Università Politecnica delle Marche

Corso di Laurea Specialistica in Ingegneria Informatica

Dipartimento di Ingegneria Informatica, Gestionale e dell'Automazione

Verso sistemi di rete Real-Time: analisi delle problematiche e implementazione del protocollo TCP su S.Ha.R.K.

Relatore:

Prof. Aldo Franco Dragoni

Candidato:

Paolo Iddas

Anno accademico 2010/2011

Alla mia

famiglia

Indice generale

Sommario

In questo lavoro di tesi è stata indagata la possibilità di effettuare una comunicazione real-time su reti geografiche sfruttando l'attuale infrastruttura di rete e i protocolli maggiormente utilizzati. Sono state analizzate le problematiche che i protocolli dello stack TCP/IP pongono nei riguardi di una comunicazione real-time e sono state evidenziate alcune soluzioni proposte.

L'attenzione viene rivolta in particolare al protocollo TCP. Come prima fase tale protocollo è stato implementato sul sistema operativo in tempo reale SHaRK. A seguito dell'implementazione è stata effettuata una analisi dei meccanismi di tale protocollo per valutarne la conformità ad una serie di requisiti fondamentali per il trasferimento di informazioni in tempo reale. Viene quindi determinato un "modus operandi" alternativo che dovrebbe consentire a tale protocollo, sulla base di una infrastruttura adatta e almeno in linea teorica, di essere utilizzato come protocollo di trasporto in un sistema di rete Real-Time. Per far ciò, dove necessario, è stata proposta una estensione al protocollo per renderlo conforme ai requisiti senza tuttavia alterarne nè la sintassi nè la semantica.

Descrizione della tesi

Nel capitolo 1 viene introdotta la problematica e gli scenari coinvolti. Vengono inoltre esposte le motivazioni che spingono ad utilizzare l'infrastruttura di rete attuale e i protocolli maggiormente utilizzati.

Nel capitolo 2 vengono esposti i concetti fondamentali alla base di un sistema real-time: vengono richiamati i principi fondamentali, gli obiettivi e i vincoli di tali sistemi. Inoltre viene sommariamente descritto il sistema operativo real-time utilizzato (S.Ha.R.K).

Nel capitolo 3 viene fatta una panoramica sulle tecnologie di rete esistenti. Vengono analizzati i protocolli utilizzati nei sistemi di rete odierni a commutazione di pacchetto in particolare quelli utilizzati nelle reti geografiche (suite di protocolli dello stack TCP/IP). Inoltre vengono esposti i principali approcci pensati per supportare il traffico RT e i principi alla base della fornitura di qualità del servizio.

Nel capitolo 4 viene presentato il protocollo TCP nel dettaglio per capirne la "sintassi", la "semantica" e il "modus operandi".

Nel capitolo 5 viene descritta una prima implementazione del protocollo TCP standard sul sistema operativo S.Ha.R.K.

Il capitolo 6, sulla base dei precedenti capitoli, mostra alcune estensioni del protocollo rispetto ai requisiti di una comunicazione in tempo reale. A tale scopo viene delineata un'architettura di supporto e viene determinato un diverso modus operandi del protocollo.

Nel capitolo 7 vengono tratte alcune conclusioni e identificati gli sviluppi futuri.

Capitolo 1: Introduzione

In questo capitolo viene introdotta la problematica e gli scenari coinvolti. Vengono inoltre esposte le motivazioni che spingono ad utilizzare l'infrastruttura di rete attuale e i protocolli maggiormente utilizzati.

1.1 Una analogia

Gran parte delle attività che noi esseri umani svolgiamo ogni giorno, o che richiediamo ad altri di svolgere, sono caratterizzate da scadenze - ovvero da vincoli temporali - che ne definiscono la validità: un risultato ottenuto oltre la scadenza prefissata potrebbe diminuirne il valore o addirittura vanificarne l'utilità.

Ad esempio capita spesso di usufruire di servizi di consegna che garantiscano che un oggetto da spedire (sia esso un documento o un pacco o altro) sia recapitato secondo tempi massimi prestabiliti; la consegna oltre la data prevista potrebbe non essere accettabile.

Inoltre per essere sicuri che tutto sia andato a buon fine, potremmo desiderare di essere avvertiti dell'avvenuta consegna.

In questo caso noi stipuliamo un contratto di qualità di servizio (con vincoli temporali) in base alla qualità di servizio offerta. Non ci curiamo di come realmente il servizio sarà espletato (invio tramite

trasporto su gomma, su rotaia o via aerea) e torniamo tranquillamente alle nostre attività perchè sappiamo che qualcuno effettuerà la consegna per noi nei tempi richiesti.

1.2 Problema (esigenze da soddisfare)

I settori applicativi in cui sono richieste attività di calcolo in tempo reale sono sempre più numerosi: in tali applicazioni il requisito principale non è il throughput o la velocità di esecuzione, bensì la garanzia che siano soddisfatti stringenti requisiti temporali sulla computazione. In particolare gran parte delle applicazioni utilizzate nei sistemi di controllo, nei sistemi multimediali, nei sistemi di telecomunicazione presentano requisiti di affidabilità temporale.

Non sono rari i casi in cui due o più apparati debbono condividere e scambiare tra di loro svariate informazioni sulle azioni compiute e sull'ambiente in cui si trovano ad operare. In questi casi per garantire l'affidabilità temporale dell'intera computazione è richiesto che anche lo scambio di informazioni sia temporalmente predicibile.

Per soddisfare tale requisito sarebbe necessario predisporre di un sistema di rete real-time.

Se tali apparati sono dislocati in punti remoti del globo, per poter comunicare si deve ricorrere all'utilizzo di reti su aree geografiche ovvero ci si deve affidare ad una infrastruttura di rete complessa (in generale una rete di reti). E' opinione comune che l'affidabilità

temporale relativa alla comunicazione su aree geografiche (WAN) non può essere garantita dai sistemi di rete odierni poichè basati su un servizio di tipo best-effort.

1.2.1 Scenari

Un sistema di rete real-time è applicabile in tutti quei contesti in cui sono fondamentali le garanzie temporali sulle comunicazioni di rete. In questi contesti non è necessario soltanto garantire l'affidabilità della comunicazione ma anche la sua predicibilità temporale.

Tali necessità si riscontrano in sistemi di controllo remoto distribuito, dove un unico canale di trasmissione è utilizzato da vari sistemi di controllo che agiscono simultaneamente su varie grandezze del sistema controllato; le stesse esigenze sono ovviamente riscontrabili in presenza di un unico controllore che condivide con altri processi l'utilizzo del canale di trasmissione. Infatti un segnale di controllo ha una finestra di validità temporale che ne definisce l'effettiva utilità: superato l'ultimo istante valido, che potremmo chiamare deadline, tale segnale risulterebbe inutile se non addirittura controproducente.

Un altro contesto applicativo potrebbe essere costituito da flussi audio/video ad alta criticità che condividono la rete con altri flussi di informazione; ad esempio uno streaming video per un chirurgo che effettua un'operazione a distanza.

Un sistema di rete real-time potrebbe fornire la possibilità di

distribuire la computazione su vari host, pur mantenendo la garanzia temporale sulla computazione medesima. Un sistema di questo tipo potrebbe anche fornire un framework per lo scheduling collaborativo; ad esempio, un host che non riesca a schedulare un task potrebbe demandarlo ad un altro host della rete, eventualmente accettando una piccola penalità dovuta all'overhead della comunicazione di rete.

In generale quindi un sistema di rete real-time è necessario ogni volta che sono richieste garanzie temporali assolute sull'invio e sulla ricezione di un oggetto (sia esso un segnale di controllo, un dato, un messaggio o un flusso di informazioni).

1.3 Soluzione: reinventare la ruota o modificarla?

Per ottenere un sistema di rete real-time su vaste aree geografiche sono possibili due approcci:

- progettare una nuova infratruttura e nuovi protocolli esplicitamente pensati per una comunicazione real-time
- utilizzare l'infrastruttura e i protocolli esistenti, provando ad adattarli ai requisiti real-time

I vantaggi della prima soluzione sono certamente la possibilità di disporre di piena libertà nella definizione dei meccanismi sia a livello hardware che software concentrandosi solo sui requisiti real-time. Tuttavia si ha lo svantaggio di dover progettare, implementare e testare nuove soluzioni con una grande richiesta di energia e risorse;

una nuova infrastruttura globale di rete richiederebbe costi insostenibili.

Viceversa utilizzare l'infrastruttura odierna vincola la libertà progettuale poichè necessita di risolvere il problema senza intaccare le funzionalità oggi presenti. Si avrebbe un enorme vantaggio dal punto di vista dei costi ma tale soluzione rappresenta una sfida estrema: l'attuale rete a commutazione di pacchetto si basa su principi completamente opposti ai requisiti real-time.

Gli stessi protocolli adoperati non sembrano a prima vista essere utilizzabili in applicazioni in tempo reale. Ad esempio il protocollo TCP può dare garanzie di trasferimento affidabile dei dati ma, così come è oggi, non presenta alcun meccanismo per garantire il rispetto di alcuna tempistica richiesta.

Tuttavia avendo scartato il primo approccio per problemi di fattibilità concreta, molti ricercatori stanno tentando di trovare soluzioni che consentano di utilizzare l'infrastruttura di rete odierna per comunicazioni real-time.

1.3.1 Perchè un RTOS?

Quando si richiedono garanzie assolute sui tempi di computazione non è possibile utilizzare i "comuni" sistemi operativi time-sharing. Tali sistemi hanno come obiettivo quello di massimizzare il throughput ovvero quello di minimizzare il tempo medio di risposta di un insieme

di processi; l'obiettivo di una elaborazione real-time è invece quello di soddisfare i requisiti temporali individuali di ciascun processo. Solo sistemi operativi real-time possono garantire il rispetto di vincoli temporali richiesti da un processo.

1.3.2 Perchè il protocollo TCP?

In molti sistemi RT si sceglie di implementare lo stack UDP/IP. Essendo UDP un protocollo molto semplice, esso consente di comunicare il più velocemente possibile. Infatti UDP non introduce alcun ritardo dovuto all'impostazione di una connessione, non mantiene alcuna informazione riguardo lo stato della connessione, necessita di meno memoria e l'overhead introdotto è sensibilmente minore rispetto a TCP.

Tuttavia va ricordato che la sola velocità non dà alcuna garanzia Real-Time. UDP non può di per se stesso garantire alcunchè: tutte le funzionalità necessarie andrebbero implementate sopra di esso.

TCP invece presenta una serie di meccanismi che possono essere sfruttati in ottica RT: essendo orientato alla connessione permette a due host comunicanti di accordarsi su parametri di servizio; grazie al meccanismo dell'ack consente di valutare il ritardo di invio e ricezione sul singolo segmento; la chiusura della connessione consente di rilasciare le risorse riservate.

TCP, rispetto ad altri protocolli di trasporto, vanta di gran lunga una

maggiore diffusione (usato in oltre il 90% del traffico su Internet) e maturità. Il protocollo è ben studiato, conosciuto e analizzato nei più piccoli dettagli. E soprattutto è stato pensato per essere estendibile, grazie alla presenza del campo opzioni.

1.4 Descrizione del lavoro

Come prima fase del lavoro è stato implementato il protocollo TCP standard sul sistema operativo in tempo reale S.Ha.R.K. in modo da essere compatibile con le specifiche originali del protocollo (RFC 793 e seguenti) e in modo tale da gestire la congestione secondo il classico set di algoritmi denominati TCP Reno (RFC 2001).

In questo modo SHaRK è stato dotato di un protocollo di trasporto che garantisca l'affidabilità della comunicazione e la lealtà (fairness) nel rispetto delle altre comunicazioni.

Per verificare la correttezza del protocollo sono stati implementati varie applicazioni di test: è stato testato l'utilizzo sia come client che come server nell'interazione con altri sistemi operativi.

In seguito è stata effettuata una indagine sulla possibilità di utilizzare il TCP come protocollo di trasporto in una sistema di rete real-time. In particolare si è valutata la conformità del protocollo ad una serie di requisiti fondamentali per la computazione in tempo reale e, dove necessario, si è esteso il protocollo per renderlo conforme ai requisiti, senza tuttavia alterarne la semantica.

Capitolo 2: Sistemi operativi in tempo reale

In questo capitolo vengono vengono esposti i concetti fondamentali alla base di un sistema real-time: vengono richiamati i principi fondamentali, gli obiettivi e i vincoli di tali sistemi. Inoltre viene sommariamente descritto il sistema operativo real-time utilizzato (S.Ha.R.K).

2.1 Sistemi real-time: concetti generali

I sistemi real time (o in tempo reale) sono quei sistemi di calcolo in cui la correttezza di funzionamento non dipende soltanto dalla validità dei risultati ottenuti ma anche dal tempo in cui tali risultati sono prodotti [SR88].

La caratteristica che distingue l'elaborazione in tempo reale dagli altri tipi di elaborazione è il tempo. Nel termine tempo reale sono sintetizzati due concetti fondamentali [BUT06]:

- tempo: indica che la validità dei risultati prodotti da un processo di elaborazione non dipende solo dalla correttezza delle singole operazioni, ma anche dal tempo entro in cui i risultati sono ottenuti
- reale: indica che la risposta del sistema agli eventi esterni deve avvenire durante l'evolversi degli eventi stessi, e quindi il tempo

interno di sistema deve essere misurato secondo un riferimento temporale uguale a quello relativo all'ambiente in cui il sistema opera, ovvero il tempo di sistema e quello dell'ambiente devono scorrere alla stessa velocità

Il termine tempo reale viene spesso utilizzato impropriamente: a volte vengono definiti real time quei sistemi che, essendo estremamente veloci, riescono implicitamente a "tenere il passo" di ogni richiesta computazionale. Tuttavia la sola velocità di esecuzione non potrà mai garantire il rispetto di vincoli temporali se il sistema non è progettato opportunamente: è richiesta sempre una gestione esplicita del tempo per verificare il corretto funzionamento del sistema e sono necessari una serie di meccanismi per garantire il rispetto dei requisiti temporali individuali di ciascun processo.

Un sistema RT è strettamente legato all'ambiente in cui opera e per tale motivo la sua velocità di reazione deve essere sufficientemente allineata ai vincoli temporali imposti dall'ambiente (ciò significa che se l'ambiente evolve lentamente, il sistema potrà evolvere lentamente).

Ne deriva che un sistema real-time non deve necessariamente essere veloce; l'importante è che garantisca che ogni processo finisca la propria elaborazione entro un tempo massimo predeterminato, chiamato deadline.

In generale mentre l'obiettivo di una elaborazione veloce è quello di minimizzare il tempo medio di risposta di un insieme di processi, l'obiettivo di una elaborazione real-time è quello di soddisfare i

requisiti temporali individuali di ciascun processo [STA88].

Affinchè questo accada è necessario che il sistema sia prevedibile. Dunque la proprietà più importante che un sistema real time deve possedere non è la velocità ma la prevedibilità.

La velocità, le performance e il throughput in un sistema real time non rappresentano un requisito base ma solamente un valore aggiunto.

Processo (o task) Real-Time

Un processo real time (o task) è un compito che il sistema deve portare a termine entro un certo istante temporale. Un processo sarà sempre caratterizzato dalla presenza di alcuni parametri che permettono al sistema di valutare la fattibilità di schedulazione e quindi accettarlo o rifiutarlo.

I parametri fondamentali che consentono di prendere tali decisioni sono il tempo di computazione nel caso peggiore (WCET: Worst Case Execution Time) e l'istante entro il quale la computazione deve essere conclusa (D: deadline).

