1. La Sincronizzazione del clock

Un sistema distribuito è costituito da un insieme di processori che comunicano attraverso lo scambio di messaggi per realizzare una applicazione. Solitamente questi processori operano in modo autonomo (seppur coordinato) e non hanno accesso ad un clock comune in quanto non necessitano di un riferimento temporale unico. Tuttavia, esiste un numero molto elevato di applicazioni che richiedono la disponibilità di un clock sincronizzato, in modo da avere una stessa visione del tempo. Queste applicazioni sono:

- Applicazioni per controllo di processo in cui le azioni di controllo, spesso legate al valore di dati campionati, si riferiscono a specifici istanti di tempo (istanti di campionamento). Si prenda in considerazione un processo gestito da un PLC che elabora i dati forniti da diversi sensori. Il PLC ripete periodicamente lo stesso algoritmo elaborando i dati forniti da vari sensori. Una stretta correlazione temporale fra i produttori dei dati (sensori) ed il consumatore (PLC) è indispensabile per un corretto controllo di processo.
- Applicazioni di misura, in cui i valori acquisiti da diversi sensori autonomi, vanno riferiti ad un'unica base dei tempi. In tale contesto, una delle applicazioni più emblematiche è il Distributed beam-forming array cioè una distribuzione lineare di sensori che viene utilizzato per individuare la sorgente di un segnale. Ad esempio, un array di microfoni può essere usato per localizzare la sorgente di un suono. L'array calcola la differenza di fase dei segnali ricevuti dai sensori nelle diverse locazioni. Da tali differenze di fase il processore può risalire al tempo di percorrenza del suono dalla sorgente ai singoli sensori e può quindi localizzare la sorgente rispetto al riferimento spaziale costituito dall'array. Tale approccio si basa sulla assunzione che tutti i sensori siano sincronizzati temporalmente in quanto la computazione è eseguita assumendo che le differenze di fase misurate dipendano da differenze nel tempo di viaggio dei segnali dalla sorgente ai sensori e non da ritardi. In un sistema centralizzato, in cui esiste un accoppiamento molto stretto fra il processore ed i sensori, tale assunzione è ragionevole poiché i vari dati audio condividono una base dei tempi implicita in quanto essi sono processati da un'unica CPU. Nel caso di una implementazione attraverso un insieme di sensori wireless autonomi, occorre invece fare riferimento ad una base dei tempi esplicita per avere una informazione affidabile.
- Applicazioni transazionali, in cui dei processi evolvono nel rispetto di vincoli di precedenza o di contemporaneità come ad esempio nelle transazioni bancarie, o in generale in algoritmi di distributed agreement (*accordo simultaneo*). Un'applicazione di distributed agreement può essere ad esempio quella della partenza simultanea di una catena di montaggio in una fabbrica, in cui l'esistenza di un tempo comune costituisce una delle principali condizioni richieste (anche se non l'unica, in quanto occorre gestire altri problemi quali ad esempio quelli legati ai guasti).

- Applicazioni relative alla gestione di protocolli di comunicazione. E' questa un'area applicativa in cui il tempo svolge un ruolo essenziale. Si pensi ad esempio a protocolli a livello di Data link per l'accesso a canale comune. Nelle applicazioni di controllo di processo, è spesso richiesto che l'accesso al canale di comunicazione avvenga in istanti ben precisi, nel rispetto di una determinata schedulazione. Ciò richiede l'esistenza di un riferimento temporale comune.

Diversi altri problemi di misura possono richiedere una stretta sincronizzazione dei clock. Ad esempio, la propagazione di un'onda di pressione lungo una condotta, rivelata da una serie di sensori distribuiti lungo il percorso o la individuazione della traiettoria di un oggetto veloce, monitorato da sensori distribuiti nello spazio.

Il tempo di cui parliamo in questo tutorial, viene inteso come una approssimazione del tempo reale fisico (il tempo reale viene solitamente indicato come real-time da non confondere con la denominazione real-time utilizzata per i sistemi dipendenti dal tempo) o semplicemente come il valore contenuto da un contatore fisico che si usa come riferimento.

Gli algoritmi che permettono ad un set di processori distribuiti, di avere la stessa visione del tempo vengono chiamati **algoritmi per la sincronizzazione dei clock**. Obiettivo della sincronizzazione è stabilire e mantenere, a livello di intero sistema, un tempo di riferimento consistente fra i vari sottosistemi di un sistema real-time distribuito. Il problema della sincronizzazione dei clock, è oggi divenuto particolarmente importante con riferimento alla sua applicazione nei sistemi distribuiti wireless di misura. Tali sistemi, essendo costituiti da un insieme di sensori che devono operare in modo coordinato, richiede una attenta sincronizzazione dei clock in modo che sia possibile generare un tempo di riferimento, comune a tutti i componenti del sistema.

Questo tutorial sarà articolato in tre parti. Nella prima verrà presentata una classificazione delle problematiche e verranno fornite alcune definizioni utili ad inquadrare la problematica della sincronizzazione dei clock. Nella seconda parte verranno esaminati alcuni degli approcci più noti in letteratura, che sono di interesse per l'applicazione considerata. Nella terza parte verranno discussi gli approcci possibili per le reti wireless e verranno esaminati i problemi legati all'ambiente operativo tipico del wireless.

2. Concetti e definizioni introduttive

Il problema della sincronizzazione dei clock può essere affrontato da diversi punti di vista a seconda della granularità desiderata nella sincronizzazione, delle varie astrazioni cui si fa ricorso nel descrivere il sistema da sincronizzare e dei requisiti

delle applicazioni che si intende sincronizzare. Possiamo riferirci alla sincronizzazione a livello di processo (task) o a livello di nodo (clock).

La sincronizzazione a livello di task ha a che fare con l'ordinamento di eventi, mentre la sincronizzazione a livello di nodi di solito si riferisce alla sincronizzazione di clock fisici o logici nei nodi (i clock fisici sono generati da un contatore che avanza continuamente, i clock logici sono ottenuti attraverso fattori di correzione sui clock fisici).

Il modello del sistema influenza il tipo di sincronia che deve essere mantenuta. Nei sistemi sincroni le computazioni sono eseguite secondo "frames" dove una frame è un intervallo fisso di tempo opportunamente delimitato. La sincronizzazione in questi sistemi viene ottenuta garantendo che i delimitatori delle frames, nei vari nodi, siano l'un l'altro, entro un prefissato intervallo di tempo (skew). I sistemi asincroni per contro non utilizzano esplicitamente la nozione di tempo. La serializzazione di azioni e task viene ottenuta utilizzando meccanismi di *lock*, *interrupt*, *sistem calls* e *semafori*. Tuttavia, la coordinazione di azioni secondo certi eventi è una forma implicita di sincronizzazione. I modelli di alcuni sistemi asincroni assumono che i singoli nodi possiedano una base dei tempi o siano temporalmente coordinati, pur mantenendo asincrone le operazioni fra i vari nodi. Quest'ultimo tipo di sistema asincrono, che si può definire *sistema quasi-sincrono,* è caratterizzato dal fatto che esso può supportare la presenza di set di nodi mutuamente sincroni, se richiesto, con frame temporali di diversa lunghezza fra i diversi set di nodi.

La progettazione di algoritmi per la sincronizzazione di clock presenta diversi problemi:

1. Innanzitutto, a causa dei diversi ritardi di trasmissione ogni processo non può avere una vista istantanea globale di tutti gli altri clock. Questo è uno dei principali problemi legati alla sincronizzazione in quanto i diversi tempi di comunicazione fra i processi influenza negativamente la qualità le operazioni di correzione del clock.
2. Inoltre, anche se tutti i *clock* fossero inizializzati contemporaneamente, essi non rimarrebbero sincronizzati a causa del *drift* nei rispettivi periodi. In realtà, i drift possono avere entità variabile a seconda della diversa qualità del clock fisico utilizzato, i clock possono avanzare con un *rate* che differisce dal tempo reale (real-time) anche di 10^{-5} secondi per secondo e possono quindi portare ad 1 secondo di errore al giorno. Il drift rate inoltre non è costante nel tempo e può modificarsi con la temperatura o con l'invecchiamento dei circuiti. Ciò complica ulteriormente il problema poiché richiede che l'aggiustamento sia reiterato periodicamente.
3. Infine, la presenza di elementi malfunzionanti (faulty) distribuiti nel sistema, complica ulteriormente il raggiungimento di una visione unica del tempo. Questo problema può essere particolarmente sentito nei sistemi wireless per applicazioni mobili, dove la posizione dei vari nodi può portare temporaneamente a zone

d'ombra, in cui la comunicazione è impossibile o in generale a causa dei disturbi che possono danneggiare diversi messaggi di sincronizzazione e quindi peggiorarne la qualità.

Gli algoritmi per la sincronizzazione dei clock possono essere usati per sincronizzare i clock con riferimento ad una sorgente esterna o per sincronizzarli fra di loro:

- Nel primo caso, definito *external clock synchronization*, viene usato un clock come riferimento, cioè come sorgente del tempo reale e l'obiettivo della sincronizzazione è fare in modo che tutti i clock si avvicinino, per quanto possibile, al tempo reale. La massima differenza fra ciascuno dei clock ed il riferimento esterno, viene definita *accuratezza*. Il clock di riferimento (MASTER CLOCK) è in genere di elevata qualità e tutti gli altri clock, di qualità inferiore, devono restare agganciati ad esso. E' questo il caso in cui occorre mantenere la lettura del clock più vicina possibile all'UTC (Universal Coordinated Time generato da un orologio atomico. Una applicazione è quella di usare un Central Time Server che riceve il segnale ad onda corta (WWW) generato da Fort Collins e poi ritrasmette periodicamente tale segnale ad altri Time Servers.
  Un sottocaso dell'External Clock Synchronization può essere considerato il Phase Synchronization in cui si assume che tutti clock abbiano esattamente la stessa velocità (*rate*) in quanto pilotati da una sorgente unica, così che in teoria evolvono in modo perfettamente sincrono. Se ogni contatore ha modulo K, ognuno evolve dal valore 0 a K-1 mantenendosi sincronizzato con gli altri. La perdita di qualche campione da parte di un nodo produce però errori nel conteggio e quindi perdita di sincronizzazione nei clock. Questo è comunque un caso molto particolare di sincronizzazione, che non verrà ulteriormente approfondito nel seguito.
- Nel secondo caso, definito *internal clock synchronization*, non è disponibile una sorgente del tempo reale e l'obiettivo della sincronizzazione è fare in modo che i vari clock differiscano l'un l'altro il meno possibile. Lo scopo è quindi ottenere un allineamento dei singoli clock (che in genere sono della stessa qualità) per generare un riferimento temporale comune. La massima differenza fra ciascuna coppia di clock ottenuta dall'algoritmo è chiamata la sua *precisione*.

