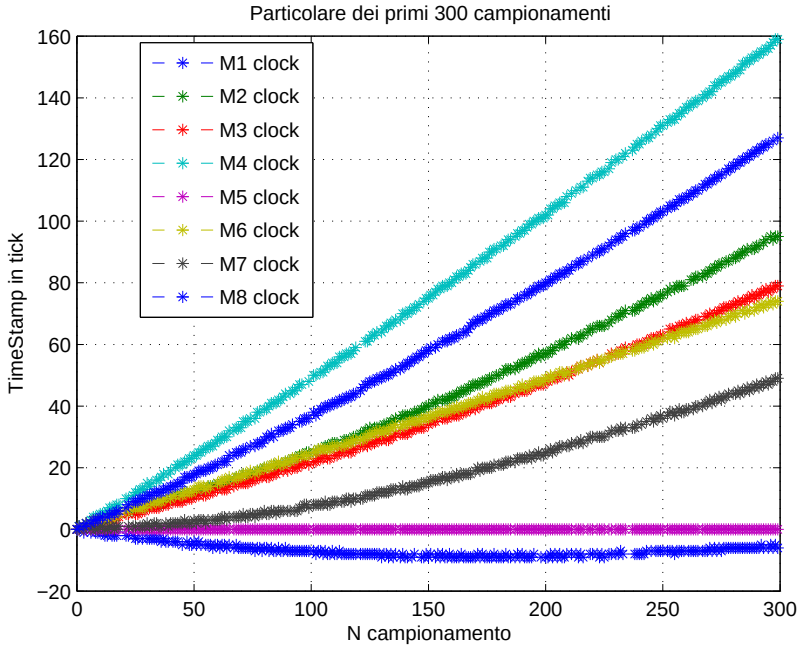

Figura 7.3. Andamento dei clock a 32 Khz nei primi 300 campionamenti

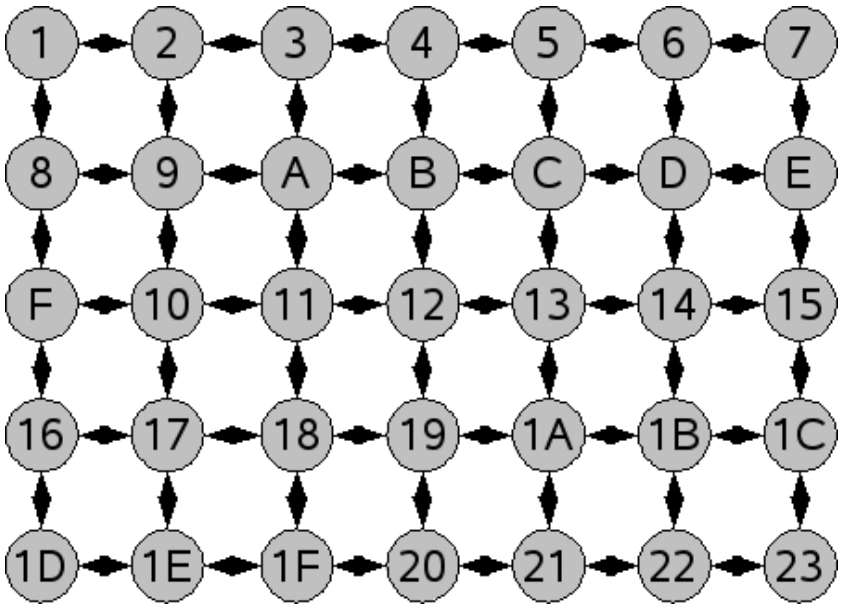

Figura 7.4. Rete composta da 35 nodi

L'esperimento è durato all'incirca 4 ore con il seguente scenario che ricalca quello proposto dagli autori di FTSP [13]:

A: al tempo 00:00 tutti i nodi sono stati accesi.

B: tra 01:00 e 01:30 15 sono stati riavviati il 40% nodi presi a caso, con una distanza di 2 minuti tra un riavvio e l'altro.

C: 01:45 tutti i nodi da M17 a M23 sono stati resi inerti (non trasmettono e ricevono messaggi di di sincronizzazione).

D: 02:15 ai nodi da M17 a M23 è stata riattivata la sincronizzazione.

Vediamo in figura 7.5 l'andamento del tempo globale stimato dei nodi rispetto ad un nodo di riferimento. Notiamo che dopo una prima fase di assestamento il valore del tempo globale stimato dai singoli nodi converge per poi mantenersi sempre compreso nei limiti di [+10,-10] ticks dal nodo di riferimento (linea di asterischi in rosso).

Ammontano al 5% del totale delle interrogazioni fatte dalla Base Station le volte in cui il messaggio di risposta non è stato ricevuto da quest'ultima. Ciò

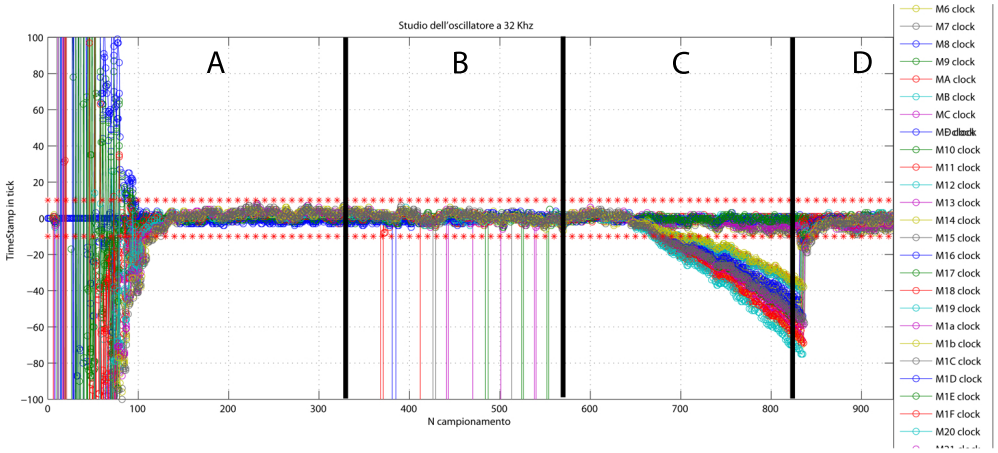

Figura 7.5. ATS con 35 nodi.