Criticità di un task

A seconda delle conseguenze provocate da una mancata deadline, si distinguono vari tipi di processi a criticità differente:

- processo hard-real-time: se la violazione della propria deadline

comporta un effetto catastrofico sul sistema (utilità dei risultati oltre la deadline è infinitamente negativa)

- processo firm-real-time: se la violazione della propria deadline rende i risultati prodotti non accettabili (utilità dei risultati oltre la deadline è nulla)
- processo soft-real-time: se la violazione della propria deadline comporta un degrado delle prestazioni del sistema senza comprometterne il suo funzionamento (utilità dei risultati oltre la deadline di tipo decrescente)

Chiaramente in un sistema real time possono essere presenti anche processi non-real-time che non essendo caratterizzati da alcuna deadline avranno la più bassa priorità e potranno essere schedulati e mandati in esecuzione nei momenti in cui la CPU risulti inoccupata.

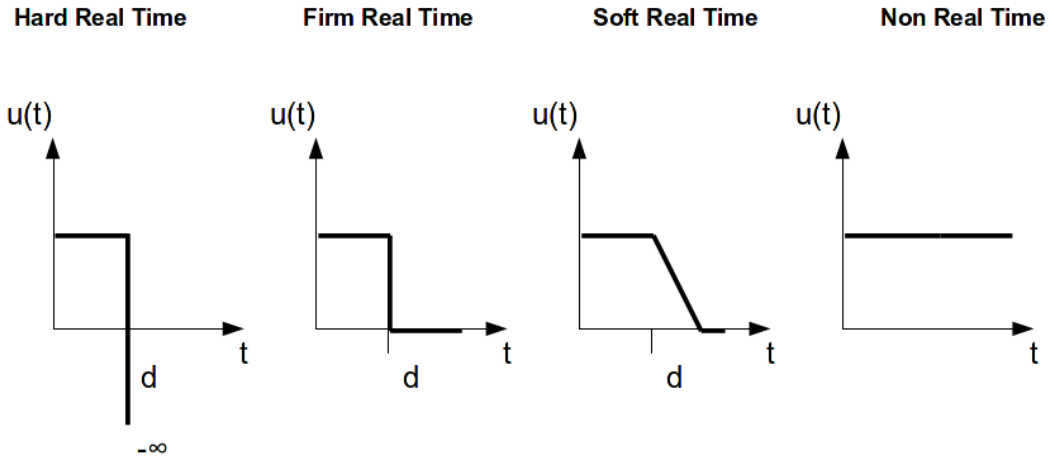

Fig. 1: criticità dei task descritti da una funzione utilità

Tipi di processi

In base alla regolarità delle attivazioni di un processo real-time, si possono distinguere i seguenti tipi di task:

- task periodici: consistono in una sequenza infinita di attività identiche, eseguite su dati diversi, dette job; tali attività sono attivate con cadenza regolare
- task aperiodici: consistono in una sequenza di attività attivate ad intervalli irregolari; sono task per i quali non è possibile stabilire a priori una periodicità
- task sporadici: sono task aperiodici in cui i job consecutivi sono separati da un minimo tempo di interarrivo

Caratteristiche desiderabili nei sistemi real time

Come visto un sistema real time ha obiettivi diversi rispetto ai sistemi time sharing e diverse sono le caratteristiche desiderabili. Esse sono:

- timeliness: i risultati devono essere corretti non solo nei valori ma anche nel dominio del tempo; il sistema deve fornire specifici meccanismi per la gestione del tempo e per l'attivazione dei task
- prevedibilità: il sistema deve essere in grado di prevedere le conseguenze delle decisioni di scheduling e valutare la fattibilità della schedulazione; tutte le primitive di sistema devono avere un tempo di esecuzione massimo definito, ossia non devono introdurre ritardi indeterminati
- tolleranza ai sovraccarichi: il sistema non deve collassare in situazioni di sovraccarico, pertanto deve essere realizzato per far fronte a tutti gli scenari possibili che possono essere previsti in fase di progetto
- monitorabilità: il sistema deve poter monitorare lo stato di esecuzione dei processi al fine di segnalare eventuali eccezioni dovute al superamento dei vincolo temporali ed intraprendere opportune azioni di recupero
- flessibilità: il sistema deve essere realizzato secondo una struttura modulare e facilmente modificabile, al fine di adattare i meccanismi di nucleo alle esigenze delle applicazioni

Prevedibilità di un sistema

Un sistema operativo in tempo reale è predicibile se è in grado di prevedere l'evoluzione dei task e di garantire in anticipo che tutti i vincoli temporali saranno soddisfatti, sulla base delle funzionalità del kernel e delle informazioni associate a ciascun task. L'affidabilità della garanzia fornita dal sistema operativo dipende da un'ampia serie di fattori che possono essere causa di comportamenti non deterministici. Tra i fattori che influiscono sulla prevedibilità di un sistema, i principali sono:

- il DMA (Direct Memory Access): migliora le performance dei dispositivi di I/O ma può rubare il bus alla CPU ritardando l'esecuzione di un task critico. In un sistema real-time si preferisce quindi disattivarlo o usarlo in modalità timeslice dove si assegna in maniera costante e fissa il bus al DMA anche se non ci sono operazioni da fare
- la cache: può causare impredicibilità poiché esistono casi in cui essa fallisce e può causare ritardi nell'accesso alla memoria da parte della CPU. Dovendo considerare quindi il caso peggiore si preferisce non usarla affatto.
- meccanismi di gestione della memoria: queste tecniche non devono introdurre ritardi non prevedibili durante l'esecuzione di task critici, ad esempio la paginazione può causare dei page fault intollerabili per un sistema hard real-time. Tipicamente si usa la segmentazione o la partizione statica della memoria.

- interrupt: sono generati da dispositivi periferici quando hanno qualche informazione da scambiare con la CPU. Quando queste interruzioni si verifcano durante l'esecuzione di un task critico generano ritardi non prevedibili ed è quindi necessario trattarle in maniera opportuna: tra le soluzione possibili solitamente si preferisce tenere gli interrupts attivati e ridurre al minimo le ISR (Interrupt Service Routine) e attivare un processo per la gestione dell'evento. Tale processo verrà schedulato come tutti gli altri processi del sistema.
- i sistemi di power management: sono meccanismi hardware che possono rallentare la CPU o far eseguire ad essa del codice utile a dissipare minor energia. È chiaro che in un sistema real-time è importante non sfondare una deadline piuttosto che consumare poca energia, quindi questi meccanismi vengono disattivati
- il meccanismo semaforico: il tipico meccanismo dei semafori utilizzato nei sistemi operativi tradizionali non è adatto all'implementazione su sistemi operativi in tempo reale, poiché è soggetto al fenomeno dell'inversione di priorità. L'inversione di priorità accade quando un task ad alta priorità è bloccato da un task a bassa priorità che fa uso di una medesima risorsa condivisa per un intervallo di tempo non limitato. Il fenomeno deve essere assolutamente evitato perché introduce ritardi non deterministici sul tempo di esecuzione di task critici ad alta priorità. Il fenomeno dell'inversione di priorità può essere

evitato utilizzando particolari protocolli di accesso alle risorse condivise.

- le primitive del kernel: le chiamate di sistema (system calls) possono influire sulla prevedibilità di risposta dei processi a seconda di come sono implementate internamente; tutte le chiamate di sistema devono essere caratterizzate da una durata massima predefinitae inoltre esse dovrebbero essere interrompibili eseguendo ad interruzioni abilitate. Ogni porzione di codice eseguita con interruzioni disabilitate infatti può causare un ritardo di attivazione di un processo critico e quindi provocare il superamento della sua deadline

2.1.1 Scheduling

Per garantire che tutti i task rispettino i propri vincoli temporali è richiesto che la schedulazione delle operazioni sia fattibile. Il concetto di fattibilità di schedulazione è alla base della teoria dei sistemi real-time ed è quello che permette di dire se un insieme di task sia eseguibile o meno in funzione dei vincoli temporali dati.

I vincoli che si possono specificare sui processi in un RTOS sono in genere di tre tipi:

- vincoli temporali: definiscono gli istanti di tempo che consentono di stabilire una schedulazione fattibile e garantire la validità del risultato

- vincoli di precedenza: definiscono le relazioni di precedenza eventualmente esistenti tra un insieme di processi; tali relazioni possono essere espresse mediante un grafo diretto e aciclico (DAG: Directed Acyclic Graph)
- vincoli su risorse: una risorsa mutuamente condivisa è una entità software che deve essere utilizzata da più processi e per la quale è necessario mantenere un ordinamento nell'accesso per mantenere la consistenza dello stato; per vincoli di risorse si intende riferirsi a vincoli di mutua esclusione su risorse condivise; rispettare un vincolo su risorse condivise significa quindi sincronizzare i processi che tentano un accesso simultaneo, al fine di consentire un uso della risorsa mutuamente esclusivo

Parametri tipici di un processo RT

Un task real-time J_i può essere caraterizzato dai seguenti parametri:

- arrival time (a_i) (anche detto release time r_i): è il tempo in cui il task è pronto per l'esecuzione
- computation time (C_i): è il tempo necessario al processore per eseguire completamente il task senza interruzioni
- relative deadline (D_i): è l'intervallo di tempo entro cui il task deve essere completato a partire dal tempo di arrivo
- start time (s_i): è l'istante di tempo in cui viene eseguita la prima

istruzione del task

- finishing time (f_i): è il tempo in cui il task termina la propria esecuzione

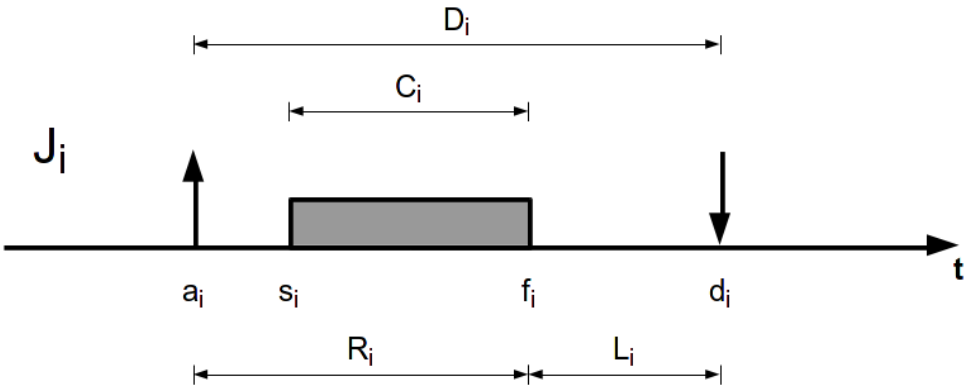

Fig. 2: parametri tipici di un job real-time

In base a tali grandezze è possibile definirne altre derivate:

- response time (R_i): è la differenza tra il finishing time ed il release time $R_i = f_i - r_i$
- lateness (L_i): rappresenta il ritardo di completamento del task rispetto alla deadline. Si noti che se un task termina prima della sua deadline, la sua lateness è negativa $L_i = f_i - d_i$
- tardiness o exceeding time (E_i): rappresenta il tempo in cui un processo è rimasto attivo oltre la propria deadline $E_i = max(0, L_i)$
- laxity o slack time (X_i): rappresenta il ritardo di attivazione

massimo che un task attivo può subire in coda pronti per non eccedere la sua deadline $X_i = d_i - a_i - C_i$

Algoritmi di schedulazione real-time

Gli algoritmi di scheduling real time possono essere distinti in:

- statici: la decisione di schedulazione è presa prima che il sistema inizi l'esecuzione dei processi. Questi metodi richiedono che le informazioni complete circa il lavoro da fare e le scadenze da rispettare siano disponibili in anticipo rispetto all'esecuzione dei processi
- dinamici: la decisione di schedulazione è presa durante l'esecuzione dei processi. Non si hanno restrizioni circa la conoscenza anticipata sui tempi di esecuzione e le scadenze da rispettare

Esistono molti algoritmi di scheduling real-time. Tipicamente si desidera l'ottimizzazione di una qualche funzione di costo definita sui parametri temporali dei processi o nei casi più critici si richiede che tutti i processi hard real time possano completare l'esecuzione entro le loro deadline.

Per valutare la bontà di un algoritmo di scheduling devono essere usate delle metriche che tengano conto dei requisiti individuali di ciascun proccesso. Ad esempio il tempo medio di risposta e il tempo di completamento totale non sono metriche utili per lo scheduling real-

time. Alcune metriche utili a tale scopo possono essere invece la lateness massima che misura la massima latenza prodotta dallo scheduling oppure il calcolo del numero dei task in ritardo.

I task periodici sono generalmente considerati di tipo hard real time, mentre i task aperiodici possono essere di tipo hard, soft o non real-time.

In generale in un RTOS l'obiettivo è garantire la schedulabilità di tutti i task classificati come hard real time. In presenza di task set ibridi (periodici e aperiodici) l'obiettivo è sempre quello di ridurre i tempi di risposta dei task aperiodici senza mettere a repentaglio la schedulabilità dei task periodici.

Anomalie di schedulazione

La prevedibilità di un sistema real-time o il rispetto delle deadline non può essere ottenuta solo mediante architetture più veloci. Dimostrazione di questo fatto si trovano in letteratura [GRA76] dove sono mostrati casi di schedulazione anomale.

Il primo caso mostra come il tempo totale di completamento di un task può aumentare se l'insieme viene eseguito su una macchina più potente (con un processore in più).

Il secondo caso illustra un esempio in cui il tempo di completamento totale può aumentare se si diminuiscono i tempi di calcolo dei singoli

processi (processore a frequenza più elevata).

Il terzo caso mostra un peggioramento delle prestazioni causato da un rilascio dei vincoli di precedenza.

2.2 Sistema usato (S.Ha.R.K.)

Il sistema utilizzato è il sistema operativo Real Time denominato S.Ha.R.K. (o SHaRK: Soft and Hard Real-time Kernel), sviluppato presso la Scuola Superiore Sant'Anna di Pisa. SHaRK è un kernel didattico progettato per l'implementazione e il testing di nuovi algoritmi di scheduling e protocolli di gestione delle risorse. Si tratta di un kernel snello e modulare in grado di gestire processi hard e soft real-time ma anche applicazioni non in tempo reale.

Può essere visto come una libreria di funzioni che estende la classica libreria C per fornire un ambiente per la multiprogrammazione con una esplicita gestione del tempo. SHaRK supporta l'architettura Intel x86 (single core) ed è aderente allo standard POSIX 1003.13 PSE52 [POS03] (realtime controller and system profile), permettendo così il porting veloce delle applicazioni scritte per altri OS aderenti a tali standard (come ad es. Linux). Alcuni driver di dispositivi sono supportati in maniera nativa, molti altri derivano da Linux grazie ad un layer di compatilità (glue code).

Gli obiettivi di tale sistema sono:

- semplicità dello sviluppo

- flessibilità nelle modifiche delle politiche di scheduling
- predicibilità
- aderenza allo standard POSIX

Il kernel è progettato per essere altamente modulare, in modo che i meccanismi di scheduling e di gestione delle risorse possano essere sviluppati indipendentemente dagli altri meccanismi di sistema e quindi possano essere facilmente modificati e sostituiti. Inoltre l'architettura di scheduling è tale che una stessa applicazione possa essere eseguita con diversi meccanismi del kernel al fine di testare l'incidenza di un algoritmo sulle prestazioni del sistema.