Gli algoritmi di sincronizzazione dei clock possono inoltre essere classificati come asimmetrici o simmetrici. Questa proprietà influenza la capacità dell'algoritmo di tollerare guasti e determina il suo costo in termini di messaggi che occorre scambiare per ottenere la precisione di sincronizzazione desiderata. Negli schemi di tipo asimmetrico (anche chiamati di tipo Master/Slave) i nodi coinvolti nella sincronizzazione del clock, giocano ruoli diversi. Un ruolo specifico è assegnato ad un nodo predefinito chiamato Master, cui tutti gli altri nodi (Slaves) fanno riferimento per la sincronizzazione. Il vantaggio più evidente degli schemi asimmetrici è il loro basso costo in termini di risorse utilizzate, legato alla natura centralizzata dello schema, anche se non va trascurato come il Master rappresenti un singolo punto di fallimento del sistema. Negli schemi di tipo simmetrico, invece, ogni nodo gioca lo stesso ruolo degli altri, per quanto riguarda la sincronizzazione. Gli algoritmi sono

completamente distribuiti e frequentemente la sincronizzazione è basata su passi indicati come *full message exchange*, in cui ogni nodo comunica con tutti gli altri nodi del sistema, causando un elevato uso delle risorse. Il maggiore vantaggio degli schemi simmetrici è relativo alla loro elevata fault-tolerance che dipende dall'assenza di singoli punti di fallimento.

La sincronizzazione di clock può essere ottenuta via Hardware o via Software. La sincronizzazione hardware [SR87], [KO87] porta ad una sincronizzazione molto stretta attraverso l'uso di un hardware dedicato in ogni processore e di solito utilizza una rete di comunicazione ad hoc solo per la distribuzione dei clock. Per contro, gli algoritmi di sincronizzazione software del clock [CAS86], [ST87], [LM85], [CRI89] usano reti di comunicazione standard e spediscono messaggi di sincronizzazione per ottenere la sincronizzazione dei clock. Tali algoritmi non necessitano di un hardware specializzato ma non sono in grado di ottenere risultati altrettanto buoni di quelli forniti dalla sincronizzazione hardware.

Gli algoritmi di sincronizzazione software possono essere classificati in deterministici, probabilistici e statistici:

- Gli algoritmi deterministici assumono sempre che i ritardi introdotti dal sistema di comunicazione siano superiormente limitati possiedano cioè un *upper bound*. Essi garantiscono quindi un valore max nella differenza fra due qualunque clock, che misura la precisione raggiungibile da tali algoritmi.
- Gli algoritmi probabilistici non assumono invece un valor max, noto, nel ritardo di propagazione e non utilizzano alcun modello di distribuzione dei ritardi, ma garantiscono una deviazione max fra i valori di due qualunque clock, con una certa probabilità (strettamente minore di 1). In ogni caso, in ogni istante un clock attraverso lo scambio di opportune informazioni, conosce se esso è sincronizzato ed in quale misura, con gli altri.
- Anche gli algoritmi statistici non assumono un valore max, noto, nel ritardo di propagazione, ma utilizzano un modello di distribuzione dei ritardi e su questa base, senza bisogno di informazioni sul livello di sincronizzazione raggiunto, garantiscono una deviazione max fra i valori di due qualunque clock, con una certa probabilità (strettamente minore di 1).

2.1. Il modello di un sistema.

I processori.

Questa sezione del tutorial si basa sul lavoro di E. Anceaume ed I. Pouat descritto in [AP98]. Si consideri un insieme P= (p_1, p_2, …p_n) di processori interconnessi da un sistema di comunicazione. A ciascun processore $p_i \in P$ è possibile associare un clock hardware Hp_i che generalmente consiste di un oscillatore e da un contatore che viene incrementato dai *tick* dell'oscillatore. Va sottolineato come, nonostante il clock sia costituito da una sequenza di eventi discreti, tutti gli algoritmi di sincronizzazione

assumono che il tempo scorra in modo continuo cioè che il clock sia una funzione continua per qualunque intervallo del tempo reale. I clock hardware deviano dal tempo reale a causa del loro invecchiamento e delle variazioni di temperatura; in ogni caso però si assume che la loro deviazione è all'interno di un intervallo noto ρ e si parla quindi di *ρ-bounded clock*.

Assunzione 1: (ρ-bounded clock)

Sia $\rho > 0$ una costante nota, si dice che il clock hardware $H_{p_i}(t)$ è ρ-bounded se vale la seguente relazione per ogni valore *t* del tempo reale:

$$\frac{1}{(1+\rho)} \leq \frac{dH_{p_i}(t)}{dt} \leq (1+\rho)$$

dove $H_{p_i}(t)$ è il valore del clock hardware del processore p_i al tempo *t*.

Gli algoritmi di sincronizzazione dei clock non sincronizzano direttamente i clock hardware (che invece avanzano sempre autonomamente), ma piuttosto introducono un clock logico Lp_i. Tale clock logico al tempo reale t, indicato come $Lp_i(t)$ viene ottenuto attraverso l'introduzione di un termine di correzione che può essere sia un valore discreto cambiato ad ogni risincronizzazione [SC90] che una funzione lineare del tempo [CRI89]. Ciò permette di separare il clock hardware dal clock logico presente in ogni nodo, lasciando che il clock hardware evolva autonomamente e introducendo le correzioni solo nel clock logico. Apportare le correzioni direttamente al clock hardware implica una stretta interazione fra il software di sincronizzazione e l'Hardware che genera il clock e ciò richiede delle architetture Hw/Sw specializzate che limitano fortemente la generalità di eventuali algoritmi di sincronizzazione. Un esempio notevole di tale approccio si trova nel protocollo BlueTooth in cui il clock delle stazioni slave si sincronizza con quello del Master attraverso la trasmissione di un opportuno OffSet fra i due clock.

Poiché come si è detto, il clock che usiamo come riferimento deve essere caratterizzato da una elevata stabilità, possiamo confrontarlo col real-time t (cioè il tempo assoluto misurato con un riferimento ideale) e introdurre il concetto di "bounded external deviation" asserendo che un clock di riferimento H_p può essere considerato corretto al tempo t quando soddisfa la condizione:

$$\left| H_{p_r}(t) - t \right| \leq \Delta$$

ove Δ indica un errore predefinito, noto a priori.

La rete di comunicazione

Gli algoritmi di sincronizzazione dei clock operano mediante lo scambio, attraverso una rete di comunicazione, di opportuni messaggi di sincronizzazione. Per tale motivo appare particolarmente importante la valutazione del ritardo introdotto dalla rete che influenza l'accuratezza della sincronizzazione ottenibile. I valori dei tempi di ritardo introdotti dalla rete possono essere estremamente variabili. Se consideriamo le reti wired, le topologie possono spaziare dal singolo Bus (una rete locale Ethernet)

in cui i tempi di propagazione sono normalmente ridotti, alla internet, in cui i messaggi di sincronizzazione devono attraversare un notevole numero di sottoreti (e routers) rendendo i ritardi estremamente variabili. Nel caso di internet, la dimensione della rete e la variabilità dei ritardi impone l'uso di architetture gerarchiche per la sincronizzazione.

Problematiche analoghe (anche se peggiorate dalla scarsa qualità della comunicazione) si incontrano nelle reti wireless. In una rete totalmente connessa in cui tutti i messaggi viaggiano per la stessa distanza fisica (è questo il caso di una rete wireless se operiamo nel contesto di una singola cella) il tempo di propagazione può essere stimato all'interno di un intervallo ben definito e può spesso essere considerato uguale per tutti i nodi. Se invece operiamo nel contesto di una rete multi-hop, le varie distanza non sono uniformi ed il tempo di propagazione può introdurre errori sostanziali.

- Possiamo assumere che il ritardo δ della comunicazione sia limitato nell'intervallo $(\delta-\varepsilon, \delta+\varepsilon)$ ed in tal caso è possibile realizzare algoritmi di sincronizzazione deterministici che possono garantire una precisione definita, nel senso che ogni clock logico differisce da tutti gli altri di una quantità max calcolabile.
- Per contro possiamo assumere che non sia possibile definire un ritardo max nella comunicazione (come avviene per esempio in caso di sovraccarico o di reti molto estese). In tal caso si può convenire di modellare il ritardo di comunicazione come una variabile con valori arbitrari o come una variabile di cui sia noto il ritardo medio e la deviazione standard. Nel primo possiamo parlare di algoritmi di sincronizzazione di tipo probabilistico mentre nel secondo parliamo di algoritmi di tipo statistico. In entrambi i casi, è possibile garantire la sincronizzazione solo con una certa probabilità non nulla.

I modelli di fallimento.

Il problema della sincronizzazione dei clock viene ulteriormente complicato dalla presenza di fault (guasti) nel sistema che in qualche modo diminuiscono l'efficienza dei vari algoritmi di sincronizzazione. Si introduce quindi il concetto di fallimento, definito come la deviazione del sistema da un comportamento corretto a causa di qualche malfunzionamento. Tutti i componenti del sistema (i clocks, i processori dei singoli nodi ed i link di comunicazione) possono essere soggetti a guasti/malfunzionamenti.