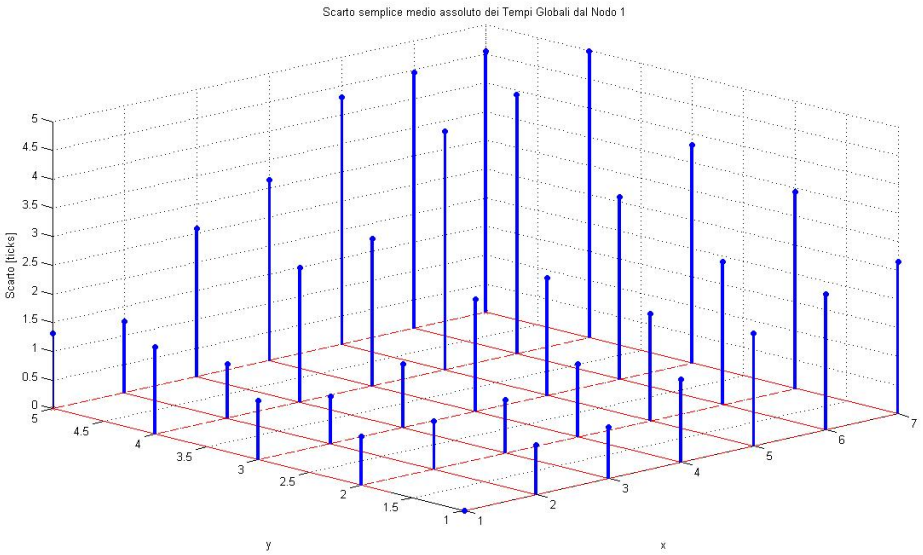

Figura 7.6. ATS con 35 nodi. Le linee blu rappresentano i nodi nella rete 5x7 nello spazio. L'estensione delle barre blu indica lo scarto assoluto medio dal nodo di riferimento M1 in posizione (1,1). Le linee rosse rappresentano i collegamenti tra i nodi.

è dovuto principalmente alle collisioni dei messaggi nel canale radio, infatti la condizione che ogni nodo è all'interno del campo radio di qualsiasi altro influenza la disponibilità del canale.

Ciascun nodo ha trasmesso alla base station oltre alla propria stima del tempo globale della rete, anche tutti i valori necessari al funzionamento dell'algoritmo. Questo per poterne effettuare una verifica del giusto funzionamento e per agevolarne eventuali analisi.

Nella figura 7.6 viene riportato l'andamento dello scarto medio assoluto della stima del tempo globale di un nodo di riferimento (in questo caso il nodo M1) rispetto i restanti nodi. Tali dati sono stati ricavati dall'elaborazione delle informazioni riguardanti il periodo di regime dell'algoritmo nel test precedentemente descritto. Il grafico evidenzia come l'algoritmo effettui una sincronizzazione locale per ottenere la sincronizzazione globale. In particolare si nota come l'errore medio sia limitato (solo 1 tick) nei nodi vicini al nodo di riferimento M1 per poi andare ad aumentare con i nodi a maggior distanza da M1 (4.5 ticks per i nodi a 10 hop).

Tale condizione risulta favorevole nel caso di una rete di sensori in quanto l'accadere di un evento si verifica in un determinato punto dello spazio, riguardando solo un piccolo sottoinsieme di nodi. I nodi vicini che hanno rilevato l'evento avranno bisogno di una buona stima del tempo globale, mentre i nodi distanti che non hanno percepito l'evento non necessitano di tale precisione. Quindi ogni evento è caratterizzato da una certa località, e avere un algoritmo di sincronizzazione con buona precisione locale permette di avere una buona stima dei valori temporali legali all'evento stesso. Questa caratteristica è favorevole nell'attuazione di TDMA, in quanto il canale di comunicazione è condiviso solo tra nodi vicini.

7.2.1 Prestazioni ATS

Per studiare le prestazioni dell'algoritmo di sincronizzazione al variare del periodo di invio dei messagi di sincronizzazione sono stati condotti una serie di esperimenti su una mesh di 3x3 nodi. Ad ogni test si è lasciato girare l'algoritmo per circa mezz'ora, tra un test e l'altro è stato variato solamente il rate di inol-

tro dei pacchetti di sincronizzazione mentre tutti gli altri parametri sono rimasti immutati.

I valori di rate di sincronizzazione utilizzati sono:

- 7 secondi in figura 7.8;
- 15 secondi in figura 7.10;
- 30 secondi in figura 7.11;
- 1 minuto in figura 7.12;
- 2 minuti in figura 7.13;
- 4 minuti in figura 7.14;

Per ogni test sono riportati due grafici, il primo rappresentante lo scarto dalla media delle stime del tempo globale di ogni nodo, mentre nel secondo grafico è riportato il max pairwise error, cioè il massimo tra le differenze delle stime dei tempi globali di ogni coppia di nodi. Si nota che, per valori del rate di sincronizzazione bassi, la compensazione dell'offset permette un funzionamento ottimo dell'algoritmo malgrado non si sia in grado di effettuare una stima corretta della deriva dei clocks dei nodi adiacenti.

Nel caso del tempo di sincronizzazione di 7 secondi la stima del tempo globale $\hat{\tau}_i(\tau_i) = \hat{\alpha}_i(\tau_i - \tau_i^*) + \hat{o_i^*}$ dove $\tau_i - \tau_i^*$ è la differenza tra il tempo locale τ_i e tempo locale τ_i^* istante di ricezione dell'ultimo pacchetto di sincronizzazione. Nel caso peggiore (un solo nodo adiacente) questa differenza è di 7 secondi, che corrispondono a circa 224000 ticks (7 sec * 32 Khz). Inoltre dai risultati sperimentali $\hat{\alpha}_i$ assume valori compresi tra $1e^{-7}$ ed $1e^{-6}$. Per valori del rate di sincronizzazione bassi si puo' affermare $\hat{\tau}_i(\tau_i) \simeq \hat{o_i^*}$ e quindi l'algoritmo funziona come se ci fosse solo una compensazione dell'offset.

All'aumentare del rate di sincronizzazione la compensazione della deriva comincia a contribuire alla stima del tempo globale. Ciò si nota osservando i grafici relativi al max pairwise error; infatti dopo la fase di assestamento dell'algoritmo (fase in cui si stimano i valori di $\hat{\alpha}_i$) il valore dell'errore rimane sempre inferiore a 4 ticks.