2.2.1 Cenni sull'architettura del kernel

Al fine di rendere l'applicazione indipendente dai meccanismi interni del kernel, SHaRK utilizza il concetto di generic kernel, il quale non implementa alcuna particolare strategia di scheduling, ma postpone le decisioni di scheduling demandandole ad entità esterne (scheduling modules). Analogamente l'accesso alle risorse condivise è coordinato da moduli esterni (resource modules).

L'indipendenza tra applicazioni e algoritmi di scheduling è ottenuta utilizzando il concetto di modello. Sono forniti due tipi di modelli: Task Models e Resource Models. Un modello di task specifica i requisiti di qualità di servizio (QoS) richiesti dal task allo schedulatore. Un modello di risorsa specifica i parametri di QoS

relativi ad un insieme di risorse condivise utilizzate da un task. I modelli vengono utilizzati dal Generic Kernel per assegnare un task ad un modulo di scheduling. Un componente del kernel, chiamato Model Mapper, consente di mappare il modello di QoS del task ad un modulo registrato nel kernel, selezionato in base ad una strategia interna.

SHaRK fornisce supporto ai principali algoritmi di scheduling real-time (EDF, RM, ...), ai server per task periodici e aperiodici (CBS, TBS, ...) e ai principali protocollo per l'accesso a risorse condivise (PIP, PCP, SRP, ...).

Capitolo 3: Sistemi di rete e il Real-Time

In questo capitolo viene fatta una panoramica sulle tecnologie di rete esistenti. Vengono analizzati i protocolli utilizzati nei sistemi di rete odierni a commutazione di pacchetto in particolare quelli utilizzati nelle reti geografiche (suite di protocolli dello stack TCP/IP). Inoltre vengono esposti i principali approcci pensati per supportare il traffico RT e i principi alla base della fornitura di qualità del servizio.

3.1 Architettura logica di rete

Per lo studio e l'analisi delle architetture di rete possono essere presi come riferimento sia il modello ISO/OSI che il modello TCP/IP. Il modello ISO/OSI è un modello generico concepito per le reti di telecomunicazione a commutazione di pacchetto. Esso è costituito da una pila (o stack) di protocolli attraverso i quali viene ridotta la complessità implementativa di un sistema di comunicazione per il networking. In particolare ISO/OSI è costituito da strati (o livelli), i cosiddetti layer, che racchiudono uno o più aspetti fra loro correlati della comunicazione fra due nodi di una rete. I layers sono in totale 7 e vanno dal livello fisico fino al livello delle applicazioni, attraverso cui si realizza la comunicazione di alto livello.

Il modello TCP/IP può essere visto come una realizzazione del modello ISO/OSI.

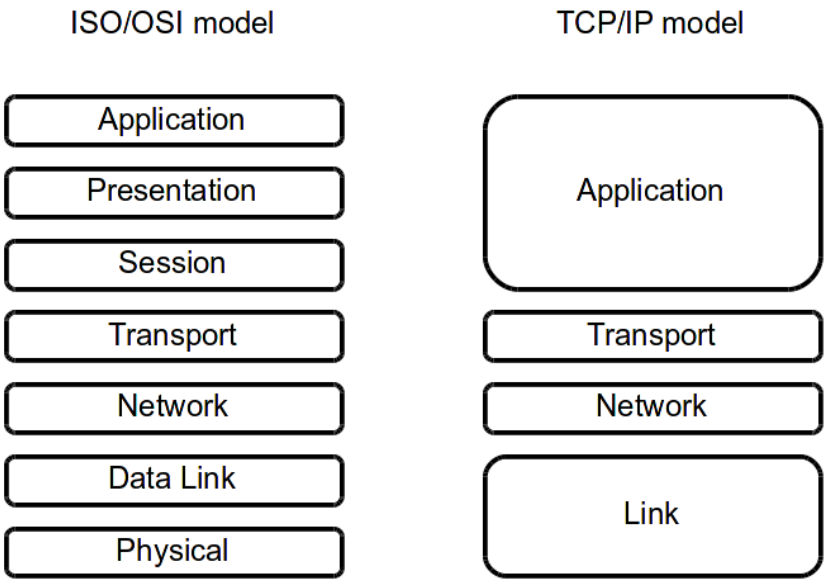

Fig. 3: modello ISO/OSI e modello suite TCP/IP

La suite di protocolli TCP/IP permette a dispositivi di qualunque dimensione, con qualsiasi hardware e con qualsiasi sistema operativo di comunicare tra loro. Tale flessibilità è stata raggiunta grazie alla suddivisione in strati (layer): nella pila protocollare ogni strato ha differenti responsabilità e fornisce servizi agli strati superiori utilizzando i servizi offerti dagli strati inferiori.

Ogni layer ha le seguenti responsabilità:

- lo strato di collegamento (link layer) include i driver di dispositivo del sistema operativo maneggiando tutti i dettagli hardware relativi alle schede di rete. Inoltre gestisce i protocolli di più basso livello per la comunicazione su rete locale.

- lo strato di rete (network layer) gestisce il movimento dei pacchetti nella rete e si occupa principalmente del loro instradamento. Il protocollo principale a questo livello è IP.
- lo strato di trasporto (transport layer) fornisce alle applicazioni i servizi per l'invio del flusso di dati tra due differenti host. In questo strato troviamo i protocolli UDP (User Datagram Protocol) e TCP (Transmission Control Protocol)
- lo strato applicazione (application layer) gestisce i dettagli relativi alle sole logiche della applicazione e grazie ai servizi offerti dagli strati sottostanti può così ignorare tutti i dettagli relativi alla comunicazione

Per realizzare il processo di stratificazione e mantenere separati i vari layers vengono utilizzate tecniche di multiplexing e demultiplexing.

Il multiplexing avviene quando i dati scendono lungo lo stack dei protocolli e viene realizzato con la tecnica dell'encapsulation: ogni strato aggiunge informazioni ai dati pre-apponendo il proprio header (e a volte post-ponendo un proprio trailer) al pacchetto dati.

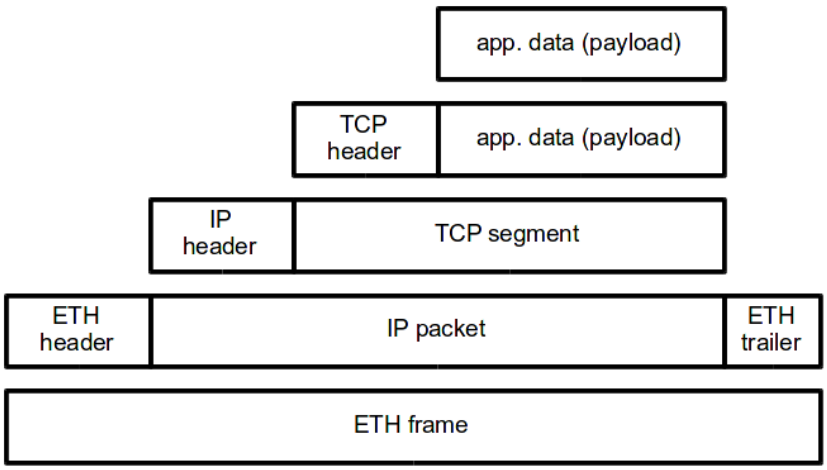

Fig. 4: esempio multiplexing tramite encapsulation

Il demultiplexing avviene quando viene ricevuto un frame dalla rete: risalendo lungo la pila protocollare ogni strato fa le scelte basandosi sulle informazioni presenti nell'header del suo livello.

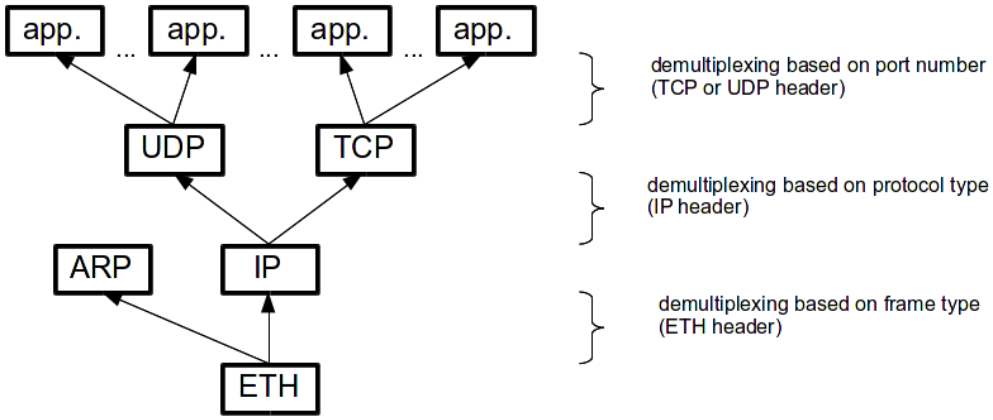

Fig. 5: esempio demultiplexing

3.2 Panoramica sulle tecnologie di rete

3.2.1 Strato di collegamento

Il servizio base di uno strato di collegamento è quello di muovere una certa quantità di dati (datagram o frame) da un nodo a un nodo adiacente su un singolo link di comunicazione. Altri servizi che possono essere offerti sono framing, accesso al link, recapito affidabile, controllo di flusso, ricerca di errori, correzione degli errori, trasmissione half-duplex o full-duplex.

Nello strato di collegamento possono essere individuati due tipi di canale:

- broadcast: (comuni nella reti a area locale) molti terminali sono connessi a uno stesso canale di comunicazione ed è quindi necessario un cosiddetto protocollo di accesso al mezzo (MAC: Media Access Control) per coordinare le trasmissioni ed evitare le collisioni
- link da punto a punto: essendo il canale non condiviso con altri apparati, la coordinazione per l'accesso al link è banale

Tutti i link di tipo broadcast hanno un canale condiviso: nasce così il problema di coordinare la spedizione e la ricezione di più nodi, ovvero il problema dell'accesso multiplo. Nel seguito si indicherà con N il

numero di nodi e con R la velocità di trasmissione del canale (banda) in bit/s.

Possiamo classificare praticamente tutti i protocolli ad accesso multiplo all'interno di una delle seguenti categorie:

- protocolli a suddivisione del canale (channel partitioning protocols): il canale viene suddiviso tra i nodi
    - TDMA (Time Division Multiplexing Access): divide il tempo in intervalli e poi divide ciascun intervallo in N blocchi di tempo (slot); ciascuno degli N slot è assegnato ad uno degli N nodi. Ogni volta che un nodo ha un frame da spedire esso trasmette i bit del frame durante lo slot di tempo a esso assegnato nel frame TDM a rotazione
    - FDMA (Frquency Division Multiplexing Access): divide il canale con banda R in differenti frequenze ciascuna con larghezza di banda R/N e assegna ciascuna frequenza a uno degli N nodi
    - CDMA (Code Division Multiple Access): assegna un codice diverso a ciascun nodo e grazie ad opportuni meccanismi consente a diversi nodi di trasmettere simultaneamente con i rispettivi receiver; in tal modo si ottiene un utilizzo migliore della banda

  Vantaggi di questi protocolli sono l'equità, l'assenza di collisioni, la determinabilità del periodo di trasmissione (determinismo temporale).

Svantaggi sono la scarsa utilizzazione della risorsa (banda disponibile), bassa scalabilità (infatti la banda dipende dal numero di nodi connessi al canale e vale esattamente R/N).

- protocolli di accesso casuale (random access protocol): un nodo trasmittente invia sempre alla massima velocità del canale; quando si verifica una collisione ciascun nodo coinvolto ritrasmette il suo frame, aspettando per un ritardo casuale (random delay), fino a quando questo passa senza collisioni
  - ALOHA: quando un nodo ha dati da trasmettere, li trasmette. Se ci sono collisioni i frame corrotti vengono distrutti; in questo caso il nodo mittente reinvia il frame dopo un'attesa casuale e si rimette in ascolto sul canale (o attende un ack) fino a quando non stabilisce che il frame è stato ricevuto correttamente
  - slotted ALOHA: il tempo è suddiviso in intervalli discreti chiamati slot; ogni stazione è vincolata a cominciare la propria trasmissione necessariamente all'inizio di uno slot temporale. La conseguenza di tale caratteristica è che due trasmissioni o collidono completamente all'interno dello stesso slot oppure non collidono affatto
  - CSMA (Carrier Sense Multiple Access): accesso multiplo a rilevazione della portante, dove per rilevazione della portante si intende il fatto che un nodo ascolta il canale verificando che esso sia libero prima di trasmettere. Se il canale è occupato il nodo attende un tempo casuale prima di

rimettersi in ascolto del canale

- CSMA/CD (Carrier Sense Multiple Access with Collision Detection): protocollo CSMA con rilevazione delle collisioni (es. Ethernet)

Tali protocolli consentono di raggiungere un'alta efficienza nel caso migliore (no collisioni), tuttavia questi protocolli non sono deterministici e l'efficienza dipende dal numero di nodi collegati.

- protocolli a turni (taking turns protocol)
  - polling protocol: richiede che uno dei nodi sia designato come nodo master il quale sonda a rotazione ciascuno dei nodi chiedendo se esso deve trasmettere.
  - token passing protocol (es. FDDI): un piccolo frame detto token viene scambiato a rotazione prefissata tra i vari nodi; il token viene trattenuto se il nodo deve trasmettere, altrimenti lo inoltra al nodo successivo.

Tali protocolli eliminano le collisioni, permettono di avere una alta efficienza e consentono di definire il tempo massimo del completamento della trasmissione (determinismo temporale).

Tuttavia sono protocolli a single point of failure: nel polling protocol se il nodo master si guasta il canale diventa inoperativo, mentre nel token passing protocol il guasto di un qualunque nodo può mettere fuori servizio l'intero canale.

3.2.2 Strato di rete

Il servizio base dello strato di rete è quello di trasportare i pacchetti da un host che spedisce a un host che riceve. Le funzioni fondamentali sono:

- instradamento (routing): individua un percorso per andare da un host A ad un host B
- commutazione (forwarding): determina il link di uscita per ogni pacchetto entrante nel router

Tra gli altri servizi vi può essere quello dell'instaurazione della chiamata che consente di impostare lo stato dei router tra sorgente e destinazione prima che i dati inizino a fluire; in sostanza può essere impostato un circuito virtuale tra sorgente e destinazione (tale servizio non è tutt'oggi presente nel protocollo IP).

Circuito virtuale vs reti datagram

Alcune tecnologie fanno uso di circuiti virtuali (Virtual Circuit, VC) per riservare risorse lungo il percorso (ATM, frame relay, X.25): possono essere visti come orientati alla connessione.

Le reti datagram (come IP) non mantengono alcuna informazione di stato relativa ad alcun circuito: è un servizio non orientato alla connessione. Dal servizio datagram deriva il concetto di servizio best effort.

Algoritmi di routing

Esistono differenti algoritmi di routing. In generale la loro complessità tende a crescere con l'aumentare delle dimensioni della rete; questo problema viene affrontato per mezzo del concetto di routing gerarchico.

Gli algoritmi di routing si possono classificare come:

- globali, quando hanno a disposizione informazioni complete sullo stato e sui costi dei link
- decentralizzati, quando nessun nodo della rete ha a sua disposizione informazioni complete sul costo di tutti i link della rete
- statici, quando la topologia del grafo della rete, ed i costi associati ai link, cambiano molto lentamente in relazione al tempo
- dinamici, quando variano il routing in relazione ad una variazione nella rete

Gli algoritmi di routing possono essere divisi in due classi principali, in base alle informazioni disponibili come input:

- link state: è un algoritmo globale e dinamico basato sull'algoritmo di Dijkstra; ogni nodo ha una conoscenza globale della rete di riferimento. L'algoritmo è di tipo iterativo e consente di definire tutti i costi per tutti i possibili percorsi
- distance vector: è un algoritmo di routing decentralizzato e

dinamico; ogni nodo ha a disposizione soltanto le informazioni riguardo ai nodi ai quali è direttamente collegato.