Con riferimento ai clock, possono essere considerati guasti di tipo incontrollato (anche definiti di tipo bizantino [LSP82] che possono produrre valori molto in accurati e/o contrastanti l'un l'altro) oppure fallimenti di temporizzazione più limitati, che escludono lo scambio di messaggi in conflitto (i così detti comportamenti malizioni) e vengono definiti *Timing failure* [CASD85].

- *Clock timing failure*: il clock ha un funzionamento scorretto nel senso che non è ρ- bounded.
- *Clock byzantine failure*: il clock fornisce informazioni inaccurate, scorrette o in conflitto. Questo tipo di guasto include la presenza dei cosidetti dual-faced clocks (clock a doppia faccia) i quali forniscono differenti valori del clock a processori differenti.

Per quanti riguarda i possibili guasti dei processori, la casistica si limita a tre possibilità:

- Processor crash failure: il processore che si comportava correttamente, ad un certo istante cessa per sempre la sua attività. [CASD85]. Questo tipo di guasto è facile da individuare, e se il sistema considerato possiede un numero abbastanza elevato di elementi (nodi) l'effetto di questo guasto viene superato mediante l'uso degli altri elementi del sistema.
- Processor performance failure: il processore completa le proprie computazioni in un tempo superiore a quello impiegato solitamente [CASD85]. Questo tipo di guasto può essere dannoso per la qualità della sincronizzazione poiché non può essere rilevato dagli altri elementi del sistema che continuano ad operare in modo normale. Si consideri ad esempio la presenza in un nodo di un task ad alta priorità che in istanti non definiti va in esecuzione rallentando l'esecuzione di operazioni indispensabili per la sincronizzazione. Gli algoritmi continueranno ad operare in maniera normale, non rendendosi conto che un valore da essi utilizzato è affetto da una in accuratezza superiore ai valori attesi. La conseguenza è una minore accuratezza complessiva.
- Processor Byzantine failure: il processore esegue delle computazioni incontrollate [LSP82]. Questo tipo di guasto è il più dannoso essenzialmente per due ragioni : innanzitutto, il tipo di errore introdotto può essere molto grave (ben oltre una semplice inaccuratezza) in quanto, ad esempio un errore nel software causato da un guasto nella RAM porta ad eseguire operazioni non previste che forniscono valori completamente errati. Inoltre, è un errore che può ripetersi sporadicamente rendendone difficile l'individuazione da parte degli altri nodi.

Infine, con riferimento alla rete di comunicazione (sia essa di tipo broadcast o di tipo punto-punto) i malfunzionamenti possono essere classificati come *errori di omissione* o *performance failure*:

- *Link omission failure*: un link **l** da un processore P_i ad un processore p_j commette una *omission failure* su un messaggio **m** se m viene inserito nel buffer d'uscita di P_i ma **l** non lo trasporta fino al buffer d'ingresso di p_j. Il fallimento consiste quindi nella perdita del messaggio da trasportare o a causa di un disturbo nella rete o a causa di sovraccarichi in qualche nodo intermedio o di traffico eccessivo.
- *Link performance failure*: un link l commette un link performance failure se, per trasportare un messaggio m, esso impiega un tempo superiore a quello massimo

specificato. Ovviamente, questa definizione può applicarsi solo a link basati su protocolli di comunicazione che garantiscono un ritardo massimo.

Poiché la comunicazione di un messaggio coinvolge, oltre al link anche un processo sender ed uno receiver, una link failure può dipendere da una failure in uno (o più) elementi o in uno o più nodi del sistema. La maggior parte degli algoritmi di sincronizzazione dei clock assume che il numero di componenti guasti (clocks, processori e link) del sistema sia limitato come espresso nella seguente definizione:
Bounded number of faulty components: in un sistema possono esserci al massimo **f** componenti faulty durante l'esecuzione dell'algoritmo di sincronizzazione dei clock, con **f** intero positivo. Il valore di f dipende dal tipo di fallimento considerato. Se ad esempio consideriamo fallimenti di tipo bizantino nei processori deve essere **f ≤ n/3** dove **n** è il numero di processori [LL84].

3. Il problema della sincronizzazione dei clock.

Dati due processi con clock *c* e *c'* diremo che questi sono sincronizzati al più di δ al tempo T se:

$$| c(T) - c'(T)| < \delta \qquad (1)$$

cioè i due processi raggiungeranno il valore del tempo reale T al più con una differenza δ l'uno dall'altro. Questa proprietà afferma che due clock logici corretti sono approssimativamente uguali. Tutti gli algoritmi di sincronizzazione deterministici soddisfano questa proprietà mentre quelli statistici e probabilistici soddisfano questa proprietà con una probabilità strettamente minore di uno.
Le azioni generate dai due processi, quindi, potranno accadere con δ T di ritardo al massimo. Sebbene questa proprietà sia una condizione necessaria per la sincronizzazione dei clock, essa non risulta sufficiente; infatti vale anche nel caso in cui tutti i clock logici sono inizialmente posti a zero e non vengono fatti partire. Le seguenti proprietà eliminano ogni soluzione banale al problema della sincronizzazione:
Proprietà 1: (Bounded correction) esiste un piccolo intervallo *Σ*, con il quale un clock corretto viene modificato in seguito ad un processo di risincronizzazione.
Proprietà 2: (Accuracy) per ogni processore $p_i \in P$ e per ogni istante di tempo reale *t*, esiste una costante *v*, detta accuratezza, tale che per ogni esecuzione dell'algoritmo di sincronizzazione del clock si ha:

$$(1+\nu)^{-1} \cdot t + a \le L_{p_i}(t) \le (1+\nu) \cdot t + b$$

con *a* e *b* costanti dipendenti dalle condizioni iniziali dell'algoritmo.

La migliore accuratezza *v* ottenibile da un algoritmo di sincronizzazione è pari al drift del clock hardware ρ (accuratezza ottima).

Diversi algoritmi per la sincronizzazione sono noti in letteratura, per tutti lo scopo è quello di ottenere un tempo comune all'interno del sistema tra i vari utenti. Questi algoritmi implementano una graduale correzione del clock per compensare il drift tra i clock locali ed arrivare ad ottenere una visione comune del tempo. Tutti gli utenti presenti nel sistema si sincronizzano periodicamente rispetto ad un valore di clock di riferimento (sia esso del master o esterno al sistema come l'Universal Time Coordinate (UTC) usato dal Network Time Protocol (NTP), uno standard dell'Internet Protocol) o tentano di convergere verso un valore comune. L'ingresso di nuovi utenti o l'uscita di altri (per es. per guasti) destabilizza il sistema e quindi sarà necessario risincronizzare il sistema. La risincronizzazione dovrà essere effettuata, comunque periodicamente all'interno del sistema in quanto tutti gli utenti hanno un clock che avanza con un clock rate diverso da quello degli altri e da quello reale.

La frequenza della risincronizzazione è una funzione del modello del sistema e delle condizioni operative. L'ideale sarebbe effettuare una risincronizzazione ad ogni tick del clock, operazione impossibile nella quasi totalità dei sistemi. In un intervallo di risincronizzazione R, se ρ è il drift di un generico clock, due clock differiranno al massimo di $2R\rho$. E' evidente come al crescere dell'intervallo R cresca anche lo skew fra i due clock. L'efficienza degli algoritmi di sincronizzazione si valuta quindi anche in funzione del max valore di R che può garantire una prefissata precisione.

In ciascun nodo il processo che si occupa della sincronizzazione è detto *time server process*. Un protocollo di sincronizzazione consiste in un gruppo di *time server process* che sono localizzati nei differenti nodi nella rete. Ciascun *time server process* può leggere il valore del clock hardware (HC) del suo nodo host. Sia HC(t) il valore del clock hardware del generico nodo al tempo reale *t*. Diremo che questo valore è corretto se, dato l'intervallo di tempo reale $[t_1, t_2]$ e il massimo tasso di drift del clock ρ, fornito dal costruttore, accade che:

$$(1-\rho)\,(t_2-t_1) \leq HC(t_2) - HC(t_1) \leq (1+\rho)\,(t_2-t_1)$$

cioè si ha un errore della misura al massimo di $\rho(t_2-t_1)$. Questa condizione, soddisfa la definizione di ρ-*bounded drift*.

Assumendo che la velocità del HC non possa essere modificata via software, viene definito il clock logico (LC), che è implementato via software e può essere aggiustato. Il valore di LC al tempo *t* può essere così espresso:

$$LC(t) = HC(t) + A(t)$$

dove A è la funzione di correzione e dipende dal tempo. Il clock logico rappresenta un'approssimazione del valore del tempo reale ed è modificato in accordo con i parametri della funzione A(t).
Alcuni algoritmi di sincronizzazione ipotizzano che vi sia una sincronizzazione iniziale dei clock, espressa dalla seguente assunzione:
Assunzione 1: (Initial synchronization) per due qualunque processori p_i e p_j, indicato con T_0 il tempo segnato dai clock logici nell'istante in cui p_i e p_j iniziano l'algoritmo di sincronizzazione, si ha che $|l_i(T_0) - l_j(T_0)| \leq \beta$, dove $l_i(T_0)$ e $l_j(T_0)$ sono i valori del tempo reale quando i clock logici rispettivamente di p_i e p_j segnano il valore T_0, e β è il valore del tempo reale.