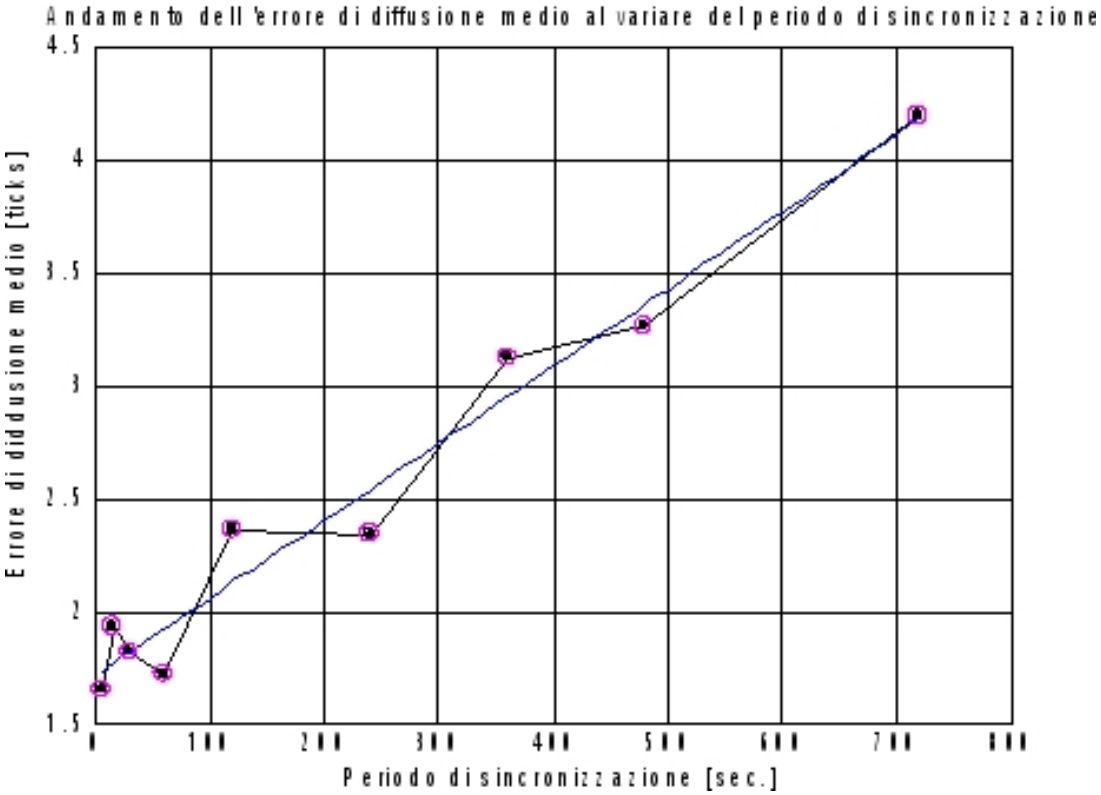

Figura 7.7. Andamento dell'errore di diffusione medio al variare del periodo di sincronizzazione. I punti indicano l'errore mentre la linea blu è l'interpolatrice descritta da Y = 0.0034 X + 1.7106.

SyncRate	7 s	15 s	30 s	60 s	120 s	240 s	360 s	480 s	720 s
Error	1.6407	1.9286	1.8081	1.7115	2.3539	2.3383	3.1204	3.2620	4.1895

Tabella 7.1. Tabella riassuntiva dell'errore di diffusione medio al variare del periodo di sincronizzazione (SyncRate).

I dati raccolti dai test sono stati elaborati per ottenere una visione dell'andamento dell'errore di diffusione medio (media degli errori massimi tra i nodi) al variare del periodo di sincronizzazione utilizzato nell'algoritmo. I valori medi sono calcolati come media dell'errore massimo ottenuto con campionamenti (ogni 10 secondi) del tempo globale di ogni nodo. Il tempo di osservazione è stato di circa 30 minuti per ogni test considerando l'algoritmo a regime. La tabella 7.1 riassume i valori ottenuti mentre in figura 7.7 ne è rappresentato l'andamento dove la retta interpolatrice è calcolata con il metodo dei minimi quadrati ed è descritta dalla relazione Y = 0,0034 X + 1,7106.