3.2.3 Strato di trasporto

Il servizio base dello strato di trasporto è fornire una comunicazione logica fra i processi applicativi che funzionano su host differenti. Dal punto di vista dell'applicazione è come se i terminali su cui girano i processi fossero direttamente connessi. I processi applicativi usano la comunicazione logica fornita dallo strato di trasporto per scambiarsi messaggi, senza doversi preoccupare dei dettagli dell'infrastruttura fisica usata per trasportare quei messaggi. E' importante ricordare che i protocolli dello strato di trasporto sono implementati nei terminali ma non nei router della rete.

Compito di ogni protocollo di questo strato è quello di dare la possibilità di comunicare a tutte le applicazioni che ne fanno richiesta: per tale motivo è necessario un servizio di multiplexing e demultiplexing che consenta di estendere il servizio di consegna da terminale a terminale in un servizio di consegna da processo a processo per le applicazioni che girano sui terminali.

In generale si può fare una suddivisione tra protocolli senza connessione e protocollo orientati alla connessione

- trasporto senza connessione (es. UDP): non viene impostata alcuna connessione quindi l'applicazione può trasmettere in

qualsiasi momento; per tale motivo non è necessario mantenere alcuno stato della connessione.

Tale servizio è molto semplice e consente di raggiungere buoni throughput tuttavia tutti i servizi aggiuntivi riguardanti il trasferimento dati sono demandate a protocolli di livello superiore o alle applicazioni

- trasporto orientato alla connessione (es. TCP): prima che un processo applicativo possa trasmettere dati, i due processi devono scambiarsi dei segmenti per poter stabilire i parametri del successivo trasferimento dati. In tal modo entrambi i lati della connessione dovranno inizializzare molte variabili di stato che saranno aggiornate durante la fase di scambio dati. Avendo lo stato della connessione è possibile fornire tutta una serie di servizi come ad esempio il trasferimento affidabile, la consegna ordinata, il controllo di flusso ecc...

3.3 Servizio best-effort

L'attuale infrastruttura di rete a commutazione di pacchetto è basata sul servizio best-effort. La consegna best effort descrive uno scenario di rete nel quale non è fornita nessuna garanzia sulla consegna dei dati o sul livello di QoS, non c'è garanzia sui tempi massimi di consegna dei pacchetti, non c'è garanzia che i pacchetti siano ricevuti nell'ordine in cui sono stati spediti ma tutte le comunicazioni

avvengono con il massimo impegno possibile (best effort, per l'appunto). Le conseguenze di questa politica di consegna sono bitrate e tempi di consegna variabili a seconda del traffico che deve gestire la rete. Il senso che sta dietro a questa filosofia è che, rimuovendo alcune caratteristiche che consentono il recupero di pacchetti persi o corrotti, e di preallocazione delle risorse, la struttura della rete risulta semplificata e opera più efficientemente. Secondo tale paradigma tutti i meccanismi per la gestione della comunicazione devono risiedere solo sugli host terminali.

I router IP convenzionali forniscono solamente servizi di tipo best-effort. La semplicità dei router con questa tecnologia rappresenta uno dei fattori chiave che ha consentito ad IP di avere molto più successo di protocolli più complessi, quali X.25 e ATM. In X.25 è implementato infatti un meccanismo di controllo di errore ad ogni hop, che prevede che i dati venivano ritrasmessi con un protocollo di tipo ARQ. In una rete ATM, invece, vengono garantiti dei livelli minimi di servizio, in termini di ritardo o di larghezza di banda.

Tuttavia, i moderni router IP forniscono meccanismi per differenziare la qualità del servizio fra i diversi flussi di traffico, o per garantire determinati livelli di QoS a determinati flussi. Questi meccanismi possono essere utilizzati in reti con capacità limitata, allo scopo di riservare le risorse necessarie ad alcune applicazioni sensibili al ritardo (delay sensitive), o che necessitano di un bitrate costante.

3.4 Tecnologie di rete e Real-Time

Un sistema di rete real-time è un sistema in cui il corretto funzionamento non dipende soltanto dalla affidabilità della comunicazione, ma anche dall'istante temporale in cui essa è portata a termine. Quindi i requisiti fondamentali che sono richiesti sono l'affidabilità e la garanzia temporale. Come visto in precedenza, le garanzie temporali richiedono predicibilità ovvero determinismo temporale.

Il principale problema del servizio best-effort nei confronti del real-time riguarda il jitter dei pacchetti: l'estrema variabilità dei pacchetti all'interno dello stesso flusso non permette di dare alcuna garanzia temporale alle applicazioni.

Un altro problema riguarda l'impossibilità di stabilire a priori la larghezza di banda disponibile per la comunicazione e il fatto che durante la stessa sessione di trasferimento dati la banda può variare sensibilmente.

Considerazioni sul ritardo dei nodi

In una rete a commutazione di pacchetto, un pacchetto parte da un host (sorgente), passa attraverso una serie di router e termina il suo viaggio in un altro host (destinatario). Quando un pacchetto passa da un nodo al nodo seguente, in ciascun nodo lungo il percorso soffre di

diversi ritardi. I più importanti di questi ritardi sono:

- il ritardo di elaborazione del nodo: è composto dal tempo richiesto per esaminare l'intestazione del pacchetto e instradarlo; ad esso si somma il tempo necesssario per controllare eventuali errori a livello di bit (il ritardo totale è dell'ordine dei microsecondi o inferiore)
- il ritardo di coda: dipende dal numero dei pacchetti che sono accodati nel nodo e aspettano di essere trasmessi attraverso il link; tale ritardo può essere molto variabile (il ritardo totale può essere dell'ordine dai millisecondi ai microsecondi e nel caso migliore vale zero)
- il ritardo di trasmissione: (detto anche ritardo store and forward) vale L/R dove L è la lunghezza del pacchetto e R è la velocità di trasmissione del link verso il nodo successivo (il ritardo totale può essere dell'ordine dai millisecondi ai microsecondi)
- il ritardo di propagazione: ritardo dovuto alla propagazione dell'unità di informazione (bit) sul link; la velocità di propagazione dipende dal mezzo trasmissivo che costituisce il link ed equivale, o è poco inferiore, alla velocità della luce; il ritardo dipende quindi dalla distanza tra il nodo e il successivo

Il ritardo totale del singolo nodo è dato da:

$$d_{\text{nodo}} = d_{\text{elab}} + d_{\text{coda}} + d_{\text{tras}} + d_{\text{prop}}$$

Bisogna notare che le quantità d_{tras} e d_{prop} sono deterministiche

mentre in generale d_{elab} e soprattutto d_{coda} possono variare considerevolmente.

Per avere una stima del WCTT (Worst Case (Network) Traversal Time) è necessario che ogni nodo del percorso possa stabilire un ritardo di nodo nel caso peggiore.

Scuole di pensiero

Nel corso degli anni si sono sviluppate diverse scuole di pensiero e visioni differenti su come possa essere offerto un servizio Real-Time in una rete a commutazione di pacchetto:

1. non interventista: sostiene che non siano necessarie modifiche sostanziali al servizio best effort in ottica RT: il continuo aumento di banda e miglioramento dell'infratruttura di rete e il conseguente aumento di velocità e delle capacità di bufferizzazione consentirà di utilizzare l'attuale rete per il traffico real-time evitando lo sforamento delle deadline
2. servizio con prenotazione: sono necessari radicali cambiamenti all'infrastruttura e ai protocolli di rete; si deve poter esplicitamente prenotare la larghezza di banda di ciascun link da A a B.
3. servizio differenziato: propone cambiamenti relativamente piccoli agli strati di rete e trasporto e semplici schemi di tariffazione e controllo solo alle estremità della rete; l'idea è quella di introdurre un piccolo numero di classi di servizio per

fornire livelli di servizio differenti in base alla classe

Il primo approccio va scartato in ottica Hard Real Time: bisogna infatti ricordare che non può essere la sola velocità a garantire il rispetto delle deadline in un sistema Real-Time (cfr. con le anomalie di schedulazione riportate nel precedente capitolo)

Il servizio con prenotazione richiede grandi cambiamenti: occorre un protocollo che prenoti la larghezza di banda dai sender ai receiver; tutti i nodi della rete devono modificare le politiche di gestione delle code nei router: servono meccanismi per classificare il traffico e per sorvegliare che ciascun flusso corrisponda alla descrizione.

L'approccio differenziato invece ritiene che attraverso l'aggiunta di alcuni elementi funzionali soprattutto alle estremità della rete, sia possibile garantire servizi quali ad esempio l'inoltro spedito o l'inoltro assicurato.

Principi generali per una comunicazione RT

Da quanto detto in precedenza, un sistema di rete Real-Time richiede di andare oltre il best-effort. Possono essere individuati i seguenti principi necessari per una comunicazine che sia a tutti gli effetti predicibile:

- differenziazione dei tipi di traffico: analogamente a quanto accade negli RTOS, dove si differenziano i task real-time da quelli non real-time, dobbiamo distinguere il traffico real-time dal traffico best-effort che potrebbe continuare ad utilizzare la

rete. Questo comporta una distinzione dei pacchetti real-time da quelli best-effort; tale distinzione potrebbe essere effettuata a livello IP (bisogna notare che a tale scopo esiste il campo Type of Service in IPv4 o Traffic Class in IPv6). Nulla vieta di effettuare questa distinzione anche a livelli superiore, laddove si rendesse necessario

- isolamento dei tipi di traffico: analogamente a quanto accade negli RTOS (principio di task isolation), il traffico real-time deve essere isolato da quello non real-time. Questo significa che il traffico non-real-time non può ostacolare quello real-time, e impedire il soddisfacimento dei vincoli per quel tipo di traffico. Inoltre varie trasmissioni real-time devono essere isolate l'una rispetto all'altra, vale a dire che il fallimento di una di esse non deve provocare il fallimento delle altre
- uso efficiente delle risorse: negli RTOS la CPU non può mai essere idle se un task è pronto per l'esecuzione. Analogamente, un flusso dati deve poter accedere al canale se esso è disponibile per trasmettere
- ammissione della chiamata: negli RTOS ogni task esplicita la richiesta di QoS sulla CPU. Analogamente ogni applicazione dovrebbe specificare la richiesta di QoS con cui vuole essere servita dalla rete. La rete valuterà se la richiesta è ammissibile e, in caso affermativo, provvederà a soddisfarla. In caso negativo l'applicazione o la rete potrebbero prevedere oppurtuni meccanismi di error-recovery o di rinegoziazione del contratto di

QoS

Meccanismi di scheduling e policing

Per fornire garanzie sulle richieste di QoS sono necessari alcuni meccanismi di base da adottare nei vari nodi della rete.

Sono necessari meccanismi di scheduling in grado di diversificare il comportamento in base al tipo di traffico e meccanismi di policing per sorvegliare il comportamento corretto di un certo flusso dati.

Un meccanismo di scheduling che ha trovato un impiego considerevole nelle architetture per la QoS è la cosiddetta modalità di accodamento equo pesato (WFQ: Weighted Fair Queueing): i pacchetti in arrivo ad un nodo sono classificati e vengono accodati nella loro coda di attesa della classe appropriata. Ogni classe può ricevere una quantità di servizio differenziata in qualsiasi intervallo di tempo. A ciascuna classe i è assegnato un peso w_i. Alla classe i viene garantito di ricevere una frazione di servizio pari a $w_i / \sum w_j$ dove la somma al denominatore è effettuata su tutte le classi. Quindi per un link con velocità di trasmissione R, la classe i raggiungerà sempre un throughput di $R \cdot w_i / \sum w_j$.

Un meccanismo di policing per caratterizzare i limiti di sorveglianza è il leaky bucket (secchio bucato). Un leaky bucket è un contenitore che può contenere fino a b token (gettoni). I token sono aggiunti al contenitore sempre alla velocità di r token al secondo fino ad un massimo determinato dalla capacità del contenitore. Ogni pacchetto,

per essere inviato nella rete deve rimuovere un token dal contenitore. Se tale contenitore è vuoto, il pacchetto dovrà aspettare un token.

Quindi la velocità r di generazione dei token serve a limitare la velocità media a lungo termine a cui i pacchetti possono entrare nella rete.

Si può dimostrare che l'utilizzo combinato di leaky bucket e WFQ consente di fornire un ritardo massimo di attesa in coda che vale:

$$d_{coda,\,max} = \frac{b_i}{R \cdot \frac{w_i}{\sum w_j}}$$

3.4.1 Servizi integrati

Intserv è un progetto sviluppare all'interno della IETF per fornire garanzie di QoS personalizzate a sessioni applicative individuali. Le caratteristiche basilari che costituiscono il nucleo dell'architettura sono:

- risorse riservate: ad un nodo della rete è richiesto di conoscere la quantità delle sue risorse (buffer, larghezza di banda del link) che sono già state riservate ad altre sessioni
- impostazione della chiamata: una sessione che richiede garanzie di QoS deve prima essere in grado di prenotare risorse sufficienti a ciascun nodo della rete sul suo percorso da sorgente a destinazione per assicurare che i suoi requisiti di QoS siano

soddisfatti, senza violare le garanzie date alle altre sessioni ammesse pprecedentemente

Per impostare la chiamata si effettueranno i seguenti passi:

1. caratterizzazione del traffico e specifica della QoS desiderata: la sessione deve dichiarare la sua QoS e caratterizzare il traffico che invierà nella rete. Nell'architettura IntServ il cosiddetto Rspec (Reserved specification) definisce la specifica QoS richiesta da una connessione, mentre il cosiddetto Tspec (Traffic specification) caratterizza il traffico che il sender si appresta ad inviare nella rete. La forma specifica di Tspec e Rspec sono definiti in [RFC2210] e [RFC2215].
2. segnalazione per l'impostazione della chiamata: Tspec e Rspec devono essere trasportati ai router in cui saranno prenotate le risorse per la sessione. Il protocollo RVSP [RFC2205] è il protocollo scelto per la segnalazione.
3. ammissione della chiamata per elemento: ogni router che riceve i Tspec e Rspec può determinare se accettare o rifiutare la chiamata

L'architettura Intserv definisce due principali classi di servizio:

- servizio garantito [RFC2212]: fornisce precisi limiti ai ritardi di coda che un pacchetto può sperimentare in un router. Può essere

utilizzato per applicazioni hard-real-time.

- servizio di carico conttrollato [RFC2211]: non dà garanzie quantitative sulle prestazioni ma considera che una percentuale molto alta di pacchetti che usufruiscono di tale servizio passerà con successo tra i router senza essere scartata e con ritardi di coda prossimi allo zero. Potrebbe essere utilizzato per applicazioni soft-real-time.

Probemi di Intserv

Il modello Intserv presenta alcuni problemi principali:

- scalabilità: la necessità di elaborare le prenotazioni e di mantenere lo stato della prenotazione in ogni nodo della rete, per ogni flusso che lo attraversa, può portare a notevoli sovraccarichi nelle grandi reti
- flessibilità: il set predefinito di classi di servizio non permette la distinzione di classi più qualitative o la definizione relativa dei servizi

3.4.2 Servizi differenziati

Allo scopo di risolvere i problemi tecnici dell'infrastruttura Intserv, è stato proposto un approccio denominato Diffserv.

Diffserv può essere definita come una architettura di rete che

specifica un meccanimo semplice, scalabile e a grana grossa per classificare e gestire il traffico di rete e per fornire QoS su reti IP.