Negli algoritmi di *external clock synchronization*, il problema principale è fornire una misura per quanto possibile precisa del tempo reale a ciascun *time server*, come misura di un clock localizzato in un nodo privilegiato. I diversi tipi di protocolli assumono che sia noto il minimo tempo di trasmissione, *min*. Esso esiste in quanto è dovuto al ritardo di elaborazione del trasmettitore e del ricevitore e al ritardo di propagazione del segnale sul mezzo. È calcolato sommando i tempi richiesti per trasmettere un messaggio vuoto tra due nodi adiacenti della rete, nell'ipotesi che non ci sia altro carico nella rete e che non avvengano guasti nella trasmissione. Nella realtà esso non è noto a priori ed è spesso stimato usando dei metodi empirici.

3.1 Elementi costitutivi di un Algoritmo di Sincronizzazione dei Clock.

In un algoritmo di sincronizzazione, qualunque sia la sua struttura, ogni processore $p_i \in P$ deve eseguire tre fasi in successione: primo, deve determinare l'istante in cui eseguire la risincronizzazione, utilizzando il *resynchronization event detection block;* secondo, deve stimare il valore dei clock logici remoti, utilizzando il *remote clock estimation block*; terzo, deve applicare il termine di correzione al suo clock logico in base al risultato ottenuto nella seconda fase, invocando il *clock correction block*. Le tecniche di realizzazione di questi blocchi funzionali variano al variare dell'algoritmo considerato, ma in generale sono equivalenti, quindi possono essere impiegate a prescindere dalla natura dell'algoritmo di sincronizzazione (deterministico, probabilistico, statistico, interno, esterno, ecc.).

3.1.1.Resynchronization event detection block

La funzione di questo blocco è quella di lanciare un evento di risincronizzazione, che informa il processore p_i di avviare l'algoritmo di sincronizzazione. Per garantire il soddisfacimento della relazione (1): $|c(T) - c'(T)| < \delta$, il processo di risincronizzazione deve essere effettuato con una frequenza che dipende dal drift del clock. Un tipico modo per risincronizzare i clock è quello di lanciare periodicamente l'algoritmo di sincronizzazione, in questo caso l'algoritmo viene detto round-based. Il

problema degli algoritmi round-based è quello di stabilire l'istante in cui devono cominciare ad essere eseguiti. Due sono le tecniche utilizzate: la prima si basa sull'ipotesi che i clock siano inizialmente sincronizzati; la seconda utilizza lo scambio di messaggi.

Tecnica 1: questa è la tecnica più adottata e assume che i clock siano inizialmente sincronizzati con un'accuratezza β (Assunzione 1). Il processore p_i stima l'inizio del round k ($k \in N, k \geq 1$) nell'istante in cui il suo clock logico segna il valore kR, dove R è la periodicità cioè l'intervallo di tempo, espresso in numero di cicli del clock logico, tra due successivi processi di sincronizzazione.
Intuitivamente, per mantenere i clock il più possibile sincronizzati, i valori di β ed R devono essere più piccoli possibile. È stato provato che un valore arbitrariamente piccolo di R non permette la distinzione tra round successivi.

Tecnica 2: la precisione γ, ottenuta dagli algoritmi che fanno affidamento su una sincronizzazione iniziale dei clock con accuratezza β, è molto sensibile al valore di β. Un metodo alternativo per determinare l'istante d'inizio di un round utilizza lo scambio di messaggi tra i processori di P. Un dato processore di P, nell'istante in cui il suo clock logico segna il tempo kR, invia ai restanti processori un messaggio. Ciascun processore $p_i \in P$ alla ricezione del messaggio, dà il via ad un nuovo round di sincronizzazione. Naturalmente la precisione dell'algoritmo dipende dal ritardo di propagazione dei messaggi; nelle reti di tipo broadcast, dove i ritardi di trasmissione hanno una minore varianza, la precisione è notevolmente migliore.

3.1.2. Remote clock estimation block.

L'obiettivo di questo blocco è quello di ottenere informazioni sui clock remoti. A causa del drift dei clock e del tempo di propagazione non nullo e variabile dei messaggi, è impossibile ottenere informazioni esatte sul valore dei clock remoti. Conseguentemente il valore dei clock remoti può essere solamente stimato. Il remote clock estimation block del processore p_i viene attivato alla ricezione dell'evento di risincronizzazione generato dal resynchronization event detection block di p_i. L'uscita del remote clock estimation block del processore p_i, costituita da un insieme di stime dei valori dei clock remoti (indicata col nome di *clock estimation set*), viene data in ingresso al clock correction block locale.
Le due tecniche possibili per stimare i valori dei clock remoti, note col nome di *Time Transmission* (TT) e *Remote Clock Reading* (RCR), sono descritte di seguito.
In genere gli algoritmi di sincronizzazione non pongono restrizioni sul tipo di tecnica da impiegare per stimare i clock remoti, ipotizzano solamente che ogni clock remoto venga stimato commettendo un errore Θ limitato; tuttavia l'errore commesso dipende dalla tecnica di stima utilizzata.

Time Transmission: nella tecnica TT, il processore p_i invia il suo clock in un messaggio. Il processore destinatario p_j usa l'informazione contenuta nel messaggio per stimare il clock di p_i. Esistono due varianti di questa tecnica, discriminate dal tipo di ritardo di propagazione del messaggio: limitato o non limitato.
La prima variante, utilizzata nei sistemi in cui il ritardo di propagazione è limitato, richiede una sincronizzazione iniziale dei clock con accuratezza β (Assunzione 1). In questa variante, p_j ad un fissato istante di tempo *T*, dove *T* è misurato con il clock logico locale, invia a tutti i processi di *P* i suoi messaggi. Contemporaneamente p_j raccoglie i messaggi inviatigli dagli altri processori, per un intervallo di tempo predeterminato, misurato sempre in riferimento al suo clock logico[1]. Il processore p_j memorizza l'istante in cui arrivano i messaggi nel suo clock estimation set. L'istante di ricezione del messaggio in arrivo dal processore p_i viene utilizzato come stima del clock di p_i. L'intervallo di tempo durante il quale p_j attende l'arrivo dei messaggi, deve essere sufficientemente grande da garantire effettivamente che p_j riceva dei messaggi.
Una versione statistica della TT può essere applicata nel caso in cui i ritardi di trasmissione non siano limitati. In questa variante, la mancanza di sincronizzazione iniziale dei clock e l'assenza di limiti sul ritardo di propagazione dei messaggi, sono compensate da un numero di *s* trasmissioni successive di messaggi di sincronizzazione. Si noti anche che questa variante presuppone la conoscenza del valore medio e della varianza dei ritardi di trasmissione.
Molti tra gli algoritmi di sincronizzazione si basano sulla tecnica TT, dato il suo basso costo, in termini di messaggi.

Remote clock reading (RCR): questa tecnica è stata introdotta per gestire il caso in cui non si conosce il limite superiore del ritardo di propagazione dei messaggi. Nella tecnica RCR il processore p_j, che intende stimare il clock di un processore remoto p_i, invia un messaggio di richiesta a p_i mettendosi in attesa della risposta per un determinato lasso di tempo. Il messaggio di risposta contiene il valore del clock logico di p_i, indicato con *T*. Il processore p_j memorizza il valore di *T* nel suo clock estimation set, questo valore sarà utilizzato per stimare il clock del processore p_i. Sia *D* la metà del round trip delay misurato dal clock di p_j. Si noti che quanto più piccolo è *D*, tanto minore sarà la lunghezza dell'intervallo di stima, e di conseguenza si avrà una migliore stima del clock remoto. Gli algoritmi di sincronizzazione dei clock probabilistici usano questa proprietà per ottenere una stima dell'errore bassa e nota. Se p_i volesse avere una stima dell'errore Θ migliore di quella ottenuta con un round trip delay di *2D*, dovrebbe inviare messaggi finché la risposta da p_j non arrivi in un tempo minore *2U* (con $U < D$), dove $U = (1-2\rho)\cdot(\Theta+\delta-\varepsilon)$.

[1] A seconda della struttura dell'algoritmo (simmetrico, asimmetrico slave-controlled o asimmetrico master-controlled) e del modello dei guasti assunto, il numero di messaggi che p_j deve attendere può variare.

La tecnica CRC ha due vantaggi: la sua correttezza non dipende dalle assunzioni fatte sulla funzione di distribuzione dei ritardi di trasmissione e, inoltre, risulta indipendente dalla sincronizzazione iniziale dei clock. Oltretutto, l'uso di questa tecnica permette ad un processore di conoscere esattamente l'errore commesso nella stima di un clock remoto e, quindi, di sapere quando ha perso la sincronizzazione con gli altri processori. Lo svantaggio, rispetto alla tecnica TT è l'uso di un numero doppio di messaggi, dovuto all'approccio request-reply.

3.1.3. Clock correction block

Il passo finale di un algoritmo di sincronizzazione dei clock è quello di correggere il clock logico in relazione al valore degli altri clock. Il clock correction block del processore p_i riceve in ingresso il clock estimation set; l'uscita del blocco non è altro che la correzione del clock logico. La regolazione del clock logico può essere fatta utilizzando le informazioni contenute nel clock estimation set oppure non utilizzandole affatto. Nel caso in cui si utilizzano le stime del clock estimation set, la correzione A_{p_i} sul processore p_i si realizza applicando una funzione di convergenza alle stime del clock estimation set (tecniche di *convergence-averaging)*; nel caso in cui non si utilizzano le informazioni del clock estimation set (tecniche di *convergence-nonaveraging*) A_{p_i} è un valore che non dipende dalle informazioni contenute nel clock estimation set.

3.2. Algoritmi di sincronizzazione interna.

Gli algoritmi di sincronizzazione interna si utilizzano quando occorre sincronizzare diversi clock fra di loro, senza che nessuno assuma il ruolo di master. Tale situazione è presente quando si opera in sistemi distribuiti equivalenti. Ad esempio, in una rete ad hoc costituita da diverse celle, occorre determinare un clock unico a partire dai diversi clock disponibili. Ciò viene ottenuto attraverso una procedura di accordo che dovrebbe portare a convergere su un clock unico.