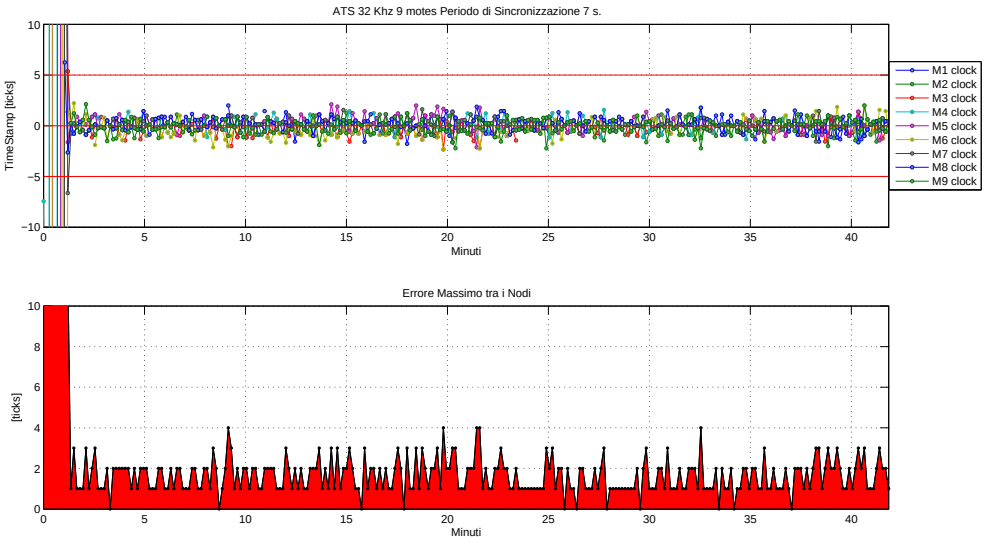

Figura 7.8. ATS con periodo di sincronizzazione di 7 secondi

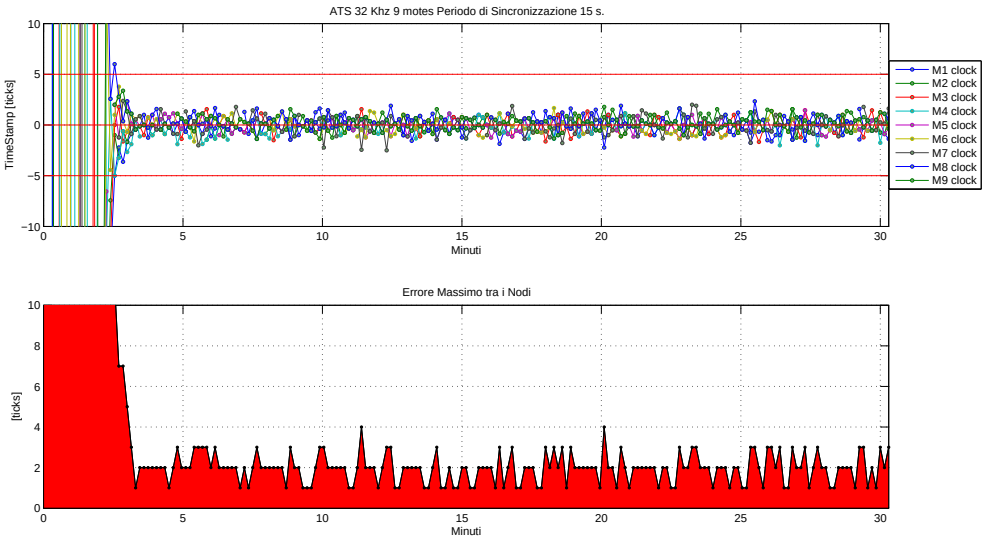

Figura 7.9. ATS con periodo di sincronizzazione di 15 secondi

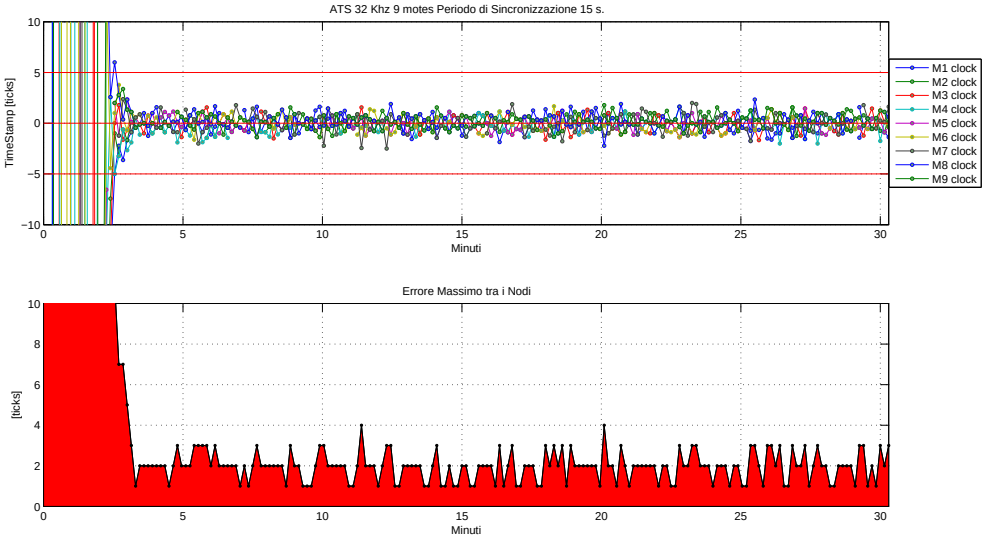

Figura 7.10. ATS con periodo di sincronizzazione di 15 secondi

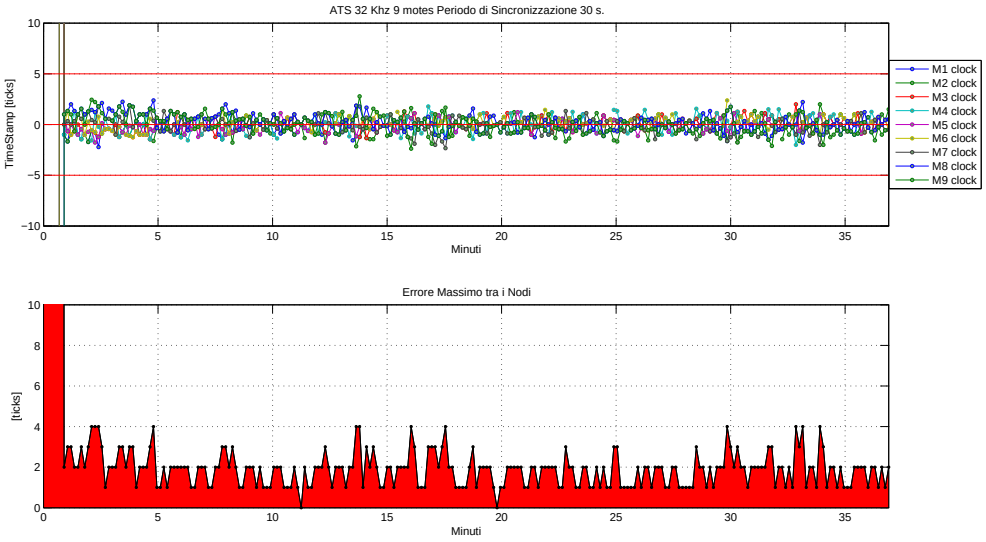

Figura 7.11. ATS con periodo di sincronizzazione di 30 secondi

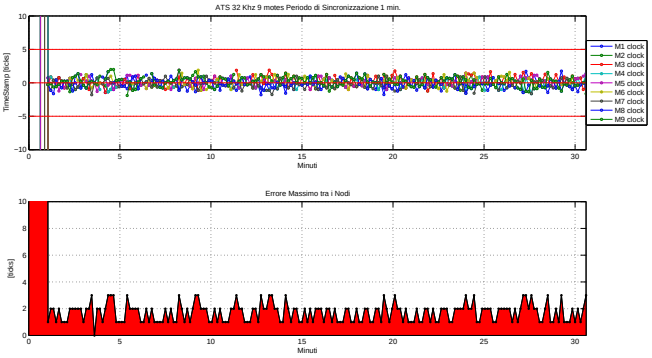

Figura 7.12. ATS con periodo di sincronizzazione di 1 minuto

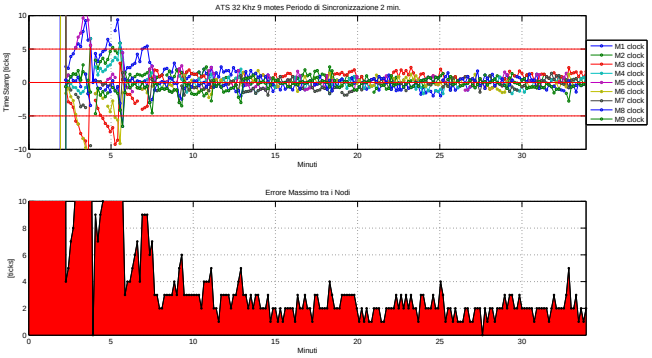

Figura 7.13. ATS con periodo di sincronizzazione di 2 minuti

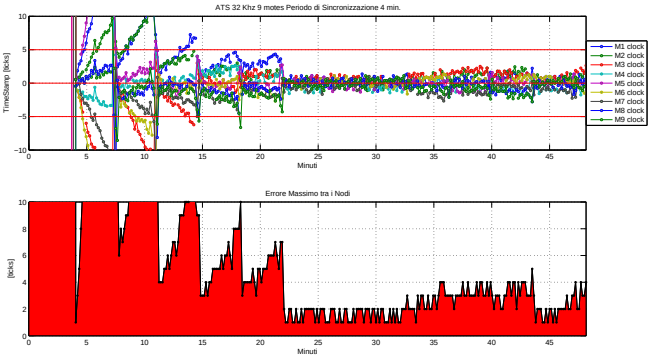

Figura 7.14. ATS con periodo di sincronizzazione di 4 minuti

7.2.2 Confronto ATS e FTSP

Grazie all'implementazione di FTSP per nodi *teslos* fornita dalla "ETH Zuerich University" [28] è stato possibile fare un confronto con ATS basato sulle stesse condizioni, cioè su una stessa piattaforma e con la stessa tipologia di rete. Entrambi gli algoritmi sono stati impostati per funzionare con periodo di sincronizzazione di 60 secondi, ciò significa che ogni 60 secondi ciascun nodo della rete ha inviato un messaggio di sincronizzazione. Il grafico di figura 7.15 mette a confronto l'errore di dispersione di gruppo ottenuto dai due differenti algoritmi. A parità di condizioni iniziali (rate di sincronizzazione, topologia di rete, sorgente temporale) l'errore di dispersione medio ottenuto con ATS (1.7 ticks) risulta mediamente inferiore rispetto ai risultati ottenuti con FTSP (2 ticks). Si nota la presenza di alcuni picchi dell'errore ottenuto con FTSP, questo è probabilmente dovuto alla specifica implementazione dell'algoritmo che non prevede un filtraggio di alcuni possibili timestamps corrotti, causati da qualche imperfezione del sistema operativo. Sono proposti di seguito i grafici dello scarto medio dal nodo