L'architettura Diffserv non richiede ai router di mantenere lo stato per flusso; classifica invece i pacchetti in un piccolo numero di aggregati per classi, alle quali i router forniscono un comportamento per hop.

L'architettura dei servizi differenziati è costituita da due insiemi di elementi funzionali:

- funzioni di estremità: alle estremità di ingresso nella rete vengono effettuate operazioni di classificazione dei pacchetti e condizionamento del traffico. Diverse classi di traffico riceveranno differenti servizi nella sezione interna della rete come descritto con il termine aggregato di comportamento in [RFC2475]
- funzioni della sezione interna: l'inoltro dei pacchetti nella sezione interna viene effettuato in accordo al cosiddetto comportmento per hop (PHB: Per Hop Behaviour) associato alla classe del pacchetto. Tale comportamento influenza come i buffer e la larghezza di banda del link di un router sono condivise fra le classi di traffico in competizione. Il comportamento sarà solamente basato sul marchio ricevuto dal pacchetto cioè in base alla classe di traffico a cui il pacchetto appartiene: in tal modo si evita la necessità di mantenere uno stato del router fra coppie individuali sorgente-destinazione risolvendo il problema di scalabilità.

All'interno della rete Diffserv sarà il comportamento per hop (PHB) specificato che determinerà il tipo di servizio ricevuto da un pacchetto.

Ad esempio sono stati definiti due tipi di PHB:

- il PHB di inoltro spedito (EF, Expedited Forwarding) [RFC2598] specifica che il tasso di partenza di una classe di traffico debba essere uguale o superiore a un tasso stabilito, ovvero viene riservata sufficiente larghezza di banda a tale traffico
- il PHB di inoltro assicurato (AF, Assured Forwarding) [RFC2597] riserva una quantità minima di larghezza di banda e di buffering. In caso di congestione si hanno preferenze di scarto tra le categorie.

Problemi di Diffserv

Anche Diffserv presenta problemi in ottica real-time. Bisogna notare che in tale architettura i dettagli di come i router individuali trattano il tipo di servzio è qualcosa di arbitrario ed è quindi difficile predire il comportamento end-to-end. Questo si complica ulteriormente se un pacchetto deve transitare attraverso due o più sistemi Diffser autonomi prima di raggiungere la destinazione. Il servizio Diffserv richiederebbe un accordo globale di come il servizio debba essere effettivamente differenziato.

Capitolo 4: Il protocollo TCP

In questo capitolo viene presentato il protocollo TCP nel dettaglio per capirne la "sintassi", la "semantica" e il "modus operandi".

4.1 Panoramica del protocollo TCP

TCP (Transmission Control Protocol) è un protocollo progettato da un gruppo di ricerca del DARPA (Departmentment of Defense (DoD) Advanced Research Project) le cui prime specifiche risalgono all'inizio degli anni '70. Nel corso del tempo il protocollo ha subito molte modifiche e si è evoluto continuamente ma lo standard di base è quello descritto nella RFC 793 del 1981 [RFC793].

TCP è pensato per essere usato come un protocollo di livello trasporto altamente affidabile e robusto tra host in una rete di comunicazione a commutazione di pacchetto e in sistemi interconnessi di tali reti. TCP fornisce un flusso affidabile di dati tra due nodi finali della comunicazione (protocollo end-to-end), servendosi del servizio di invio pacchetti inaffidabile fornito dal protocollo IP sottostante. Tra le sue operazioni ci sono quelle di suddividere i dati passati a esso dall'applicazione in segmenti di una certa dimensione, confermare i pacchetti ricevuti, settare timeouts per la ritrasmissione di pacchetti persi e altro.

Più specificatamente TCP fornisce un servizio orientato alla connessione, affidabile e a flusso di byte. [STE94]

Servizio orientato alla connessione significa che due applicazioni che usano TCP devono stabilire una connessione prima di poter scambiare dati. Da questo punto di vista TCP è un protocollo point-to-point ovvero permette di stabilire una connessione tra coppie di processi (unicast) e non fornisce in maniera trasparente alcun servizio multicast.

Servizio affidabile indica che i dati affidati a TCP saranno sicuramente consegnati all'altro host. L'affidabilità viene raggiunta attraverso l'utilizzo dei seguenti meccanismi:

- segmentizzazione: i dati dell'applicazione sono suddivisi in segmenti della dimensione opportuna; tale dimensione del segmento è decisa dal TCP in base alle caratteristiche della rete
- timer: quando TCP manda un segmento esso mantiene un timer, aspettando la conferma dell'arrivo dei dati dall'altro host. Se il riscontro (ack) non arriva in tempo, TCP reinvierà il segmento non ancora confermato
- acknowledgement: quando TCP riceve dati da un altro host, esso invia un segmento di conferma (ack); tale segmento può essere mandato immediatamente o si può anche aspettare per attendere l'arrivo di altri segmenti e fare in modo di confermare in un solo colpo un insieme di pacchetti (delayed ack)
- checksum: TCP mantiene un checksum sul suo header e dati.

Con tale meccanismo il protocollo previene qualsiasi modifica dei dati durante il transito nella rete: se un segmento ricevuto non passa il controllo della checksum allora TCP lo scarta (l'altro host lo ritrasmetterà allo scadere del proprio timeout)

- riordino dei segmenti: poiché i segmenti TCP sono trasmessi come datagrammi IP, e poiché i datagrammi IP possono arrivare fuori ordine, se necessario TCP, grazie al meccanismo dei numeri di sequenza, riordina i dati passandoli all'applicazione nel corretto ordine

Servizio a flusso di byte significa che l'unità di informazione è un byte (8 bit, anche detto ottetto); essendo un flusso dati, non esiste alcun marcatore dei dati inserito in TCP ovvero non esiste un meccanismo per strutturare i dati (tale responsabilità è lasciata ai protocolli di strato superiore): ciò significa che ad es. l'applicazione da un lato può fare tre invii da 20 byte e dall'altro lato l'altra applicazione può fare una ricezione singola da 60 byte.

TCP fornisce anche meccanismi per il controllo del flusso e controllo della congestione.

Il controllo del flusso è il meccanismo che consente di rispettare i vincoli (capacità di bufferizzazione) imposti dagli host; esso viene realizzato lato ricevente tramite il meccanismo delle finestre scorrevoli (sliding window protocol): la quantità di dati che il trasmittente può inviare non può superare la dimensione della finestra messa a disposizione (annunciata) dal ricevente.

Il controllo della congestione invece consente di rispettare tutte le altre comunicazioni presenti nella rete evitando il collasso della rete; tale meccanismo rende il protocollo equo e imparziale (fairness); viene tipicamente realizzato lato trasmittente: il tasso di trasmissione dati viene regolato dal trasmittente a seconda delle condizioni sperimentate della rete (tuttavia il controllo della congestione può essere realizzato in molti modi e tale aspetto del protocollo è tutt'oggi oggetto di ricerca e sperimentazioni che hanno come obiettivo quello di rendere il protocollo sempre più performante)

4.2 Header TCP

Tutti i servizi che sono stati accennati vengono realizzati grazie all'insieme delle informazioni presenti nei vari campi dell' header TCP.

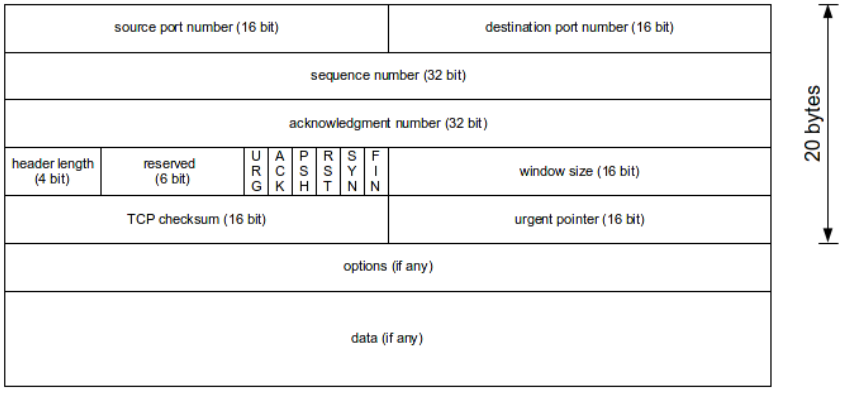

Fig. 6: header TCP

La parte fissa dell'header TCP ha dimensione 20 Byte (160 bit) ed è così composta:

- source port number (16 bit) e destination port number (16 bit): sono i numeri di porta sorgente e destinazione utilizzati per il multiplexing e demultiplexing delle applicazioni: questi due campi, assieme alla coppia source address e destination address dell'header IP, identificano univocamente ogni connessione (questa combinazione di 4 elementi viene a volte chiamata socket come nella specifica originale)
- sequence number (32 bit) è utilizzato per identificare i byte nel flusso dati dal TCP mandante al TCP ricevente
- acknowledgement number (32 bit) contiene il prossimo numero di sequenza che il TCP si aspetta di ricevere
- header length (4 bit) dà la lunghezza dell'header in parole di 32 bit; questo è richiesto perché la lunghezza del campo opzioni è variabile
- FIN, SYN, RST, PSH, ACK, URG, reserved (12 bit): sono flags-bit di controllo della connessione che possono essere presenti anche combinati; ogni flag specifica il tipo di segmento
- windows size (16 bit): dimensione della finestra annunciata dal TCP che ha spedito il pacchetto, ovvero è il numero di byte, a partire da quello specificato nel campo acknowledgement, che il ricevente è disposto ad accettare; tramite la finestra viene realizzato il controllo del flusso end-to-end

- checksum (16 bit): campo utilizzato per il controllo di errore; la checksum è il complemento a uno della somma in complemento a uno delle coppie adiacenti di ottetti facenti parte dell' header TCP, dei dati e di uno pseudo header (contenente alcuni campi dell'header IP) [RFC 1071]
- urgent pointer (16 bit): utilizzato per definire quali dati vanno contrassegnati come urgenti e valido solo se il flag URG è attivo

Le opzioni costituiscono la parte dell'header di lunghezza variabile: il campo options contiene le opzioni utilizzate per definire parametri della connessione o informazioni aggiuntive sul segmento; le opzioni sono pensate per estendere le funzionalità del TCP e permettere il tuning delle connessioni.

4.3 Gestione della connessione: apertura e chiusura

Una connessione TCP tra un host A e un host B è suddivisibile in tre fasi distinte:

- apertura della connessione
- scambio dati
- chiusura della connessione

L'apertura della connessione consente di stabilire e concordare i vari parametri della connessione (ad es. la dimensione massima del

segmento (MSS)) e avviene tramite il meccanismo denominato three way handshake che consiste nello scambio di 3 segmenti:

1. l'host A invia all'host B un segmento SYN contenente il proprio numero di sequenza iniziale (ISN: Initial Sequence Number), il proprio numero di porta, e la propria finestra.
2. l'host B risponde con il segmento SYN contenente il proprio numero di sequenza iniziale e conferma l'ISN dell'host A ponendo nel campo acknowledgement il valore A.ISN+1 (SYN "consuma" un numero di sequenza).
3. l'host A conferma il SYN dell'host B inviandogli l'ack contenente il valore B.ISN+1

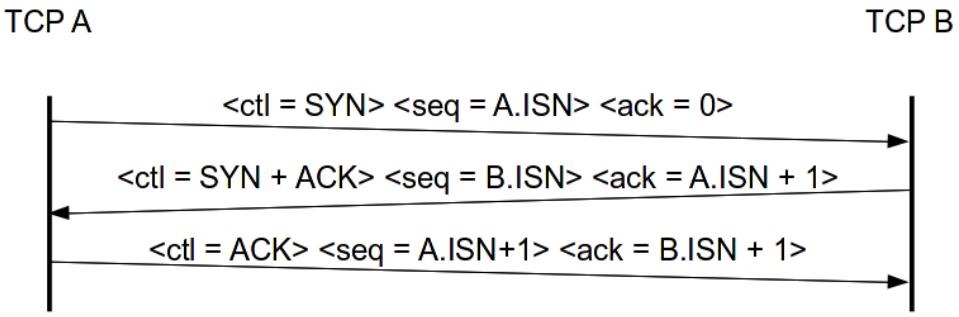

Fig. 7: apertura connessione: 3-way handshake

L'host A che apre la connessione si comporta da client e realizza quella che viene detta apertura attiva (active open), mentre l'host B si comporta da server accettando la connessione entrante e realizza quella che viene detta apertura passiva (passive open).

La chiusura della connessione invece è tipicamente a 4 vie per consentire il meccanismo della half-close: un host che chiude la connessione indica che non invierà più dati ma è disposto a riceverne.

Lo scambio di segmenti per una mezza chiusura è dunque il seguente:

1. l'host A decide di chiudere la connessione e invia un segmento FIN all'host B
2. l'host B conferma il segmento ricevuto ponendo il valore A.FIN+1 nel campo acknowledgement (FIN "consuma" un numero di sequenza)

La stessa procedura sarà ripetuta in maniera duale quando l'host B intenderà terminare la connessione.

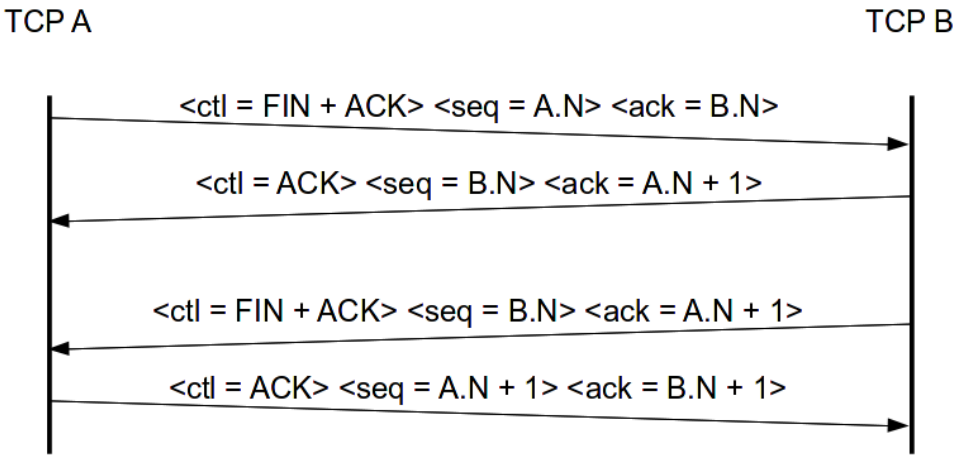

Fig. 8: chiusura connessione: half close e 4-way handshake

Il rilascio vero e proprio della connessione sarà effettuato solo dopo aver atteso un tempo definito 2*MSL (Maximum Segment Lifetime): tale tempo di attesa (quiet time) viene utilizzato per impedire che i segmenti ritardati dalla rete derivanti da una connessione precedente possano essere interpretati come facenti parte di una nuova connessione.

Lo scambio dei dati, che può essere full-duplex, avviene solo per una connessione stabilita. TCP può essere usato per lo scambio dati sia di applicazioni interattive che per applicazioni per il trasferimento di grandi quantità di dati.

Per ottenere buone performance e un utilizzo efficiente del canale TCP opera secondo la tecnica del pipelining: un flusso di tipo pipelining consente al mittente di avere più segmenti trasmessi ma non ancora riscontrati in ogni dato istante. Il pipelining consente di migliorare notevolmente il throughput di una sessione rispetto alla modalità stop and wait (spedizione di un singolo segmento per volta e attesa del riscontro) in particolar modo quando il rapporto tra la dimensione del segmento e il ritardo di andata e ritorno è piccolo. La tecnica del pipelining costringe i lati sender e receiver alla necessità di bufferizzazione: lato sender devono essere memorizzati i pacchetti inviati e non ancora riscontrati e lato receiver si devono memorizzare i pacchetti ricevuti in attesa di consegna all'applicazione.