Gli algoritmi di sincronizzazione interna devono poter operare in presenza di comportamenti difettosi dei processi e/o dei clock. E' relativamente facile, realizzare degli algoritmi di sincronizzazione fault-tolerant se si effettuano delle restrizioni sul tipo di guasto, mentre diventa più difficile determinare algoritmi che possano operare con guasti arbitrari.

I due algoritmi base messi a punto per la sincronizzazione interna sono stati messi a punto da Lamport [LSM85] e sono:

a) Interactive Convergence Algorithm (CNV);
b) Interactive Consistency Algorithm (COM e CSM).

Questi algoritmi hanno la proprietà che se un numero sufficientemente elevato processi sono non faulty allora i loro clock restano sincronizzati.
Il problema della sincronizzazione affrontato da Lamport consiste nel mantenere i clock sincronizzati una volta che lo siano stati inizialmente. Il tipo di messaggio usato dall'algoritmo di sincronizzazione del clock richiede che mittente e destinatario siano già sincronizzati. Ottenere una sincronia iniziale dipende da come sono letti i clock e dai processi di sincronizzazione tra i due utenti. Gli algoritmi di sincronizzazione descritti non si occupano di questo problema ma assumono che:

A0: *tutti i clock sono inizialmente sincronizzati approssimativamente allo stesso valore*;

A1: *il clock di un processo non faulty avanza approssimativamente con lo stesso passo del tempo reale*.

A causa delle leggere differenze dei clock rate, i valori dei singoli clock si allontaneranno lentamente tra loro; ciò richiede un reset periodico dei clock. Il clock logico di un processo è pari al clock fisico più un offset. Agendo sull'offset si mantiene la sincronia dei clock logici.
La sincronizzazione dei clock richiede che ciascun processo possa leggere non tanto il proprio clock (cosa che può fare senza problemi), ma quelli degli altri processi. I valori dei clock (intesi come velocità di avanzamento), inoltre, variano continuamente nel tempo. Questo crea problemi agli algoritmi di sincronizzazione a meno che questi non sono così veloci che la variazione dei clock non risulta essere significativa durante il periodo di risincronizzazione. Una soluzione a questo problema potrebbe essere quella di richiedere che un processo non legga il valore del clock di un altro processo, ma legga la differenza tra il suo clock e quello dell'altro. E' utile pertanto assumere che:

A2: *un processo non faulty p può leggere la differenza Δqp tra il clock di un altro processo non faulty q ed il suo con un errore, al più, di ε.*

Con questa assunzione affermiamo che ogni processo può leggere il clock di qualunque altro.

Prima di descrivere gli algoritmi, specifichiamo quali condizioni devono essere soddisfatte. Il primo requisito è:

S1: *in ogni istante i valori dei clock di tutti i processi non faulty devono essere approssimativamente uguali.*

Questa condizione è necessaria ma non sufficiente. Per esempio è soddisfatta se tutti i clock sono uguali a zero e fermi. I clock logici devono mantenersi vicini al tempo

reale. Assumiamo che i clock siano periodicamente risincronizzati e che il clock logico avanzi con lo stesso clock rate di quello fisico ad eccezione del periodo di risincronizzazione. Ciascuna risincronizzazione, quindi, causa un salto arbitrario del clock.
Una seconda condizione indica l'intervallo di incremento del clock:

S2: *esiste un limite Σ sull'ammontare di variazione del clock di un processo non faulty durante la risincronizzazione.*

Questa condizione ha due importanti conseguenze:

- se Σ è molto più piccolo del periodo di risincronizzazione, allora la risincronizzazione introduce un piccolo errore nel tasso medio di avanzamento del clock. Quindi i clock mantengono una buona approssimazione con il tempo reale assoluto;
- la risincronizzazione può causare il cambiamento del valore del clock di un processo di una quantità A. Se A >0 allora scompariranno A secondi nel clock time. Tutto ciò che doveva essere fatto in questi A secondi non può più avvenire. Se A <0 allora A secondi sono aggiunti e questo può causare dei problemi. Un modo semplice per risolvere questi problemi è di lasciare, all'inizio o alla fine del periodo di risincronizzazione, un intervallo di lunghezza Σ durante il quale il processo è *idle*. Questa soluzione è accettabile se Σ è piccolo.

3.1.1.Interactive Convergence Algorithm (CNV)

Il nome deriva dal fatto che l'algoritmo sincronizzerà correttamente i clock, ma la precisione con cui essi saranno sincronizzati dipenderà da quanto essi erano distanti prima della sincronizzazione. E' applicato ad almeno $3m+1$ processi e riesce a manipolare fino a m guasti. Opera nel seguente modo:

ciascun processo legge il valore del clock di ogni altro processo e setta il suo clock al valor medio dei valori letti, salvo che legge un valore di clock differente dal suo per più di δ, in questo caso sostituisce questo valore con il valore del suo clock per fare la media.

Per vedere perché lavora così, consideriamo due processi non faulty che differiscono dopo la risincronizzazione. Per semplicità ignoriamo l'errore di lettura del clock di altri processi e supponiamo che tutti i processi eseguano l'algoritmo allo stesso istante.
Siano p e q i processi non faulty, sia r un terzo processo e siano c_{pr} e c_{qr} i valori usati da p e q come valori del clock del processo r quando calcolano la media. Se r è non faulty allora c_{pr} e c_{qr} saranno uguali. Se r è faulty allora c_{pr} e c_{qr} saranno diversi di al

più 3δ, poiché c_{pr} è a meno di δ dal valore del clock di *p*, c_{qr} è a meno di δ dal valore del clock di *q* e il valore del clock di *p* e *q* è a meno di δ l'uno dall'altro.

Sia *n* il numero totale di processi e *m* il numero di processi faulty e assumiamo che $n>3m$. I processi *p* e *q* settano i loro clock alla media degli *n* valori c_{pr} e c_{qr}. Abbiamo che $c_{pr}= c_{qr}$ per gli $n-m$ processi non faulty e $|c_{qr}-c_{pr}|\leq 3\delta$ per gli *m* processi faulty. Possiamo dedurre allora che la media calcolata da *p* e *q* differisce al più di $(3m/n)\delta$. Avendo assunto che $n>3m$ allora $(3m/n)\delta<\delta$, quindi l'algoritmo ha successo.

Quest'ultima affermazione, giustifica l'assunzione che l'algoritmo richiede che il numero dei processi totali sia più grande di tre volte il numero di processi faulty.
Possiamo tenere i clock dei processi non faulty sincronizzati a meno di δ l'uno dall'altro effettuando di frequente la risincronizzazione di modo che, i clock che erano inizialmente a meno di $(3m/n)$ δ secondi l'uno dall'altro, non si allontaneranno per più di δ secondi.

Nella presentazione dell'algoritmo abbiamo ignorato due fattori:

1. il tempo speso per eseguire l'algoritmo;
2. l'errore nella lettura dei clock di altri processi (al più ε).

Il fatto che un processo *p* non legga tutti gli altri clock allo stesso istante comporta che non farà la media dei valori dei clock, ma userà le differenze Δqp definite in **A2**. L'errore di lettura del clock ε, definito in **A2**, significa che se δ è la massima differenza vera tra due clock allora la differenza letta dal processo *p* potrà essere δ+ε. Un valore della differenza Δqp letta dal processo *p* sarà considerato grande, e rimpiazzato da zero, se è maggiore di δ+ε. Quindi, *p*, calcolata la media dei Δpq, incrementerà il suo clock con questo valore.

3.1.2 Interactive Consistency Algorithm

Nell'algoritmo precedente, un processo setta il suo clock al valor medio di tutti i clock. Poiché un valore "non buono" di un clock può sbilanciare la media allora è necessario escluderlo. Un altro metodo che si può usare è quello di usare il mediano poiché esso fornisce un buon valore a patto che solo pochi clock non siano buoni. Il mediano calcolato da due processi sarà approssimativamente lo stesso se il set di valori dei clock è uguale. La Clock Synchronization Condition **S1** sarà valida se vale la seguente condizione per ogni processo *r*:

CC1: *dati due processi non faulty, essi ottengono approssimativamente lo stesso valore del clock di r, anche se r è faulty.*

Questa condizione, anche se assicura che tutti i processi calcoleranno lo stesso clock, non garantisce che i valori calcolati siano corretti. Per esempio, **CC1** è soddisfatta se ogni processo ottiene sempre il valore zero per ogni clock, questo però viola la Clock Synchronization Condition **S2**. Per soddisfarla è necessaria una seconda condizione:

CC2: *se r è non faulty, allora ogni processo non faulty ottiene approssimativamente il valore corretto del clock di r.*

Se la maggior parte dei processi è non faulty, allora il valore mediano del clock calcolato da ogni processo è approssimativamente uguale al valore "buono" del clock. Poiché i clock "buoni" non variano velocemente, resettando un clock al valore di un altro clock "buono", la **S2** è soddisfatta per piccoli valori di Σ. Le condizioni **CC1** e **CC2** sono simili ai requisiti per la soluzione dell'Interactive Consistency o "Byzantine Generals" problem. In questo problema, alcuni processi *r* possono spedire un valore a tutti i processi in modo tale che le seguenti due condizioni siano valide:

IC1: *tutti i processi non faulty ottengono lo stesso valore*;

IC2: *se il processo r è non faulty, allora tutti i processi ottengono il valore che r ha spedito.*

I processi privi di guasti hanno visione mutuamente consistente dei clock. La precisione con cui i clock sono sincronizzati dipende dall'accuratezza con cui i processi leggono gli altri clock e da come i clock variano durante la procedura di sincronizzazione.

Distinguiamo due tipi di algoritmi:

- COM Algorithm, che richiede almeno $3m+1$ processi per gestire fino ad m guasti;
- CSM Algorithm, che richiede l'uso di *digital signature* non modificabili per gestire fino ad m guasti con almeno $2m+1$ processi.