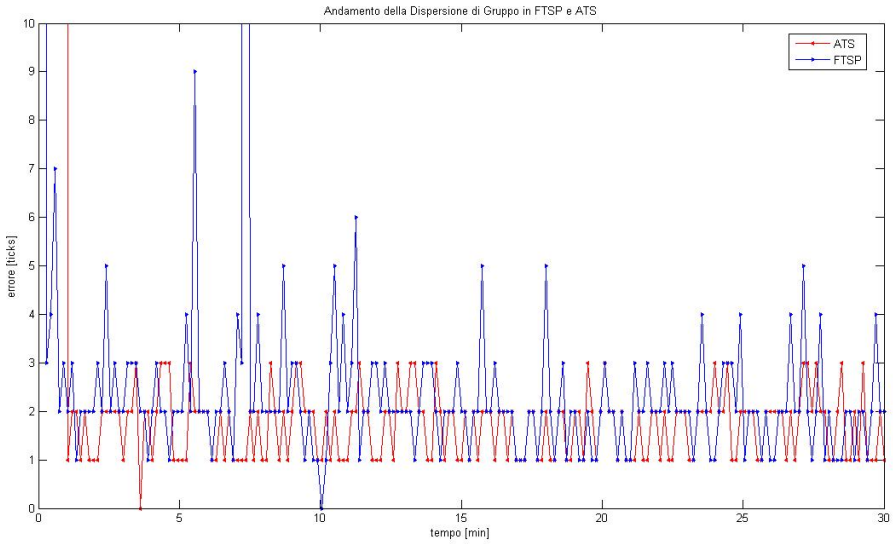

Figura 7.15. Confronto dell'errore di Dispersione di Gruppo tra ATS e FTSP

di riferimento e dell'errore massimo istantaneo relativo ad ATS in figura 7.16 e a FTSP in figura 7.17.

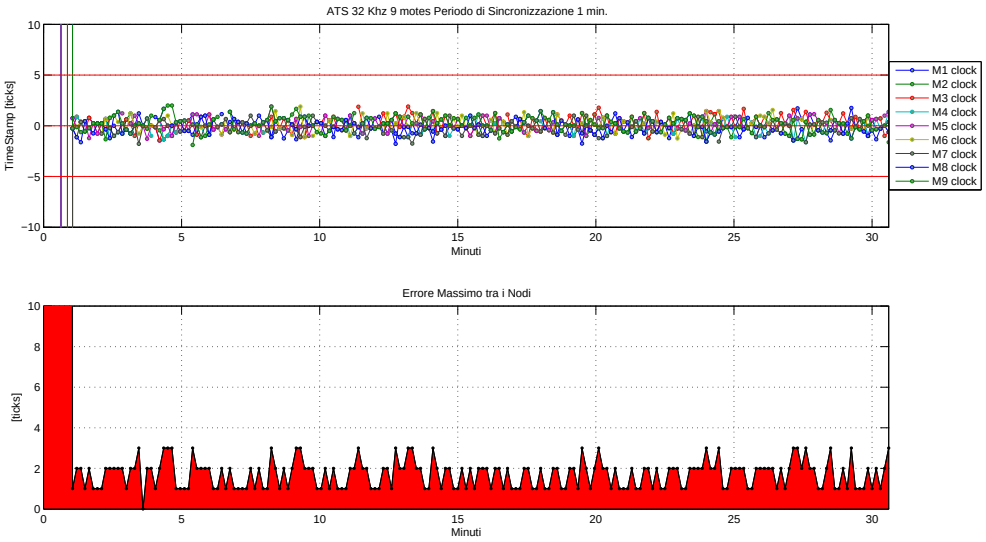

Figura 7.16. ATS con rate di sincronizzazione di 1 minuto

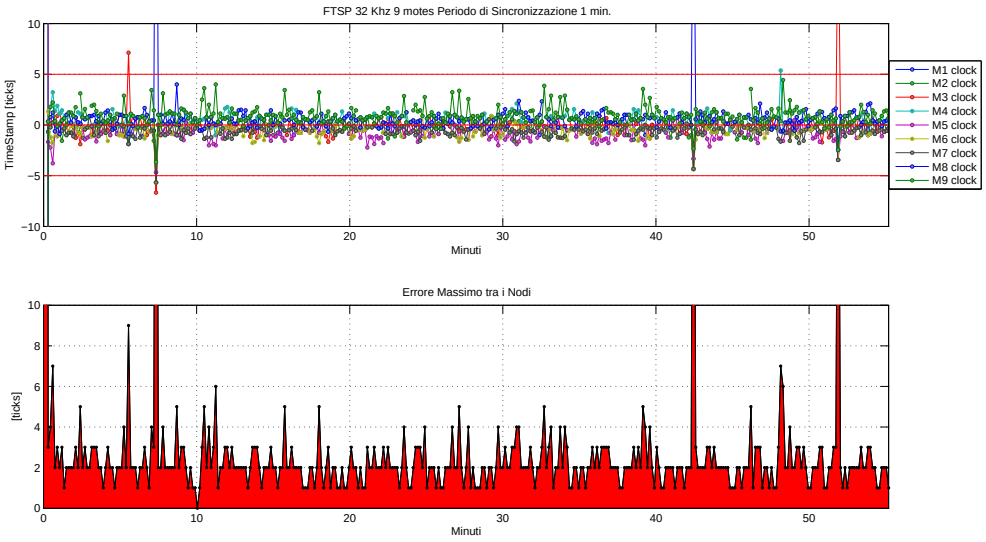

Figura 7.17. FTSP con rate di sincronizzazione di 1 minuto

7.3 Confronto con gli algoritmi precedenti

Presentiamo di seguito i vantaggi e gli svantaggi di Average TimeSync Protocol confrontandolo con i precedenti algoritmi conosciuti. Per ogni soluzione si considera una rete di n nodi in cui l'algoritmo viene eseguito per m sessioni di sincronizzazione. Ciascun nodo è considerato all'interno del raggio di comunicazione degli altri $n-1$ nodi, in modo da semplificare lo scenario studiato. In questo modo si ha che ogni nodo ha $n-1$ potenziali nodi vicini.

La tecnica usata da RBS [9] sfrutta i timestamps solo dalla parte del nodo ricevente per eliminare il tempo di accesso e di invio del messaggio. Questo è il vantaggio principale che si ottiene utilizzando questo algoritmo, infatti si eliminano i ritardi non deterministici causati dal nodo trasmittente. Tuttavia se si utilizza il timestamp di livello MAC, come avviene nel nostro metodo, si ottiene praticamente lo stesso beneficio ottenuto in RBS, anzi si eliminano pure lo jitter dovuto alla gestione degli interrupt e quello di decodifica del messaggio (da onda elettromagnetica a dati binari). Per quanto riguarda la congestione della rete, in una sessione di sincronizzazione il nodo segnalatore invia un pacchetto *beacon* a tutti i nodi che a loro volta inviano un altro messaggio per scambiarsi le informazioni. Per n nodi ed m sessioni di sincronizzazione RBS ha una complessità (intesa come numero di trasmissioni e ricezioni di messaggi per ottenere la sincronizzazione) di mn^2. Infatti per ciascuno degli m pacchetti beacon trasmessi, un nodo scambia informazioni con tutti gli altri $n-1$ nodi ricevitori. Per m sessioni di sincronizzazione vengono trasmessi m messaggi beacon dal nodo segnalatore; ciò implica mn ricezioni dagli n nodi ricevitori. Ogni sessione ciascun nodo ricevitore invia un messaggio agli altri $n-1$ nodi. In tutto risultano $m+mn$ messaggi spediti e $nm + nm(n-1)$ messaggi ricevuti da elaborare. Le informazioni registrate da ciascun nodo sono proporzionali al numero di nodi coinvolti nella sincronizzazione, quindi $O(n)$. L'occupazione del canale radio risulta $m + mn$. All'aumentare del numero di nodi n si rende necessaria l'introduzione di più nodi segnalatori; in merito in [9] si propone una strutturazione della rete di sensori in cluster di nodi, dove ogni cluster è fornito di un nodo segnalatore. All'aumentare di n, la rete necessita di una strutturazione gerarchica dei nodi e di una sincronizzazione a priori dei nodi segnalatori. Tutto ciò si traduce in un aumento della

complessità di funzionamento dell'algoritmo a discapito della scalabilità . È da sottolineare che in caso di guasto di un nodo segnalatore tutti i nodi che facevano riferimento ad esso non potranno più usufruire del servizio di sincronizzazione. L'errore medio nel caso di comunicazione single hop è di 29.1 μs.