Lo scambio dati quindi avviene tipicamente a burst ovvero inviando in sequenza un certo numero di segmenti fino a saturazione o del canale

(raggiunta la finestra di congestione) o delle capacità di ricezione dell'altro host (raggiunta la finestra annunciata dall'altro host).

TCP è un protocollo ad eventi (event-driven o event-triggered) e ack-clocked: il tasso di invio su rete è regolato dal tasso di ricezione degli ack.

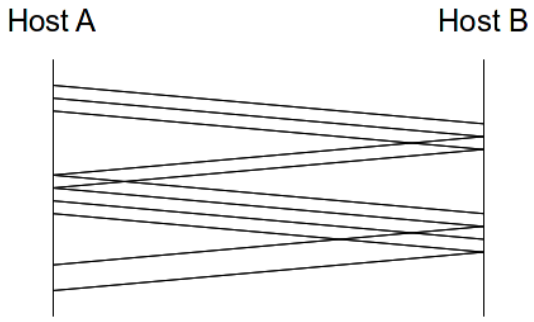

Fig. 9: esempio scambio dati TCP

4.4 Stati della connessione

Una connessione TCP avanza attraverso una serie di stati durante il suo ciclo di vita. Il passaggio da uno stato ad un altro avviene in risposta ad eventi che possono essere generati dalle chiamate utente, dal tipo dei segmenti entranti e dallo scadere dei timeout. Il seguente diagramma riassume l'insieme degli stati e un sottoinsieme delle transizioni di stato che si possono verificare per ogni connessione.

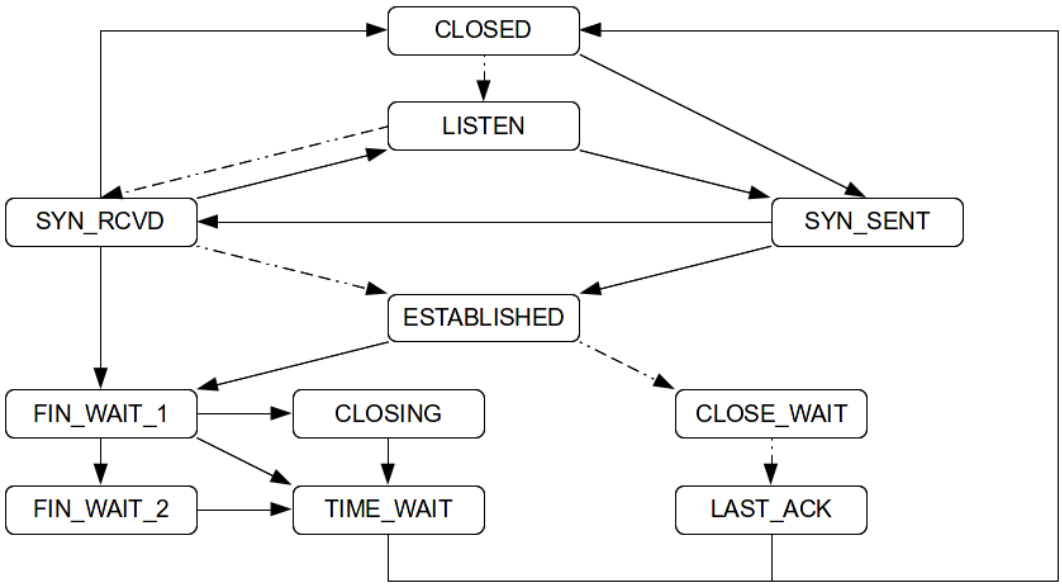

Fig. 10: diagramma transizioni di stato di una connessione

4.5 Meccanismi base di controllo della congestione

L'ambito in cui TCP ha subito notevoli modifiche rispetto alla specifica originale è sicuramente il controllo della congestione.

Tale problematica nasce dal fatto che la rete è una entità dinamica e non è possibile conoscere il reale stato della rete; il carico della rete non è deterministico ma aleatorio.

Sono stati proposti moltissimi algoritmi per risolvere tale problema e non ne esiste uno definitivo.

Uno dei più utilizzati è quello descritto nella RFC 2001 che si compone

di quattro algoritmi distinti, con scopi differenti ma che sono tipicamente implementati tutti assieme per offrire il giusto compromesso tra controllo della congestione, fairness e performance.

Tale combinazione di algoritmi prende anche il nome di TCP Reno ed è composto da:

- slow start: risolve parzialmente il problema di conoscere lo stato della rete; si basa sull'idea di aumentare esponenzialmente il numero dei pacchetti da iniettare in rete (soglia definita da sshtresh) in modo da sondare le condizioni di massimo carico a cui è possibile trasmettere dati
- congestion avoidance: viene utilizzata una finestra di congestione (cwnd) che serve a regolare la massima quantità di dati che si possono iniettare nella rete e a stabilire se si è in fase di slow start o in fase di congestion avoidance; ogni volta che si verifica una congestione (indicata dalla scadenza di un timeout o da un ack duplicato) vengono regolati i valori di sshtresh e di cwnd che viene dimezzata; in fase di congestion avoidance cwnd viene aumentata in maniera additiva
- fast retransmit: permette di aumentare le performance del TCP; alla ricezione del terzo ack duplicato TCP può ritrasmettere immediatamente il primo segmento non ancora confermato senza aspettare la scadenza del timeout
- fast recovery: dopo che fast retransmit abbia mandato quello che appare essere il segmento perso, tale algoritmo impone di

tornare in fase di congestion avoidance e non slow start

Nel complesso i quattro algoritmi insieme utilizzano un approccio conservativo secondo cui la crescita del tasso dei segmenti da iniettare in rete è additivo mentre il tasso di decrescita è moltiplicativo (AIMD: Additive Increase Multiplicative Decrease).

Il meccanismo utilizzato è un meccanismo adattativo e reattivo (evita la congestione adattando il proprio comportamento reagendo a situazioni di avvenuta congestione).

Altri meccanismi di controllo della congestione

Nel corso degli anni sono state proposte molte estensioni e/o varianti a TCP Reno per cercare di migliorarne le performance, e inoltre sono stati anche proposti algoritmi completamente alternativi ad esso.

Tra questi possono essere menzionati NewReno, Wetwood, Westwood+, Vegas e molti altri.

Alcuni di questi algoritmi sono di tipo proattivo ovvero cercano di evitare in anticipo il verificarsi di situazioni di congestione.

4.6 Altri meccanismi disponibili

Il protocollo TCP è ad oggi utilizzato per qualsiasi tipo di comunicazione in ambiti molto differenti; la sua flessibilità (e il superamento dei limiti delle specifiche iniziali) è dovuta anche grazie

all'utilizzo delle opzioni e a meccanismi che consentono di ottimizzare le performance e ridurre l'overhead della comunicazione.

Estensione della finestra

Il numero di byte che può essere iniettato in una rete prima di ottenere un riscontro dipende dal prodotto del ritardo per la banda disponibile (capacity = delay*bandwidth). Da tale relazione si nota che all'aumentare del ritardo o della larghezza di banda, aumenta anche il numero di byte in transito sul canale (pipe). In caso di ampie larghezze di banda su lunghi percorsi, il numero di segmenti in sospeso nella rete può essere talmente alto da superare i limiti intrinseci imposti dal campo window size nell'header del TCP. Per risolvere questi problemi ed altri legati alle performance sono utilizzati i meccanismi di Windows Scale, Timestamp e PAWS [RFC 1323]. Tali meccanismi si basano sull'utilizzo di opzioni da scambiare nell'header che consentono di estendere la semantica del protocollo.

Estensione del meccanismo di ACK cumulativo

TCP è un protocollo a finestra scorrevole che utilizza il paradigma dell'ack cumulativo: un ack conferma tutti i byte con numero di sequenza precedente al valore dell'ack stesso. Tuttavia il meccanismo dell'ack cumulativo a volte è limitante e può portare ad effetti indesiderati sul carico della rete: quando viene perso un segmento all'interno di un burst, TCP potrebbe ritrasmettere inutilmente

segmenti già ricevuti dall'altro host. Per evitare tali situazioni sono stati proposti i meccanismi del Negative Acknowledgement (NACK o NAK [RFC 1106]) o del Selective Acknowledgement (SACK [RFC2018]). Anche questi meccanismi sono possibili grazie all'uso di opzioni scambiate nell'header e vanno ad estendere la semantica del protocollo.

Capitolo 5: Implementazione TCP in SHaRK

In questo capitolo viene decritta una prima implementazione del protocollo TCP standard sul sistema operativo S.Ha.R.K.

5.1 Obiettivi

L'implementazione ha avuto come obiettivo quello di fornire al sistema operativo SHaRK le funzionalità che permettono di sviluppare applicazioni che fanno uso del TCP. Essendo TCP un protocollo complesso, esteso e in continua evoluzione, attualmente è stato implementato quanto descritto nello standard RFC 793 [RFC793] e il controllo della congestione secondo la RFC 2001 [RFC2001].

Tra gli obiettivi dell'implementazione, in ordine di importanza, ci sono:

- predicibilità (primaria in un RTOS)
- organizzazione modulare (requisito di qualsiasi codice di una certa complessità)
- semplicità e flessibilità (permettere facili cambiamenti futuri)
- performance

L'implementazione del protocollo [STE95] risulta non banale soprattutto a causa del fatto che implementazioni formalmente e

funzionalmente corrette possono risultare molto povere dal punto di vista delle performance (ad es. silly window sindrome [STE94], calcolo della checksum [RFC 1071], determinazione del retransmission timeout [RFC2988], ...)

5.2 Libreria per la rete

Per permettere la comunicazione tra differenti computer, SHaRK fornisce come base una libreria di rete che implementa lo stack UDP/IP su rete Ethernet. La libreria è organizzata in strati rispettando la suddivisione logica prevista dallo standard: ogni protocollo è implementato in maniera logicamente e fisicamente separata dagli altri. Tuttavia vi sono oggetti dello stack di rete che sono utilizzati da diversi layer (ad es. netbuff).

Il driver a basso livello è implementato al fine di rispettare le richieste del sistema real-time (cercando di evitare i ritardi non predicibili nel mandare e ricevere frames). Questo risultato è raggiunto risolvendo due problemi differenti: la gestione degli interrupt nella fase di ricezione e l'esclusione mutua nell'accesso alla scheda di rete nella fase di trasmissione.

Il primo problema è risolto usando un task soft real-time per gestire gli interrupt della scheda di rete: l'ISR (Interrupt Service Routine) consiste nella scrittura di un messaggio su una porta di comunicazione sulla quale attende il task ricevente in lettura

bloccante. Tale task è garantito assieme a tutti gli altri task del sistema e in tal modo non può mettere a repentaglio la loro schedulabilità. Poiché un minimo tempo di interarrivo per un frame non può essere previsto, il task ricevente non può essere un task sporadico hard real-time, e quindi il task usa un modello di task soft-real-time servito da un Costant Bandwidth Server (CBS).

Il secondo problema attualmente viene risolto adottando il paradigma di programmazione a memoria condivisa: un task che vuole trasmettere accede alla scheda di rete in maniera mutuamente esclusiva garantita da semafori. Tale soluzione è molto semplice e introduce un overhead piuttosto basso (un altro metodo potrebbe essere basato sull'utilizzo di un task servente periodico dedicato a mandare frame nella rete a beneficio di altri task: ogni task che avesse dati da spedire potrebbe inserire il proprio messaggio in una mailbox da dove il task trasmittente prenderebbe i messaggi da inviare).

5.3 Architettura

Prima di passare all'implementazione vera e propria di qualsiasi sistema informatico è buona norma progettare i dettagli architetturali. Sono stati identificati gli oggetti che faranno parte del sistema e le principali interazioni tra di essi. Di seguito sono elencati i principali elementi individuati:

- tasks: vi sono tre compiti distinti per realizzare la comunicazione: ricezione, trasmissione e ritrasmissione dei segmenti non confermati; risulta così naturale effettuare una divisione in tre task; bisogna notare che il task ricevente e il task timer gestiscono le proprie fasi per tutte le connessioni attive nel sistema mentre viene creato un task trasmittente per ogni nuova connessione
- buffers: l'applicazione viene disaccoppiata dalle logiche della comunicazione attraverso l'utilizzo di due buffer circolari; un buffer viene utilizzato per l'invio e uno per la ricezione dati
- transmission control block (TCB): il TCP è un protocollo stateful e per tale motivo devono essere memorizzate molte variabili per gestire la comunicazione; tutte queste variabili vengono raggruppate in un'unica struttura dati la quale è condivisa tra tutti i task dello stack di rete

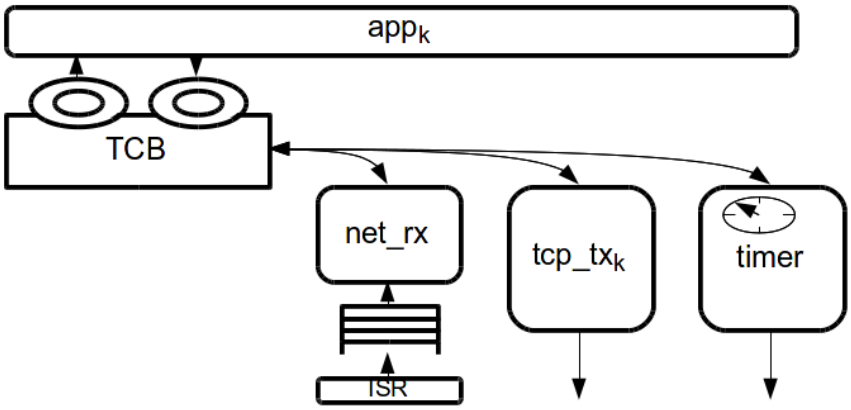

Fig. 11: elementi architetturali dello stack di rete

5.4 Interfaccia fornita alle applicazioni

Alle applicazioni sono fornite due tipi di interfacce: una in *./drivers/net/include/tcp.h* ispirata a quella descritta nella specifica standard originale e una in *./drivers/net/include/tcp_socket.h* di tipo socket conforme a quella dei sistemi POSIX.

L'interfaccia socket non è altro che una rimappatura delle funzioni offerte dal modulo tcp.

Per utilizzare i servizi offerti dal TCP sono presenti le seguenti funzioni:

```
/* function to manage TCP module */
void tcp_init(void *dummy);
void tcp_exit();

/* function to manage TCP connection */
int tcp_create_connection(TCP_ADDR local_addr, TCP_ADDR remote_addr);
void tcp_delete_connection(int id_conn);

/* function to use a TCP connection */
int tcp_open(int id_conn, TCP_OPEN_TYPE type);
ssize_t tcp_send(int id_conn, const void *buff, DWORD count);
ssize_t tcp_receive(int id_conn, void *buff, DWORD count);
int tcp_close(int id_conn, TCP_CLOSE_TYPE type);

enum tcp_states tcp_status(int id_conn);
```

Il modulo che consente di utilizzare le funzionalità del protocollo va inizializzato tramite la funzione tcp_init la quale registra il modulo e inizializza le strutture dati necessarie.

Ogni connessione va dapprima creata tramite la funzione *tcp_create_connection* che alloca il TCB, lo inizializza e lo registra nella tabella delle connessioni. Tale funzione ritorna un intero che va usato come identificativo univoco da utilizzare nelle altre funzioni.

La chiamata alla funzione *tcp_open* è di tipo bloccante, ovvero l'applicazione chiamante sarà bloccata in attesa dell'avvenuta apertura della connessione. In maniera analoga si comporta la funzione *tcp_close*.