3.1.2.1. COM Algorithm

Questo algoritmo, indicato come COM(m), lavora in presenza di al più m processi faulty quando il numero totale di processi n è più grande di $3m$.

Consideriamo il caso in cui $n=4$, $m=1$ e descriviamo l'algoritmo COM(1).
Il processo r spedisce il valore del suo clock ad ogni altro processo che, successivamente, lo trasmette ai due processi restanti. Il processo r usa il suo stesso valore. Ciascun altro processo i, invece, ha ricevuto tre copie: una direttamente da r e due dagli altri due processi. Il valore ottenuto dal processo i è definito mediano delle tre copie. Si possono avere due casi:

- *r* è non faulty. In questo caso almeno due copie ricevute da ogni altro processo *p* devono essere uguali a quella spedita da *r* (una ricevuta da *r* e l'altra trasmessa da un qualsiasi processo non faulty poiché c'è al più un processo faulty). Il mediano di un set di tre valori, due dei quali uguali a *v*, è *v*. In questo modo è verificata la **CC1** a meno di ε. Da **CC1** segue anche **CC2** essendo, per ipotesi, *r* non faulty.
- *r* è faulty. La condizione **CC1** è inutile da verificare, dobbiamo solo provare la **CC2**. Poiché, al più, ci può essere un solo processo faulty, i tre processi oltre *r* saranno sicuramente non faulty. Ciascuno trasmette correttamente il valore ricevuto da *r* agli altri. Tutti hanno lo stesso set di copie, quindi scelgono lo stesso mediano. Allora la **CC2** è verificata.

Questo algoritmo sarà ripetuto quattro volte, una per ogni processo *r*.

L'ultima questione da risolvere è:

come fa un processo a spedire il clock ad un altro?

Il processo non spedisce il clock, ma la differenza tra clock. Il processo *p*, per esempio, spedisce una copia del clock di *q* ad *r* con un messaggio contenente il valore Δpq (cioè la differenza del clock di *q* da quello di *p* stesso). Il processo *r*, ricevuto il messaggio effettua il seguente ragionamento:

- *p* mi dice che il suo clock differisce da quello di *q* di Δpq;
- io so che il mio clock differisce da quello di *p* di Δpr;
- allora il mio clock differisce da quello di *q* di $\Delta pq+\Delta pr$;
- quindi spedisco la differenza $\Delta pq+\Delta pr$.

3.1.2.2.CSM Algorithm

Con il precedente algoritmo, abbiamo visto che una delle condizioni indispensabili era che $n>3m$. Questo limite può essere eliminato usando le *digital signature*.

Assumiamo che un processo possa generare un messaggio che può essere copiato ma non può essere identificato come alterato. Così, se *r* genera un messaggio *signed*, e copie di questo messaggio sono trasmesse da processo a processo, allora l'ultimo ricevente può dire se il messaggio ricevuto è identico all'originale spedito da *r*. Con le *digital signature*, assumiamo che un processo faulty non possa apporre la *signature* di un altro processo ad ogni messaggio non attualmente *signed* da quel processo.

Si consideri il caso $n=3$ ed $m=1$: L'algoritmo CSM opera nel seguente modo:

Il processo *r* spedisce un messaggio *signed*, contenente il valore del suo clock, agli altri due processi e ne conserva una copia. Ognuno di questi processi trasmette una copia agli altri. Ciascun processo *p* avrà quindi una pila contenente fino a due messaggi con *signature* corretta, uno ricevuto direttamente da *r* e l'altro ricevuto dal terzo processo. Il processo *p* può ricevere meno di due messaggi in quanto un processo faulty potrebbe fallire la trasmissione. Il valore del clock calcolato dal processo *p* è il più grande dei valori contenuti nella pila di messaggi con *signature* corretta. Se nessun messaggio è ricevuto allora sarà fissato un valore arbitrario.

I valori dei clock spediti da processi non faulty possono essere perturbati al più di ε e ciò rende vere le seguenti due proprietà:

CSM1: *per due processi non faulty p e q, se p ha un clock di valore c con signature corretta, allora q ha un clock con signature corretta il cui valore è, a meno di* m_ε, *c.*

CSM2: *se il processo r è non faulty e il suo clock ha valore c, allora ogni altro processo ha almeno un clock con signature corretta con valore, a meno di* ε, *pari a c, e ogni clock con signature corretta è stato letto con un ritardo di* $c+m_\varepsilon$.

Le condizioni **CC1** e **CC2** seguono immediatamente da queste due proprietà con la convenzione che approssimativamente significa a meno di m_ε .

Per concludere la descrizione di quest'algoritmo, dobbiamo dire come i clock possono essere *signed* e trasmessi, in modo tale da essere perturbati al più di ε quando trasmessi da un clock non faulty e come possono essere settati in avanti al più di ε da uno faulty.

Assumiamo che i processi e le linee trasmissive siano infinitamente veloci, in modo tale che un messaggio è trasmesso da un processo ad un altro in un tempo nullo. In questo modo ε è nullo e il clock, che il processo *r* spedisce e tutti gli altri ritrasmettono, mantiene il suo valore pari a c_r . Un processo non faulty trasmette questo valore in un tempo nullo, così il clock è spedito senza perturbazioni. Un processo faulty non può cambiare il valore del clock, ma può ritardare la trasmissione e ciò equivale a ritardare il clock al più di ε.

In pratica però i processi e le linee trasmissive non sono infinitamente veloci, quindi dobbiamo assumere un ritardo al più pari ad ε. Se includiamo il tempo necessario per generare il messaggio come parte del ritardo di trasmissione, possiamo allora dire che:

A2: *un messaggio da un processo non faulty è ricevuto a destinazione* $\gamma \pm \varepsilon$ *secondi dopo che è stato trasmesso, con* γ *costante;*

A3: *un messaggio da un processo faulty è ricevuto a destinazione almeno γ-ε secondi dopo la trasmissione.*

Questa assunzione permette l'implementazione di uno schema di lettura del clock che soddisfa **A2**: il processo *p* legge il clock del processo *q* poiché questo ha spedito un messaggio con il tempo corrente. Per calcolare Δpq, il processo *p* aggiunge γ al valore ricevuto e sottrae il valore del suo clock.

Il meccanismo delle *digital signature* prevede che ogni processo, quando ritrasmette un messaggio, aggiunga la sua *signature*. Contando il numero di *signature* nel messaggio, un processo sa quante volte esso è stato ritrasmesso e può correggere il valore del clock contenuto nel messaggio aggiungendo un appropriato γ. L'effetto è di introdurre un errore al più di ε ogni qualvolta il messaggio è trasmesso da un processo non faulty, e permette ad uno faulty di settare il clock in avanti, come richiesto, al più di ε. Ovviamente un processo può generare una signature per ogni messaggio.

3.1.3 Alcune considerazioni

Abbiamo descritto tre algoritmi di sincronizzazione del clock di tipo deterministico. L'*interactive convergence algorithm CNV* è il più semplice, infatti richiede solo che ogni processo legga il clock d'ogni altro. Gli *interactive consistency algorithm* sono più complessi in quanto richiedono una elevata quantità di messaggi trasferiti.
Le soluzioni proposte sono quelle ottime in quanto richiedono il minor numero di trasferimenti dei messaggi, infatti inizialmente saranno richiesti m+1 cicli per manipolare *m* processi faulty. Ciascun ciclo aggiunge un ε d'errore alla sincronizzazione, minimizzando il numero di cicli si minimizza l'errore.

Si può richiedere, inoltre, di minimizzare anche il numero di messaggi generati. Il COM genera circa n^{m+1} messaggi. Nelle applicazioni di controllo di processo *n* ed *m* sono piccoli, quindi il numero di messaggi è ragionevole, in altre applicazioni questo numero potrebbe salire vertiginosamente. Il CSM genera quasi lo stesso numero di messaggi del COM e può essere ridotto a $2n^2$ eliminando i messaggi ridondanti, per esempio un processo non spedirà lo stesso messaggio due volte allo stesso processo.
Nel CSM possiamo ridurre il numero di messaggi spediti facendo sì che il processo *p* soddisfi le seguenti due regole:

- *p* non spedisce un clock se già ne ha spedito uno più veloce;
- *p* non spedisce un clock se già ne ha spedito uno simile.

Quest'ultimo algoritmo è da preferire perché richiede meno processi per avere lo stesso grado di *fault tolerance*.

Nello scegliere quale algoritmo applicare, non bisogna considerare il solo parametro errore (ε), in quanto esso sarà influenzato dalla modalità di lettura dei clock, dal tempo di arrivo dei messaggi e dai ritardi di elaborazione e spedizione, ma bisogna anche valutare il numero di messaggi spediti, il numero di cicli richiesti, ecc.… per ottenere lo stesso grado di sincronizzazione.