L'algoritmo TPSN [11] sfrutta il timestamp al livello MAC per ridurre gli errori non deterministici, sia per nodi mittenti che destinatari di messaggi di sincronizzazione. La sincronizzazione avviene sfruttando una struttura gerarchica generata durante una prima fase dell'algoritmo chiamata *level discovery phase*. In seguito ogni nodo di livello i si sincronizza con un nodo di livello superiore $i-1$ attraverso lo scambio di due messaggi, il primo dal nodo di livello i al nodo di livello $i-1$ mentre il secondo va dal nodo di livello $i-1$ al nodo di livello i.

Per m sessioni di sincronizzazione ci saranno quindi $m4\,(n-1)$ elaborazioni di pacchetti, per un totale di $m2(n-1)$ messaggi che circolano nella rete. Questo considerando i messaggi necessari all'effettiva sincronizzazione e non contando i messaggi necessari alla creazione della struttura gerarchica iniziale che aggraverebbero l'overhead dell'algoritmo. Il costo in termini di memoria occupata nel nodo è trascurabile ($O(1)$) visto che ogni nodo a livello i mantiene in memoria solo l'offset con il nodo del livello $i-1$ con cui ha fatto la sincronizzazione. L'inconveniente di TSPN è che non effettua una stima della deriva dei clock dei nodi, costringendo a inviare pacchetti di sincronizzazione con una frequenza superiore ad un qualsiasi algoritmo che prevede tale stima (es FTSP, ATS). Inoltre l'affidarsi ad una struttura gerarchica ne limita la capacità di sopportare guasti, infatti nel caso in cui un nodo a livello i non ha nessun nodo al livello superiore $i-1$, un nuovo albero di copertura deve essere ricreato incidendo sul numero di messaggi trasmessi dal nodo e sulla complessità dell'algoritmo. L'errore medio nel caso di comunicazione single hop è di 16.9 μs.

Tiny Sync/Mini Sync [10] e LTS [12] sulla base dello stesso scambio di pacchetti di TPSN propongono oltre alla compensazione degli offset anche una stima della deriva dei clocks dei nodi, permettendo una migliore precisione della sincronizzazione a parità di messaggi scambiati e all'aumentare del periodo di inoltro. Nella sincronizzazione di un'intera rete si prevede l'ausilio di una struttura gerarchica

della rete stessa, solitamente la generazione di un albero di copertura tramite l'utilizzo di un algoritmo di *leader election*, presentando gli stessi problemi di TPSN nel caso di guasto di qualche nodo.

FTSP [13] similmente ai precedenti algoritmi la sincronizzazione si basa su un nodo radice con cui ogni altro nodo implementa la compensazione sia di offset che della deriva. In aggiunta implementa meccanismi di adattamento alla variazione della topologia della rete ed eventuale guasto di nodi (compreso il nodo radice). Il funzionamento dell'algoritmo è indipendente dalla topologia della rete, l'unico vincolo è rappresentato dal nodo radice che costituisce il riferimento temporale di tutta la rete. Un altro problema di FTSP è rappresentato dal fatto che l'algoritmo permette ad ogni nodo di eleggersi radice dopo un certo periodo in cui non si ricevono messaggi di sincronizzazione. Questo potrebbe tradursi in continue generazioni di diversi nodi radice nel caso di reti di nodi con connessioni saltuarie. Infatti ogni nodo può raggiungere la sincronizzazione attraverso l'informazione trasmessa dal nodo radice o da qualsiasi altro nodo sincronizzato. Si tratta di uno dei migliori algoritmi in letteratura. In generale per le m sessioni in cui l'algoritmo viene eseguito vi sono $2mn$ elaborazioni (mn trasmissioni e mn elaborazioni necessarie) per un totale di mn messaggi che viaggiano nel canale di trasmissione. I dati salvati nella memoria del nodo sono un numero limitato di *reference-points* (coppia di tempo globale e locale relativi allo stesso istante) e quindi $O(1)$. Dai test fatti su piattaforme Mica2 l'errore medio dell'algoritmo usando due nodi è 1.48 μs mentre nel caso di reti multihop l'accuratezza degrada di 0.5 μs per hop.

La tecnica utilizzata in RFA [15] è molto interessante sotto molti punti di vista. Per ottenere la sincronia un nodo deve solamente essere in grado si osservare un evento proveniente dagli altri nodi, senza necessità di risalire a che nodo ha generato tale evento. I singoli nodi non hanno stati interni al di là del loro tempo interno t. La sincronia è ottenuta senza alcuna dipendenza dalla topologia della rete (nessun nodo radice) e indipendentemente dell'istante di avvio dei singoli nodi. Tuttavia al modello matematico originale sono state apportate una serie di modifiche per adattarlo all'implementazione su reti di sensori. La principale di

tale modifiche consiste nel ritardare, di un tempo casuale limitato, l'emissione del segnale da parte di ciascun nodo. Infatti quando i nodi raggiungono la sincronia dovrebbero emettere il loro segnale tutti allo stesso istante, ma questo risulta essere il caso peggiore in una WSN essendo il canale di comunicazione condiviso, si rischia un numero elevato di collisioni che ritarderebbero l'inoltro dei messaggi. Inoltre gli autori assumono che tutti i nodi condividono uno stesso periodo T allo scadere del quale il nodo deve emettere il segnale. Ciò implica l'assunzione che tutti i clock della rete abbiano stessa velocità e non vi è quindi alcuna compensazione della deriva dei clocks. Questo si traduce nella necessità di un frequenza di sincronizzazione relativamente superiore a qualsiasi algoritmo che stimi tale deriva. Per m sessioni di esecuzione dell'algoritmo ogni nodo invia m messaggi e elabora tutti i messaggi che riceve, nel caso peggiore sono $m(n-1)$. Quindi la complessità è di $2mn$ mentre l'occupazione del canale è mn. Le informazioni memorizzate sono limitate ad una coda che mantiene i messaggi ricevuti nella corrente sessione dell'algoritmo, sono quindi $O(n)$. L'algoritmo è stato testato su piattaforma MicaZ/TOS1 utilizzando 24 nodi che vengono sincronizzati con un'approssimazione di 100 μs utilizzando un clock locale a 7.3 MHz per ogni nodo. Risulta singolare il ricorso ad FTSP per ottenere una base di riferimento temporale globale per registrare il timestamp dei segnali generati dall'algoritmo RFA.