Invece la funzione *tcp_send* non è bloccante: tale funzione ritorna il numero di byte accettati per l'invio; è compito dell'applicazione controllare che tutti i suoi dati siano stati accettati e in caso negativo ritentare l'invio dei dati mancanti.

In maniera analoga si comporta la funzione *tcp_receive*: l'applicazione stabilisce il numero massimo di byte che intende ricevere attraverso la variabile count; la funzione tornerà il numero di dati effettivamente passati all'applicazione disponibili nel buffer puntato da *buff.

5.5 Separazione tra applicazioni e TCP: buffer circolari

Per disaccoppiare l'applicazione dai task di rete è stato scelto l'utilizzo dei buffer circolari. Un buffer circolare è una struttura dati che usa un singolo buffer di dimensione fissa sul quale accedono due tipi di entità: almeno un task scrittore e almeno un task lettore. La scelta del buffer circolare è determinata dal fatto che, grazie a tale soluzione, è necessario allocare la memoria solo una volta prima della connessione e quindi permette di minimizzare l'impredicibilità dovuta al tempo di allocazione dinamica della memoria. Il buffer è stato implementato per permettere l'accesso non bloccante di un solo scrittore e di un solo lettore grazie all'utilizzo di un contatore specifico per ognuno dei task (evitando così di avere variabili condivise in scrittura). Tale tecnica permette di evitare bloccaggi nell'accesso al buffer e risolve il problema della distinzione tra buffer pieno e buffer vuoto. E' bene notare che eventuali politiche di bloccaggio possono essere sempre implementate sopra politiche non bloccanti.

```
/* buffer circolare */
typedef struct circular_buffer
{
    /* pointer to buffer memory location */
    void *buffer_start;
    /* max number of items in the buffer [0, 65535] (2 byte) */
    unsigned short int capacity;
    /* pointer to writer and reader position */
    void *writer_next;
    void *reader_next;
```

```
    /* writer and reader counters */
    unsigned int writer_count;
    unsigned int reader_count;
} circular_buffer;
```

5.6 Stutture dati (variabili di stato)

Va ricordato che TCP è protocollo stateful nel senso che tutti i dettagli sullo stato di una connessione sono mantenuti a tale livello. Per tale motivo per ogni connessione è necessaria una struttura dati denominata *tcb* (trasmission control block) che contenga tutte le variabili per il controllo della connessione.

```
/* transmission control block (TCB) structure */
typedef struct tcp_tcb_t {
    unsigned int local_connection_name;

    /* {local_addr, remote_addr} tuple identify a connection */
    TCP_ADDR local_socket;
    TCP_ADDR remote_socket;

    /* different behaviour in segment processing */
    TCP_OPEN_TYPE open_type_flag;
    TCP_CLOSE_TYPE close_type_flag;

    /* connection state */
    enum tcp_states state;

    /* TCP Variables */
    struct tcp_send_data snd;
    struct tcp_recv_data rcv;
```

```
    /* local and remote maximum segment size */
    WORD local_mss;
    WORD remote_mss;
    WORD mss;

    /* rx and tx buffers */
    circular_buffer *cbuf_rx;
    circular_buffer *cbuf_tx;

    /* data segment send list */
    list_desc *pending_send_list;
    /* data segment recv list */
    list_desc *pending_recv_list;

    /* Timer counters */
    TIME user_timeout;
    TIME time_wait_timeout;
    struct rto_struct rto_var;

    /* congestion window (slow start) */
    WORD cwnd;
    /* slow start threshold size (congestion avoidance) */
    WORD sstresh;
    /* last ACK number (fast retransmit + fast recovery) */
    TCP_SEQ_NUM last_ACK_number;
    /* duplicate ACK counter (fast retransmit + fast recovery) */
    unsigned int dupl_ACK_counter;
} tcp_tcb_t;
```

Tale struttura viene definita nel file *./drivers/net/include/tcp_tcb.h*.

E' previsto che tale struttura sia acceduta solo dai task di rete e non dai task applicativi. Poichè tale struttura è condivisa, l'accesso ad

essa è regolato da un mutex utilizzando il protocollo SRP (Stack Resource Policy). L'utilizzo di tale protocollo evita il deadlock e le catene di bloccaggi e inoltre riduce i cambi di contesto rispetto ad altre soluzioni.

L'accesso alla struttura non è diretto ma viene effettuato attraverso le funzioni presenti in *./drivers/net/include/tcp_tbl.h* che contiene anche tutte le funzioni per gestire le connessioni del sistema. Infatti ogni connessione viene registrata in una tabella, la quale viene scandita ogni volta che è necessario fare demultiplexing ovvero quando si deve definire a quale connessione appartenga un determinato segmento entrante dalla rete.

5.7 Gestione eventi

Tutta la logica di processamento dei segmenti è racchiusa in un insieme di funzioni nel file *./drivers/net/include/tcp_proc.h*. E' presente una funzione per ogni evento che può verificarsi durante il ciclo di vita della connessione. Gli eventi sono divisibili in tre gruppi:

- eventi generati dall'applicazione
- eventi generati dall'arrivo di un particolare segmento
- eventi generati dalla scadenza di un timeout

Ognuna di queste funzioni necessita di conoscere lo stato della connessione (accede al TCB) e ne modifica le variabili.

Il processamento avviene in maniera quasi del tutto analoga a quanto descritto nella RFC 793: le operazioni da effettuare dipendono dal tipo di evento verificatosi e dallo stato attuale della connessione.

5.8 Comunicazione tra task e sincronizzazione

Per portare a termine con successo il trasferimento dati, i task di rete e quello applicativo devono poter comunicare e/o sincronizzarsi.

Per la comunicazione e sincronizzazione tra task, SHaRK offre vari meccanismi:

- semafori POSIX
- mutex
- semafori interni
- porte di comunicazione
- CAB (Circular Asyncrhronous Buffers)
- code di messaggi POSIX
- segnali POSIX

A seconda delle esigenze specifiche è stata scelta la soluzione che è stata ritenuta più adatta alla situazione.

TCP è un protocollo intrinsecamente bloccante: il task trasmittente deve bloccarsi continuamente ogni volta che non può trasmettere dati o per il controllo del flusso o per il controllo di congestione. E'

necessario stabilire un meccanismo che consenta di bloccare e sbloccare tale task. In questo caso è stata preferita la soluzione a semafori interni per la loro semplicità e il basso overhead ma lo stesso risultato poteva essere ottenuto con altri meccanismi.

Capitolo 6: TCP e il Real-Time

Questo capitolo, sulla base dei precedenti capitoli, mostra alcune estensioni del protocollo rispetto ai requisiti di una comunicazione in tempo reale. A tale scopo viene delineata un'architettura di supporto e viene determinato un diverso modus operandi del protocollo.

6.1 Considerazioni e scenario

Nei precedenti capitoli si è visto che per cercare di dare "vere" garanzie di comunicazione real-time è necessaria una infrastruttura di rete che vada oltre il best-effort e che in qualche modo riesca a fornire meccanismi o per riservare la banda o che permettano di stabilire dei parametri di lavoro nel caso peggiore (ad es. un valore di WCTT: Worst Case (Network) Traversal Time).

Nuovi componenti strutturali vanno aggiunti all'insfratruttura di rete per proteggere una applicazione dalle congestioni e dal jitter impredicibile.

Ipotesi preliminari

Nel seguito viene fatta l'assunzione di disporre di una infrastruttura di rete real-time. Ciò significa che si suppone che esistano meccanismi atti a riservare le banda richiesta e che durante l'apertura della

connessione venga creato una qualche forma di circuito virtuale. In tale scenario non ha senso parlare di congestione poichè la banda viene richiesta all'instaurazione della chiamata. In tale scenario si supporrà che il valore del WCTT (Worst Case (Netwok) Traversal Time) possa essere ritenuto costante per tutta la durata della sessione di trasferimento dati.

6.2 Analisi delle architetture in ottica RT

La prima fase è quella di studiare l'architettura di uno stack di rete real-time dal più ampio punto di vista possibile. Per conservare la più ampia generalità, si suppone che la suddivisione dello stack possa avvenire per ogni livello dello stack, per ogni fase della comunicazione (trasmissione e ricezione) e per ogni applicazione che faccia utilizzo della rete.

Quindi, per esempio, si potrebbe decidere di istanziare un task che gestisca solamente la fase di ricezione del protocollo Ethernet per ogni applicazione e consegni i frame ad una altro task sopra di esso.

Naturalmente non tutte le architetture sono realizzabili implementativamente. Ad esempio non è possibile dividere lo stack in n task per n applicazioni nella fase di ricezione. Infatti, nella routine di gestione dell'interrupt di rete non si hanno informazioni a sufficienza per determinare quale sia il task incaricato del processamento del frame. A questo problema si potrebbe ovviare

eseguendo un early-demultiplexing nella routine di gestione dell'interrupt; tuttavia, così facendo, si appesantirebbe la gestione dell'interrupt di rete e si uscirebbe dallo schema architetturale ISO/OSI.

Di conseguenza dal lato di ricezione, un protocollo della pila ISO/OSI può essere diviso in più task se e solo se il protocollo della pila inferiore è in grado di effettuare un'operazione di demultiplexing tra i task del livello superiore. Supponendo che il protocollo di trasporto si occupi dell'operazione di demultiplexing, ne consegue che non ha senso supporre una soluzione con più di un task per protocollo dal lato ricezione.

In base a questa caratterizzazione, sarebbe possibile istanziare, in fase di trasmissione $p \cdot n$ task, dove p è il numero di protocolli utilizzati nello stack e n il numero di applicazioni. Una soluzione del genere, però, genererebbe nel sistema un overhead significativo, nel context-switch, nello scheduling, e nella memoria utilizzata.

Nel seguito, quindi, saranno esaminate una ad una solo le soluzioni architetturali che si ritengono applicabili:

1. State of the art: l'applicazione gestisce direttamente la trasmissione, esiste un task che effettua la ricezione
2. Stack task: un unico task trasmette e riceve per tutte le applicazioni
3. Sender and receiver tasks: un unico task gestisce la

trasmissione per tutte le applicazioni, un unico task effettua la ricezione

4. Office boys (fattorini): n task sono utilizzati per la trasmissione e un task si occupa della fase di ricezione

Caso 1 - SHaRK state of the art

Questa è l'architettura attuale dello stack di rete UDP/IP in SHaRK.

A livello architetturale, esiste un solo task per la ricezione dei messaggi. All'arrivo di un interrupt dalla scheda di rete, il frame in ingresso viene salvato in un buffer che viene letto, in modalità bloccante, dal task in ricezione (Soft-Real-Time, garantito per mezzo di un server CBS). Una volta letto il pacchetto dal buffer, il receiver task provvede a processarlo per tutti i livelli dello stack per mezzo delle apposite chiamate di funzione.

Dal lato di trasmissione, l'intero stack è gestito incapsulandolo nell'applicazione, che effettua delle chiamate di funzione sulla libreria di rete. Questo significa che il tempo di stack deve essere incluso nel WCET dell'applicazione.

Questo approccio comporta diverse difficoltà implementative in ottica RT. In primo luogo è difficile mappare i parametri dei task sulle richieste di QoS per la rete; in secondo luogo è necessario distinguere i task che utilizzano la rete dagli altri task.

Vantaggi:

- overhead piuttosto basso
- facilità di realizzazione

Svantaggi:

- lo stack di rete non fornisce garanzie real-time
- l'onere di strutturare una applicazione che trasmetta pacchetti in maniera periodica e predicibile ricade interamente sulle spalle del programmatore dell'applicazione
- poichè tutto lo stack è incluso nell'applicazione, la stima del WCET diventa molto difficile

Caso 2 - Stack task

In questa architettura vi è un solo task che gestisce l'intero stack. Questo task deve indubbiamente schedulare la risorsa di rete tra le applicazioni che debbono comunicare.

Un approccio di questo tipo, a dispetto della estrema semplicità concettuale, è estremamente complesso a livello implementativo. Infatti questo unico task si dovrebbe far carico non solo di inviare i vari pacchetti nella rete, ma anche di ricevere le risposte ed inoltrarle all'applicazione destinataria. Oltre a questo, il task dovrebbe "switchare" tra due modi (ricezione e trasmissione), e decidere in merito alla possibile acquisizione della risorsa di rete da parte di una nuova applicazione che desideri comunicare.

L'estrema complessità delle operazioni fa sì che il WCET di un tale

task sia molto difficile da stimare. Anche in presenza di una stima, questa potrebbe risultare estremamente alta, pregiudicando la quantità di banda utilizzabile dalle applicazioni Hard Real Time. In questo modo il collo di bottiglia non sarebbe più il canale, ma il task stesso. Dal punto di vista implementativo tale opinione è corroborata dal fatto che il task dovrebbe gestire tutte le variabili della connessione per ogni singola applicazione che intende comunicare, tenendole tra l'altro distinte tra loro.

Data la complessità implementativa e gli svantaggi impliciti, si ritiene che questa architettura non sia adatta ad uno stack di rete real-time.

Caso 3 - Sender and receiver tasks

Questa architettura prevede la presenza di due task distinti, uno per la trasmissione e uno per la ricezione, col compito di servire tutte le applicazioni che intendono comunicare su rete.

In un contesto del genere il sender non può bloccarsi in alcun modo, perchè un suo blocco pregiudicherebbe l'intero servizio di rete, e non si limiterebbe a penalizzare la singola applicazione all'origine del blocco (principio del task-isolation). Ne consegue che il task deve servire le applicazioni secondo una opportuna politica di scheduling. In particolare deve distinguere i dati provenienti dalle diverse applicazioni, e le connessioni create per servire le applicazioni medesime. Ad esempio nel TCP, in presenza di uno slow start su una connessione, tutti i segmenti appartenenti a quella connessione

debbono rimanere in attesa fin quando non viene riscontrato un ACK per la connessione medesima. Più precisamente possiamo dire che il sender task deve garantire l'isolamento tra le varie connessioni. Più che i singoli pacchetti lo scheduler potrebbe gestire, a un livello più elevato, le connessioni. In questo caso lo scheduler deciderebbe soltanto quale connessione ha il "diritto" di occupare il canale; una volta presa questa decisione selezionerebbe dal buffer un numero congruo di pacchetti per l'invio. Il task receiver ha una ulteriore responsabilià rispetto ai casi precedenti, che consiste nello sbloccare le connessioni in attesa di riscontro, trasmettendo un opportuno messaggio al sender.

Caso 4 - Office boys (fattorini)

Questa architettura prevede la presenza di un solo task receiver per tutte le connesioni e di un task sender per ogni connessione. La decisione riguardo alla connessione da schedulare (ovvero, la risposta alla domanda: a quale connessione assegno il canale?) viene presa "gratuitamente" dallo scheduler della CPU. Deve però esistere un mapping tra le richieste di QoS dell'applicazione sulla rete e i parametri dei task serventi sulla CPU (periodo, deadline, etc...).

Quindi, dal punto di vista dell'applicazione, abbiamo due QoS: il primo per il servizio dell'applicazione medesima sulla CPU, il secondo per il servizio sulla rete. Questo QoS andrà mappato nei parametri di servizio del task "servente" (il sender) sulla CPU, ovvero nella richiesta di QoS del servente per la CPU.

Un problema relativo a questa architettura consiste nell'accesso concorrente all'n-esimo TCB da parte del receiver e dal servente n. All'arrivo di un ACK il receiver ha necessità di accedere al TCB per modificare le variabili di stato della connessione; nello stesso istante di tempo il TCB potrebbe essere in uso da parte del servente n. Dunque il receiver ha alta probabilità di bloccarsi in attesa dell'n-esimo TCB; ricordiamo che il blocco del receiver è uno scenario indesiderabile.