3.2 Algoritmi di sincronizzazione esterna.

Algoritmo di Cristian

Indichiamo con Cp(t) il valore del clock (software) sulla macchina p quando il tempo UTC o comunque un tempo assunto come riferimento vale t. Facciamo l'ipotesi che tutti i clock siano inizialmente sincronizzati:

Se i clock della rete fossero idealmente esatti, si avrebbe:

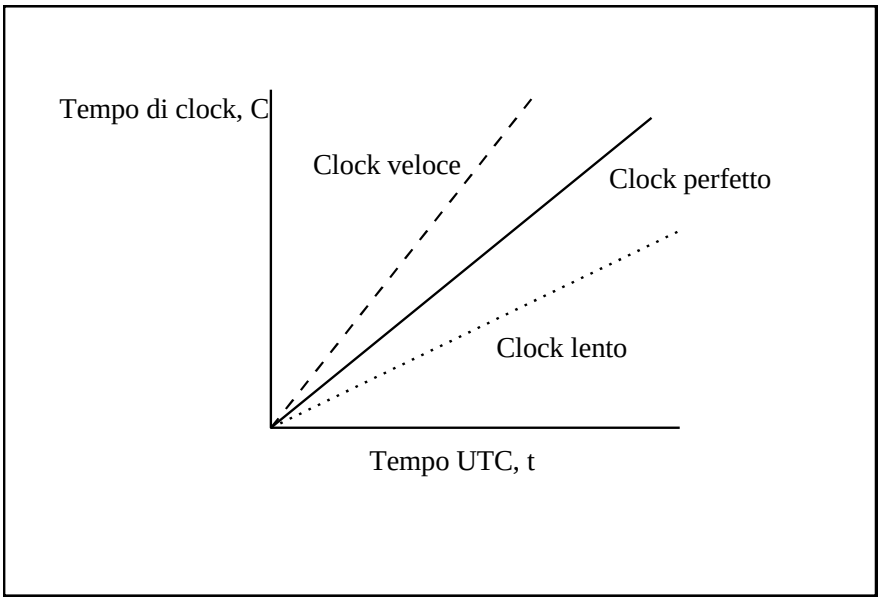
$Cp(t) = t$ per ogni p e t

ovvero:

$$\frac{dCp(t)}{dt} = 1$$

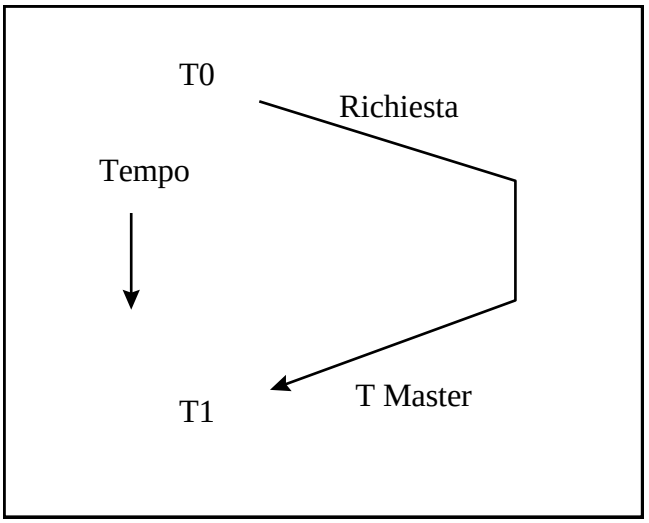
In realtà, se i timer lavorano "in accordo con la loro specifica", esiste una costante ρ tale che:

$$1-\rho \leq \frac{dC}{dt} \leq 1+\rho$$



La costante ρ viene specificata dal produttore ed è nota come ***maximum drift rate***.
Se due clock si allontanano dal tempo di riferimento in direzioni opposte, dopo un tempo dt che sono stati sincronizzati possono segnare tempi che di differenziano di una quantità che può arrivare fino a $2*\rho*dt$.
Se si vuole garantire che i clock non si discostino mai di una quantità superiore a d, essi devono quindi essere risincronizati al più ogni $d/2*\rho$.

Per ottenere ciò Cristian ha proposto la seguente soluzione:
periodicamente e certamente ad intervalli non maggiori di $d/2*\rho$, ciascuna macchina manda un messaggio al time server chiedendogli il tempo corrente e aggiusta il suo timer in accordo alla risposta ricevuta.
Poiché il time server impiega un tempo non nullo per spedire la risposta ed inoltre esiste un ritardo di trasferimento che può variare a seconda del carico della rete, il time slave dovrà misurare l'intervallo di tempo che trascorre tra la spedizione del messaggio di richiesta (T0) e la ricezione della risposta (T1) e tenerlo in considerazione per l'aggiornamento del suo clock.
Il time slave, alla ricezione della risposta, porterà pertanto il valore del suo clock ad un valore pari alla somma del tempo comunicato dal master e la sua stima del ritardo che è pari a (T1-T0)/2.



Si tenga presente che la stima dello slave non è esatta, visto che è fatta con un clock, quello dello slave, che è caratterizzato da drift. Inoltre, è estremamente difficile tenere conto dei ritardi interni che sono influenzati dalla schedulazione effettuata dal Sistema Operativo.

3.3 La sincronizzazione dei clock nelle reti wireless.

La sincronizzazione dei clock nelle reti wireless presenta alcune difficoltà aggiuntive rispetto a quelle presenti nelle reti wired.
Innanzitutto la comunicazione è meno affidabile, per cui occorre valutare l'effetto legato alla perdita di alcuni pacchetti, sulla sincronizzazione ottenuta.
Inoltre la presenza di ostacoli aleatori, derivanti dalla mobilità dei dispositivi considerati può produrre lunghi periodi di fault (legati alla impossibilità di comunicare) che può vanificare l'attività di sincronizzazione effettuata.
Resta poi da considerare l'effetto derivante, nelle reti ad hoc, dalla necessità per alcuni pacchetti di attraversare più celle con ritardi che non sempre è possibile quantificare esattamente.

Se l'obiettivo della sincronizzazione è giungere ad un clock comune molto preciso per applicazioni di misura, allora occorre tener conto del tipo di rete utilizzata e delle particolari condizioni operative. In questa prima fase presumeremo di operare nel contesto di reti ad hoc di piccole dimensioni, costituite da una sola cella. Questa condizione operativa, elimina tutti i problemi e le imprecisioni che l'uso di più celle interconnesse comporta. D'altra parte è pur vero che parecchie applicazioni di misura possono essere basate su reti di piccola dimensione, implementabili con una sola cella. Lo studio di reti ad hoc

multicelle è lasciato ad una seconda fase dello studio, dopo che alcuni degli aspetti non completamente chiari nei protocolli saranno stati superati.

I due standard che attualmente sono più utilizzati per applicazioni wireless sono: IEEE802.11 e Bluetooth.

Il primo è uno standard ormai maturo, presente sul mercato già da diversi anni che ha una vasta applicazione nella realizzazione di Wlan. Si presta alla realizzazione di reti di piccole medie dimensioni, con diversi utenti e con la possibilità di essere facilmente integrato in reti wired.

Il secondo è uno standard emergente, nuovo, non completamente specificato per alcuni aspetti, ma che tuttavia sta incontrando un interesse formidabile e sta espandendo le aree applicative. Quella che più ci interessa nel contesto di questo lavoro, e cioè la possibilità di utilizzarlo per applicazioni di misura, non era originariamente stata presa in considerazione, come si può evincere da una attenta valutazione delle specifiche e delle funzionalità incorporate. Ciò nonostante, diversi aspetti di semplicità nell'uso, gli automatismi presenti le piccole dimensioni, il basso costo, ne fanno uno strumento molto interessante per applicazioni di misura.

I due sistemi (IEEE802.11 e Bluetooth) hanno modalità operative molto diverse per la sincronizzazione dei clock che possono portare a risultati diversi. Va ribadito, che in questa fase di attività, alcuni aspetti dei protocolli non sono completamente chiari e solo l'esame e la valutazione di implementazioni reali potrà mostrare i veri limiti dei risultati possibili.

3.3.1. IEEE 802.11 CLOCK SYNCHRONIZATION

Nello standard IEEE 802.11, la sincronizzazione dei clock dei Mobile Hosts (MH) viene effettuata periodicamente dall'Access Point (AP), che assume il ruolo di coordinator. L'AP spedisce periodicamente una frame di sincronizzazione, detta *Beacon*, ogni *Beacon interval*. L'AP appone nella *Beacon Frame* il valore del suo clock. Spedisce poi la *Beacon Frame* in broadcast a tutti i MHs, i quali devono sincronizzare i loro clock locali col clock dell'AP.

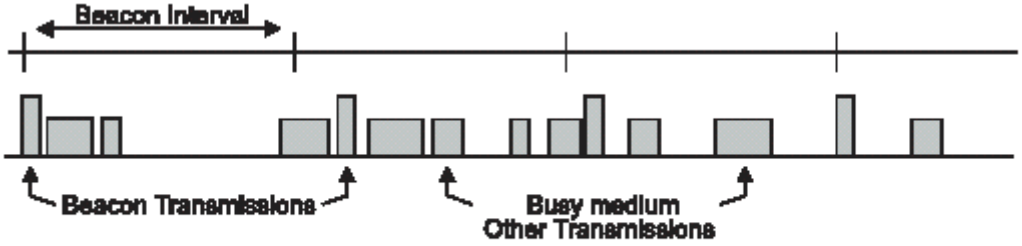


Figure 64—Beacon transmission on a busy network

Sebbene la trasmissione della *beacon* potrebbe essere ritardata, perchè il protocollo *CSMA* di accesso al mezzo (vedi paragrafo successivo) non consente la trasmissione se il canale è occupato, le *beacon* susseguenti saranno schedulate per la trasmissione sempre al *beacon interval* nominale (vedi figura). Comunque, non appena il canale si libera, la *beacon frame* viene trasmessa subito, avendo priorità superiore rispetto alle altre frame.

Quando un MH riceve la *Beacon Frame*, legge il clock dell'AP e lo aggiusta sommandogli il tempo di propagazione della frame nel canale wireless e il tempo che ha impiegato la frame a risalire dal *Physical Layer* al *MAC Layer*. Infine, attribuisce al proprio clock locale il valore cosi' ottenuto.

Poiche' tutto questo processo viene effettuato a livello di *MAC*, dopo ogni sincronizzazione il clock del MH risulta perfettamente allineato con quello dell'AP. Infatti, l'aleatorieta' e' dovuta all'incertezza della stima del tempo di propagazione della *Beacon Frame* nel canale wireless ed alla stima del tempo impiegato dalla frame per arrivare all'interfaccia *PHY-MAC*. Entrambi questi tempi sono talmente piccoli e la varianza attorno ai rispettivi valori medi talmente ridotta che la loro stima si discosta pochissimo dai valori reali e l'errore che si introduce nell'aggiustamento dei clock e' praticamente trascurabile (nell'ordine di qualche microsecondo).