L'algoritmo proposto da Solis et al. (2006) [17] risulta supportare reti multihop e sfrutta messaggi asincroni. Utilizza una tecnica simile a quella di ATS anche se il problema viene formulato in maniera diversa. Gli autori specificano che nessun albero di copertura con nodo radice deve esser costruito. I nodi della rete per la sincronizzazione fanno affidamento ad un nodo di riferimento che viene cambiato periodicamente e non è necessario identificarlo. Tuttavia non è chiarito che importanza abbia il nodo di riferimento all'interno della rete né le modalità di cambiamento di tale nodo, né gli effetti sulla sincronizzazione da tale cambiamento. La frequenza dell'oscillatore utilizzato è 921.6 KHz. L'algoritmo è stato implementatato in nodi di tipo Mica2 ottenendo un'accuratezza media inferiore a 2 μs $\simeq$3 ticks (6 μs nel caso peggiore). La stima degli offset e della deriva relativi è fatta con uno scambio bilaterale di informazioni, perciò, nel caso

	Skew	Complessità	Canale	Memoria	Scalabilità	Topologia
RBS	Si	(mn^2)	$(m+mn)$	$O(n)$	Poca	Si (cluster)
TPSN	No	$(4m\,(n-1))$	$(m+mn)$	$O(1)$	Buona	Si (albero)
TinySync	Si	$(4m\,(n-1))$	$(m+mn)$	$O(1)$	Buona	Si (albero)
FTSP	Si	$(2mn)$	(mn)	$O(1)$	Ottima	Si (nodo radice)
RFA	No	$(2mn)$	$(m+mn)$	$O(n)$	Ottima	No
Solis et al	Si	$(2mn)$	$(mn(n-1))$	$O(n)$	Ottima	No
ATS	Si	$(m(n+k))$	(mn)	$O(k)$	Ottima	No

Tabella 7.2. Tabella riassuntiva delle prestazioni degli algoritmi. m è il numero di sessioni. n il numero dei nodi. $k << n$ è una costante. Skew, indica se l'algoritmo prevede la compensazione della deriva. Complessità, è un parametro che indica il numero di trasmissioni e ricezioni eseguite dalla rete per mantenera la sincronizzazione. Canale, indica il numero di messaggi che viaggiano nel canale. Memoria, indica la quantità di dati da memorizzare. Scalabilità, indica se l'algoritmo scala bene in caso di variazione del numero di nodi. Topologia, indica se l'algoritmo dipende dalla struttura della rete.

peggiore, per m sessioni di sincronizzazione vengono spediti $mn(n-1)$ pacchetti. I messaggi elaborati sono tutti i messaggi ricevuti dai vicini che, nel caso peggiore. sono $mn(n-1)$. I dati salvati nel singolo nodo sono proporzionali al numero di nodi vicini cioè $O(n)$.

L'algoritmo ATS, esposto in questo documento, sfrutta il timestamp a livello MAC e utilizza per ogni nodo la compensazione della deriva e dell'offset dei nodi vicini come in [17],[13],[10]. Durante m sessioni di funzionamento dell'algoritmo, la complessità (trasmissioni e elaborazioni conseguenti) è dell'ordine di $m\,(n+k)$. Dove k è il numero massimo di nodi vicini per cui si registrano *data-points* (per il nodo i, sono le coppie di valori di tempo globale stimato da j e da i). La memoria necessaria all'algoritmo e dell'ordine di $O(k)$ e quindi approssimabile ad $O(1)$, mentre l'occupazione del canale risulta di mn messaggi. Diversamente dai precedenti protocolli, ATS sfrutta una tecnica innovativa utilizzata per la risoluzione

di *problemi di consenso*, dove l'informazione relativa alla sincronizzazione non parte da un nodo radice per arrivare ad un nodo foglia, ma è un'informazione distribuita su tutta la rete. Questo esalta le proprietà di scalabilità e di robustezza ai guasti dell'algoritmo; infatti nessuna nuova operazione è necessaria in caso di guasto o aggiunta di un nodo. ATS risulta particolarmente indicato per WSN costituite da nodi mobili in quanto l'algoritmo risulta indipendente dalla topologia della rete e completamente distribuito e robusto. Dai tests condotti ed esposti nel precedente capitolo, le prestazioni di ATS risultano simili a quelle del protocollo FTSP (a parità di condizioni iniziali), anzi vengono superate nel caso di variazioni di topologia della rete visto che ATS non ha alcun legame con essa.

Conclusioni

In questa tesi si è studiata la sincronizzazione temporale intesa come servizio indispensabile per una rete distribuita e in particolare per una rete di sensori wireless (WSN). Si sono studiate le soluzioni adottate dalle reti di elaboratori (computer) prendendo atto che tali soluzioni non risultano adatte per un utilizzo in reti di sensori wireless. Ciò è dovuto in gran parte ai vincoli stringenti in termini di capacità computazionale, di banda di trasmissione, di memoria e di energia che un'applicazione per WSN deve soddisfare.

Si sono quindi studiate le tecniche specifiche per WSN presenti in letteratura notando la difficoltà di confronto visto che l'unico termine di paragone utilizzato è la precisione di sincronizzazione intesa come divergenza temporale dal tempo di riferimento. Questo parametro purtroppo è fortemente dipendente dalla qualità dell'implementazione dell'algoritmo e dalla piattaforma hardware utilizzata. Infatti, tale precisione è fortemente influenzata dalle caratteristiche della sorgente di tempo utilizzata nel singolo nodo (ci si riferisce alla tipologia e alla frequenza dell'oscillatore utilizzato).

Si è quindi cercato di definire delle metriche in modo da poter valutare e confrontare gli algoritmi di sincronizzazione indipendentemente dalla loro implementazione e dal dispositivo hardware utilizzato. Si è in seguito descritto l'algoritmo *Average TimeSync* che determina i parametri di offset e deriva dei nodi come valore medio tra tutti i nodi della rete. Tali parametri vengono calcolati attraverso il meccanismo di *Average Consensus*. Si è cercato di confrontare *Average TimeSync* con gli algoritmi presenti in letteratura, cosa che ha permesso di evidenziarne l'ottima scalabilità e l'alto grado di robustezza ai guasti rispetto alle altre soluzioni.

Il lavoro è proseguito con lo studio della particolare architettura composta dai nodi Tmote Sky e dal sistema operativo TinyOS-2.x. Gran parte del lavoro di tesi è stato dedicato allo sviluppo in linguaggio *nesC* dell'algoritmo *Average TimeSync* e del software di corredo necessario ai tests. Si è descritto approfonditamente lo schema del progetto, riportando le varie problematiche affrontate e le soluzioni adottate nella fase di sviluppo. Si è descritto nel dettaglio la struttura dell'architettura e il software realizzato per il test dell'algoritmo. Si è verificata sperimentalmente la convergenza dell'algoritmo, inoltre sono stati eseguiti una serie di test per valutarne la bontà. I risultati ottenuti hanno dimostrato che l'algoritmo è in grado di sincronizzare reti di sensori indipendentemente dal numero di componenti. Ovviamente, al crescere del numero dei nodi (e quindi al crescere degli hop) l'errore medio aumenta linearmente. Inoltre si è dimostrato come l'errore locale, inteso come scarto medio assoluto del tempo globale stimato da un nodo rispetto ai nodi vicini, rimanga sempre limitato. Questa risulta essere un'ottima proprietà, infatti se pensiamo al funzionamento di un sistema di *tracking* di veicoli, la rilevazione di movimento di un oggetto riguarda una zona limitata di spazio, per cui sarà richiesto un servizio di sincronizzazione localizzato. Lo stesso vale se si vuole implementare tecniche TDMA in quando la condivisione del canale di trasmissione avviene solo tra nodi vicini, e per quei nodi si richiede un alto grado di sincronizzazione. Con un periodo di sincronizzazione di 60 secondi l'algoritmo permette di raggiungere un errore di diffusione medio di 53.5 μs (1.7115 tick / 32 KHz) utilizzando una rete di 3x3 nodi e come sorgente hardware di tempo l'oscillatore esterno a 32 KHz del nodo. I risultati sono paragonabili agli algoritmi presenti in letteratura, in particolare è stato possibile effettuare un confronto sperimentale con l'algoritmo FTSP, ottenendo risultati positivi. Si è dimostrato che l'algoritmo è altamente robusto (ai guasti e a cambiamenti di topologia di rete), in quanto garantisce il servizio di sincronizzazione in caso di variazione del numero di nodi della rete.

Il lavoro svolto si presta ad ulteriori studi ed approfondimenti. Ulteriori miglioramenti possono essere apportati al protocollo utilizzando la tecnica del "Piggybacking" in modo da sfruttare tutti i messaggi che il nodo invia per fare sincronizzazione. Così facendo si evita di spedire appositi pacchetti per effettuare la sincronizzazione.
Inoltre si potrebbe sfruttare il servizio di sincronizzazione sviluppato come base fondamentale per l'implementazione di una infrastruttura che permetta la gestione d'accesso al mezzo di trasmissione da parte dei nodi (TDMA scheduling) o più in generale per poter coordinare qualsiasi tipo di azione di gruppo.

Bibliografia

[1] K. Römer, M. Friedemann 2004. "The Design Space of Wireless Sensor Networks". IEEE Wireless Communications 11 (6): 54-61.

[2] D.L. Mills 1991. "Internet time synchronization: the network time protocol". IEEE Trans. Communications 39, 10 (Oct.), 1482-1493.

[3] D.L. Mills 1994. "Improved algorithms for synchronizing computer network clocks". In Proc. of ACM Conference on Communication Architectures (ACM SIGCOMM '94). London, UK.

[4] D.L. Mills 2006. "Network Time Protocol Version 4 Reference and Implementation Guide". http://www.eecis.udel.edu/%7emills/database/reports/ntp4/ntp4.pdf.

[5] National Institute of Standards and Technology, IEEE 1588 web site. http://ieee1588.nist.gov/.

[6] "Precision clock synchronization protocol for networked measurement and control systems". INTERNATIONAL STANDARD. IEC 61588. First edition 2004-09.

[7] A. McCarthy. "Special Focus: Understanding the IEEE 1588 Precision Time Protocol". http://zone.ni.com/devzone/cda/pub/p/id/130.

[8] A. Pakdaman, T. Cooklev. "IEEE 1588 over IEEE 802.11b for Synchronization of Wireless Local Area Network Nodes". San Francisco State University John Eidson, Agilent Technologies. http://ieee1588.nist.gov/Presentation%20PDFs/4_IEEE%20802.11b_afshaneh.pdf.

[9] J. Elson, L. Girod, D. Estrin 2002. "Fine-grained network time synchronization using reference broadcasts". In Proceedings of the 5th symposium on Operating systems design and implementation (OSDI'02), pages 147-163, 2002.

[10] S. Yoon, C. Veerarittiphan, M.L. Sichitiu. “Tiny-sync: Tight time synchronization for wireless sensor networks”. ACM Journal of Sensor Networks, 3(2), 2007.

[11] S. Ganeriwal, R. Kumar, M.B. Srivastava 2003. “Timing-sync protocol for sensor networks”. In Proceedings of the first international conference on Embedded networked sensor systems (SenSys’03), pages 138-149, 2003.

[12] J. van Greunen, J. Rabaey. “Lightweight time synchronization for sensor networks”. In 2nd ACM International Workshop on Wireless Sensor Networks and Applications, pages 11-19, September 2003.

[13] M. Maròti, B. Kusy, G. Simon, A. Lédeczi 2004. “The flooding time synchronization protocol”. In SenSys ’04: Proceedings of the 2nd international conference on Embedded networked sensor systems. ACM Press, pages 39-49, 2004.

[14] Q. Li, D. Rus 2006. “Global Clock Synchronization in Sensor Networks”. IEEE Transactions on computer, vol. 55, no.2, february 2006.

[15] G. Werner-Allen, G. Tewari, A. Patel, M. Welsh, R. Nagpal. “Firefly-inspired sensor network synchronicity with realistic radio effects”. In ACM Conference on Embedded Networked Sensor Systems (SenSys’05), San Diego, November 2005.

[16] O. Simeone, U. Spagnolini. “Distributed time synchronization in wireless sensor networks with coupled discrete-time oscillators”. EURASIP Journal on Wireless Communications and Networking, 2007:Article ID 57054, 13 pages, 2007. doi:10.1155/2007/57054.

[17] R. Solis, V. Borkar, and P. R. Kumar. “A new distributed time synchronization protocol for multihop wireless networks”. In To appear in 45th IEEE Conference on Decision and Control (CDC’06), San Diego, December 2006.

[18] L. Schenato, G. Gamba. “A distributed consensus protocol for clock synchronization in wireless sensor network”. In to appear in Proceedings of IEEE Conference on Decision and Control, 2007.

[19] Crossbow Technology Inc. web site, http://www.xbow.com.

[20] Moteiv Corporation web site,
http://www.moteiv.com - http://www.sentilla.com.

[21] Ember Corporation web site, http://www.ember.com.

[22] Chipcon AS. web site, http://www.chipcon.com.

[23] IEEE 802.15 WPAN Task Group 4 (TG4) web site, http://www.ieee802.org/15/pub/TG4.html.

[24] Zigbee Alliance web site, www.zigbee.org/.

[25] P. Levis. "TinyOS 2.0 Overview". http://www.tinyos.net/tinyos-2.x/doc/html/overview.html.

[26] D. Gay, P. Levis, D. Culler, E. Brewer. "nesC 1.1 Language Reference Manual". May 2003.

[27] C. Duffy. "The TinyOS-2.x Priority Level Scheduler". http://www.cs.ucc.ie/ cd5/PLScheduler.html

[28] tinyos.cvs.sourceforge.net http://tinyos.cvs.sourceforge.net/tinyos/tinyos-2.x-contrib/ethz/snpk/tos/lib/net/TimeSync/