Naturalmente, lo scostamento dei clock degli MHs puo' essere ridotto semplicemente operando piu' spesso il processo di sincronizzazione. Fattori determinanti nella scelta di quanto dev'essere il *Beacon Period* sono senz'altro la precisione dei clock dei MHs e la probabilita' di perdita delle frame di sincronizzazione (*fault probability*).

3.3.1.1. 802.11 Medium Access Control (MAC)

Il livello di *MAC* dell'802.11 si basa sul metodo di accesso al canale *CSMA/CA*, ovvero *Carrier Sense Multiple Access / Collision Avoidance.* Esso differisce leggermente dal *CSMA/CD (CD = Collision Detect)* utilizzato nelle reti *Ethernet*. E' simile in quanto è un metodo *CSMA*, cioè le stazioni monitorano il canale e si accorgono se c'è segnale (qualche altra stazione sta trasmettendo), evitando in questo caso di trasmettere. Differisce in quanto i canali radio non consentono l'uso del meccanismo di *Collision Detection (CD).* Allora, è essenziale evitare che si verifichino collisioni, o quantomeno ridurre al minimo la loro probabilità.

Il meccanismo di *Collision Avoidance* è illustrato dalla seguente figura:

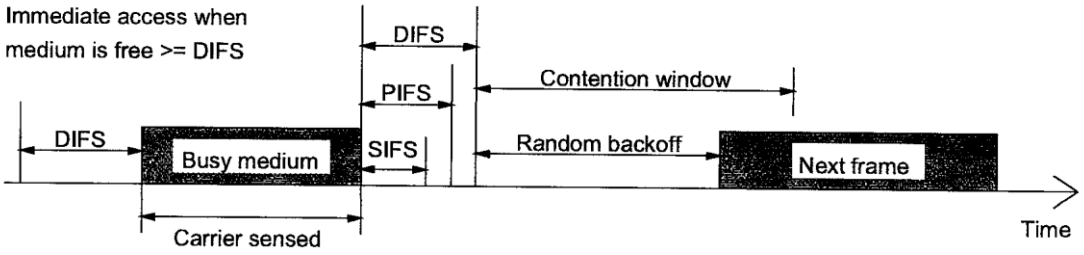


Figure 2: Primary access mechanism.

Il periodo di tempo in cui è più probabile che avvengano le collisioni è quello immediatamente seguente la trasmissione di una frame, dal momento che due o più stazioni possono essere in ascolto del canale occupato in attesa di trasmettere quando questo diventi libero.

Nel protocollo CSMA/CA dell'802.11, viene calcolato un tempo casuale di backoff uniformemente distribuito in un intervallo la cui ampiezza massima è chiamata *Congestion Window (CW).* La *CW* viene raddoppiata ogni volta che si verifica un *fault* nella trasmissione di una frame, cosa che viene rilevata dall'assenza della frame di riscontro *(ACK).* Questo *meccanismo di backoff esponenziale* aiuta a ridurre le collisioni in presenza di un numero crescente di stazioni che desiderano trasmettere in contemporanea.

In più, come mostrato nella figura di sopra, il protocollo di *MAC* prevede che le stazioni, prima di trasmettere quando il canale diventa libero, attendano un intervallo iniziale, detto *Interframe Space (IFS).* L'*IFS* può assumere tre diversi valori, che servono a gestire tre livelli di priorità. Le frame di priorità massima vengono trasmesse dopo un ***SIFS*** *(Short IFS).* L'*IFS* di lunghezza successiva, detto ***PIFS*** *(Point coordination function IFS),* viene usato per trasmettere le frame *time critical*, le quali devono essere trasmesse prima delle frame asincrone di dati. Queste ultime devono attendere un ***DIFS*** *(Distributed coordination function)* doppo che il canale diventa libero, ed in più anche un ritardo di backoff esponenziale (vedi figura).

3.3.2. BLUETOOTH CLOCK SYNCHRONIZATION

La sincronizzazione dei clock prevista dallo standard Bluetooth avviene "al volo" alla ricezione di ogni frame. Infatti, il payload contenente i dati e' preceduto da un lungo campo *Synchro* che serve a sincronizzare perfettamente il clock di tutte le stazioni della *piconet*, ovvero la cella controllata dal master. La tecnica trasmissiva prevista dal Bluetooth e' *time slotted*. Il protocollo prevede che master e slave trasmettano a slot temporali alterni. Slot adiacenti non hanno mai la stessa frequenza di trasmissione, cioè ogni cambio di slot corrisponde ad un salto di frequenza, per cui e' necessario mantenere una sincronizzazione perfetta all'interno della *piconet*.

Il funzionamento di Bluetooth per quanto riguarda la sincronizzazione lascia alcuni punti non completamente chiari. Gli slot degli slave devono essere

perfettamente sincronizzati a quelli del master perché il protocollo funzioni correttamente. A tale fine, Bluetooth mantiene il sincronismo con una precisione che può arrivare ad alcuni microsecondi (20 nel caso peggiore). Questo è un buon risultato perché l'applicazione di qualche algoritmo di sincronizzazione potrebbe ulteriormente migliorare la qualità della sincronizzazione del clock. Il principale problema è che, dalla lettura dei documenti, non appare chiaro se l'allineamento degli slit sia visibile alle applicazioni o resti confinato nel livello Baseband di Bluetooth.

Se tale allineamento è visibile dalle applicazioni, allora può essere usato per sincronizzare i vari clock e generare un unico tempo di cella. Se invece non è visibile, allora la sincronizzazione va effettuata soltanto a livello applicativo, con la presenza di notevoli errori.

In ogni caso, un altro problema e' che la sincronizzazione dei clock e' limitata alle stazioni di una stessa *piconet*. Stazioni di *piconet* diverse possono avere clock non sincronizzati, anzi ciò è auspicabile per evitare interferenze *inter-piconet*. Quindi, non esiste un clock di riferimento uguale per tutti. Inoltre, essendo la sincronizzazione implicitamente ottenuta durante la ricezione delle frame, non e' possibile nemmeno estrarre un clock utilizzabile a livello utente. Viceversa, sarebbe utile poter disporre di un *clock software*, a livello applicativo, e dunque utilizzabile dall'utente, che fosse unico per tutte le stazioni. La realizzazione di questo obiettivo non è semplice e sarà l'oggetto dello studio nella eventuale seconda parte della convenzione.

4.3 Considerazioni finali.

I risultati ottenuti attraverso i simulatori, hanno evidenziati i due aspetti che caratterizzano i due differenti sistemi IEEE802.11 e Bluetooth.

Per 802.11 il problema consiste nel fatto che le frame di sincronizzazione possono essere ritardate di tempi variabili (anche se baunded) a causa della presenza di trasmissioni precedenti in corso. Ciò introduce una aleatorietà nella sincronizzazione che ne peggiora il comportamento.

Nel caso di Bluetooth invece, il problema esiste se non è possibile accedere all'informazione relativa agli istanti di inizio degli slot. Ciò rischia di introdurre nei time stamp inseriti nelle frame di sincronizzazione, delle incertezze troppo elevate rispetto alla qualità desiderata (qualche microsecondo di tolleranza). Questo aspetto va verificato ed approfondito meglio nel futuro.

In ogni caso, in entrambi i sistemi va considerato che il clock che si vuole sincronizzare è un clock logico, che è accessibile con un certo ritardo a causa del sistema operativo e del fatto che ogni operazione effettuata via software richiede del tempo che non è sempre possibile quantificare perfettamente.

Ad esempio, l'inserimento di un Time stamp in un pacchetto comporta la lettura del tempo attuale, la scrittura del tempo nel pacchetto, il passaggio del pacchetto al livello che si occupa della sua trasmissione ed infine un tempo di attesa (che non è sempre possibile computare con accuratezza).

Pertanto, i risultati ottenibili attraverso una qualunque delle metodologie di sincronizzazione, vanno correlati alla struttura dell'Hardware, e non è da escludere che lo studio di un hardware ad hoc possa migliorare pesantemente i risultati ottenibili.

Resta poi da valutare, in una rete wireless, cosa succede nelle applicazioni multicella e capire come sincronizzare l'intero sistema.

Riferimenti bibliografici

[AP98] - E. Anceaume, I. Pouat: Performance evaluation of clock synchronization algorithms, INRIA, n.3526, Oct. 1998.

[CAS86] – F. Cristian, H. Aghilli, R. Strong: Clock sinchronization in the resence of omission and performance failures and processor joins. Proc. Of 16th symposium on fault tolerant computer systems. July 1986.

[CASD85] – F. Cristian et alii: Atomic broadcast; from simple message diffusion to byzantine agreement. Proc of 15th International Symposium on Fault tolerant Computing Systems. 1985

[CRI89] – F. Cristian: Probabilistic clock synchronization. Distributed Computing. Volume 3, pagg. 146-158. Springer Verlag.

[KO87] – H. Kopetz, W. Ochsenreiter: Cloch synchronization in distributed real-time computer systems. IEEE Transaction on computers, august 1987.

[LL84] – J. Lundelius, N. Linch: An upper and lower bound for clock synchronization. Information and Computation, 1988.

[LMS85] – L. Lamport, P. M. Mellias-Smith: Synchoniziong clocks in the presence of failures. Journal of ACM, July 1985.

[LSP82] – L. Lamport et alii: The byzantine general problems. ACM Transaction on Parallel Systems, 1985.

[SC90] – F. Schmuk, F. Cristian: Continuous clock amortization need not affect the precision of a clock synchronization algoritm. Proc. 9th international Symposium on Principles Of Distributed Computing, 1990.

[SR87] – K. G. Shin, R. Ramathan: Clock synchronization of a large multiprocessor system in the presence of malicious fault. IEEE transactions on computers, 1987.

[ST87] – T.K. Srikanth, S. Toueg: Optimal clock syncronization. Journal of the ACM, July 1987.