Una possibile soluzione consiste nell'assegnare ogni competenza che comporti l'accesso al TCB al servente. Il receiver si limiterebbe a segnalare, successivamente al demultiplexing, la ricezione di un segmento al servente, che provvederebbe poi al suo processamento. In questo scenario il servente, come prima azione, processa eventuali segmenti in ingresso dal receiver, aggiorna i parametri della trasmissione e quindi prosegue nei suoi normali compiti.

Si può confrontare questa architettura con lo stato dell'arte (caso 1).

Nell'architettura del caso 1 sarebbe possibile effettuare un mapping tale da far corrispondere il QoS della rete con quello sulla CPU per ogni singola applicazione. Tuttavia nei parametri si deve includere il tempo che il task sfrutta per la sua normale computazione; ad es.: il WCET del task, oltre al tempo dello stack, comprende il tempo necessario a computare i dati da inviare. Questo comporta che il task chiederebbe molta più banda rispetto a quella necessaria, e il canale sarà sicuramente sotto-utilizzato.

Nell'architettura del caso 4, invece, il WCET del servente è il tempo che realmente esso richiede per utilizzare il canale.

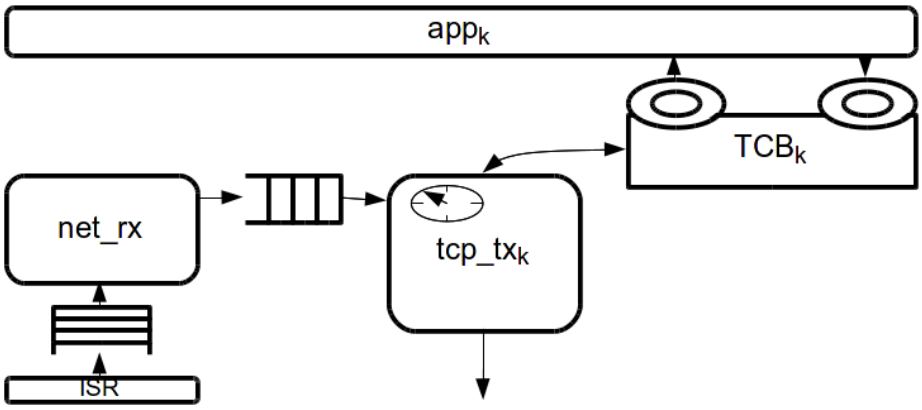

Fig. 12: elementi architetturali dello stack di rete RT

Vantaggi:

- si ottiene la schedulazione del canale per mezzo dello scheduling degli n serventi sulla CPU
- essendo il servente periodico, il timer TCP è implicitamente definito nel periodo del task servente
- nessun accesso bloccante ad eventuali risorse condivise

Svantaggi

- overhead di context-switch sarebbe significativo in presenza di numerose connessioni (n elevato). Nel caso peggiore si potrebbe avere un context-switch per ogni burst trasmissivo

A questo punto sarabbe opportuno chiedersi quali vantaggi comporterebbe una soluzione del genere rispetto al prima caso, e se non sia invece più semplice adottare quel tipo di architettura.

Innanzitutto si nota che la natura intrinsecamente bloccante del protocollo TCP inciderebbe sul task applicativo complicando di molto la definizione dell'applicazione in ottica real-time. Il programmatore dovrebbe conoscere i meccanismi intrinseci al protocollo, e quindi il suo uso non sarebbe trasparente al programmatore; ciò complicherebbe le logiche dell'applicazione, rendendo difficoltose le stime del periodo e del WCET.

Se il servente TCP è reso periodico, il timer di ritrasmissione è implicito nel suo periodo. Questo fa in modo che il TCB sia ad accesso esclusivo da parte del servente, che dunque è l'unico gestore dello stato della connessione TCP.

In conclusione questa sarà l'architettura scelta come riferimento nel prosieguo.

6.3 Estensioni real-time al TCP

Ricordiamo che per una comunicazione in tempo reale sono stati identificati i seguenti principi:

- differenziazione dei tipi di traffico
- isolamento dei tipi di traffico

- uso efficiente delle risorse
- ammissione della chiamata

Considerati tali principi, il principale servizio che TCP dovrebbe offrire consiste nello scheduling del canale, guidato da un contratto stabilito dalle richieste di QoS per ogni connessione, allo stesso modo di quanto accade con i task Hard Real Time nello scheduling della CPU.

Il problema del task trasmittente

L'invio dei dati nel protocollo TCP è concepito non come un unico processo indivisibile, ma come un insieme di tante piccole sotto-operazioni. Una applicazione che intenda inviare dei dati non se li vedrà trasmettere tutti in una volta, ma in tanti piccoli segmenti e in diversi burst (nel TCP standard ogni burst consiste nell'invio nella rete di un numero variabile di segmenti). Questo implica che la singola connessione TCP non può essere effettuata in un singolo job, ma deve essere servita per mezzo di una serie di job distinti. Ognuno di essi porterà a termine una serie di sotto-operazioni relative ad un singolo burst. Tale serie di job può essere concepita in diversi modi:

1. un task aperiodico per ogni job
2. un task sporadico che raggruppa tutti i job
3. un task periodico

Il primo approccio è perfettamente ragionevole e consente di certo di

fornire garanzie sul singolo burst, ma rende molto difficile fornire una garanzia sulla globalità della trasmissione; il secondo approccio è riconducibile ad uno degli altri due e ricondotto al primo soffre degli stessi problemi.

Il terzo approccio è quello che si adotterà nel prosieguo, perchè consente di fornire garanzie sia sul singolo job che sulla totalità della trasmissione, vista come la somma dei singoli job (che consiste nell'intera vita del task medesimo).

Utilizzando un task periodico per la trasmissione dati, il protocollo TCP diventa time-triggered e ACK-regulated (bisogna ricordare che TCP standard è invece event-triggered e ACK-clocked). Rendere TCP un protocollo ad invio temporizzato avrà come side-effect positivo il fatto che l'implementazione dei diversi timer è resa superflua, in quanto implicita nel periodo scelto.

Il problema del task ricevente

Il task ricevente dovrebbe essere time-triggered o event-triggered? Ovvero l'invio dell'ACK avviene su base temporale (ad ogni periodo del task ricevente) o in modo simile all'interrupt (all'istante di arrivo di una generica quantità di pacchetti)?

Un task ricevente non periodico non potrebbe garantire il tempo massimo di reinvio dell'ack.

Va notato che se il ricevente è temporizzato (periodico e real-time), è possibile calcolare il tempo massimo in cui esso andrà in esecuzione e

invierà l'ack dei segmenti riscontrati. Il massimo tempo di ritardo possibile è dato proprio da un periodo del receiver.

Per tale motivo si ipotizza che anche il task ricevente sia periodico.

Se il ricevente è periodico, come stabilire i parametri del task ricevente?

- li stabilisce il sistema locale;
- li stabilisce il trasmettitore;

Riguardo al periodo si possono fare le seguenti considerazioni:

- se il periodo del ricevente è minore del trasmittente si hanno vantaggi in termini di performance temporali ma sicuramente vi saranno periodi in cui il receiver andrà in esecuzione senza riscontrare nulla
- se il periodo del ricevente è maggiore del trasmittente sicuramente in ogni periodo il receiver dovrà riscontrare uno o più ack. Tuttavia ciò influirà negativamente sul periodo del trasmittente che dovrà accorciare il suo periodo per assicurare la ricezione dell'ack

Per tale motivo il compromesso migliore è quello di scegliere lo stesso periodo per trasmittente e ricevente. In questo modo i due task saranno sfasati al massimo di un periodo.

I parametri di QoS

Perchè avvenga una comunicazione, vi debbono essere un mittente e

un destinatario per i dati che fluiscono nella rete. Nella terminologia delle reti essi prendono il nome di “client” (C) e “server” (S) rispettivamente. Sia l’applicazione client che quella server forniranno al layer TCP dei parametri di QoS. In particolare il client esprimerà una “domanda” di servizio, mentre il server esprimerà una “offerta” di servizio. Ovvero il client è l'host che invia dati e il server semplicemente conferma la ricezione dei dati.

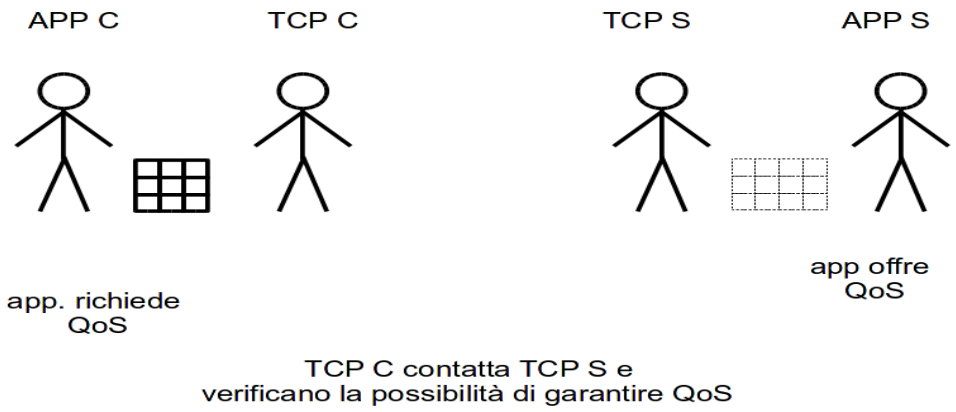

Fig. 13: analogia concettuale del servizio TCP

L’applicazione che fa richiesta di una connessione TCP, fornirà ad esso una serie di parametri di QoS_{KC} . Allo stesso modo l'applicazione ricevente sull'host remoto fornirà a TCP una serie di parametri che definiscono la QoS_{KS} offerta.

In prima istanza si possono fissare i parametri nel modo che segue:

- Q, è la massima quantità di dati che è possibile trasmettere (o ricevere) in un messaggio, nel caso peggiore, in byte. Da tale

quantità è sempre possibile ricavare il numero di segmenti S necessari per inviare Q, tramite la relazione $S=\lceil Q/MSS \rceil$

- T, è il minimo tempo tra la produzione di due messaggi consecutivi da parte dell'applicazione
- D, è la deadline relativa della trasmissione, ovvero l'intervallo di tempo entro il quale l'applicazione desidera avere conferma dell'avvenuta ricezione

La tupla costituita dai tre parametri (S_{KC}, T_{KC}, D_{KC}) definisce il singolo canale trasmissivo visto dall'applicazione che deve inviare dati.

La tupla costituita dai parametri (S_{KS}, T_{KS}) definisce i parametri di QoS dell'applicazione ricevente.

Nel prosieguo supponiamo per semplicità di analisi che $D_{KC}=T_{KC}$.

Ad esempio per un applicazione che intenda inviare 8KB ogni 2s, ipotizzando un MSS di 1KB, la tupla di QoS_{KC} sarà del tipo (8, 2, 2).

Risulta subito evidente che la comunicazione può effettivamente avere successo se e solo se:

$$\frac{S_{KC}}{T_{KC}} \leq \frac{S_{KS}}{T_{KS}}$$

ovvero se la banda richiesta dal client non supera la banda di servizio offerta dal server.

Nella figura che segue è mostrato il modus operandi che si vuole ottenere.

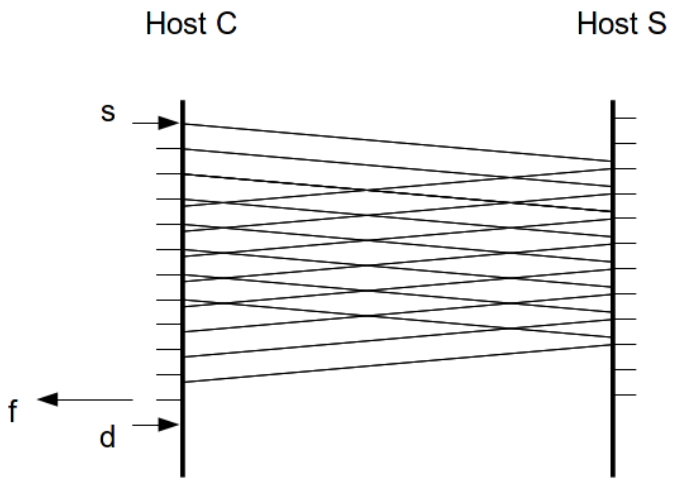

Fig. 14: "modus operandi" periodico

In ogni periodo di attivazione il TCP_C potrà inviare un burst di segmenti deteminato in base alle necessità.

Mapping del QoS nei parametri del task servente

Il mapping del QoS nei parametri del task servente può essere visto come la definizione di un contratto di servizio che l'applicazione richiede al TCP. Tali parametri impongono dei vincoli per il task TCP.

Chiamiamo TCP_C il servente TCP dal lato client e TCP_S il servente TCP dal lato server.

Date le precedenti definizioni, i parametri da determinare per

l'attuazione del contratto sono:

- il periodo di TCP_C (T_C)
- il periodo di TCP_S (T_S)
- la quantità massima di segmenti a dimensione massima che TCP_C deve trasmettere in un periodo (S_C)
- la quantità massima di segmenti a dimensione massima che TCP_S può ricevere in un periodo (S_S)
- il WCET di TCP_C (C_C)
- il WCET di TCP_S (C_S)

Si nota che C_C e C_S saranno funzione del massimo numero di segmenti inviabili nel corso di un unico periodo (ad es. $C_C \propto S_C \cdot C_{pkt}$, dove C_{pkt} è il computation time necessario all'invio di un singolo pacchetto).

Tali parametri sono necessari per stabilire la fattibilità della comunicazione che è determinata principalmente da due vincoli: il fattore di utilizzazione del task sulla CPU definito da $U_{cpu}=C/T$ e l'occupazione di banda della rete definita come $B_{net}=S \cdot MSS/T$.

Calcolo del periodo del TCP servente

Avendo ipotizzato che il TCP_C è un task periodico, come va stabilito il suo periodo?

Dalle richieste dell'applicazione è possibile definire in generale la

banda richiesta al task TCP_C come:

$$\frac{S_{KC}}{T_{KC}} = \frac{S_C}{T_C} \quad \rightarrow \quad T_C = \frac{T_{KC}}{S_{KC}} \cdot S_C \quad (1)$$

Nel caso peggiore, per essere sicuri di riscontrare l'ultimo ack lato trasmittente, possiamo stabilire che il tempo disponibile per l'invio dei dati è dato dalla deadline richiesta a cui va sottratto il WCTT per il transito sulla rete e un periodo del task trasmittente lato server T_S (che abbiamo stabilito essere equivalente a T_C). Indicando con α il numero di periodi del servente TCP_C non utilizzati per l'invio di dati (se non in situazioni di emergenza) ma riservati unicamente alla notifica all'applicazione della corretta ricezione, possiamo stabilire che il tempo disponibile per l'invio è dato da $T_{KC} - WCTT - (\alpha+1) \cdot T_C$, con $\alpha \geq 0$. Ad esempio con $\alpha = 1$ l'ultimo periodo del task servente lato client può essere riservato per segnalare l'arrivo dell'ack.

Sostituendo a T_{KC} nella relazione (1) il tempo realmente necessario disponibile per l'invio $T_{KC} - WCTT - (\alpha+1) \cdot T_C$, si ottiene con semplici passaggi algebrici che il periodo del task trasmittente deve valere:

$$T_C = \frac{(T_{KC} - WCTT) \cdot S_C}{S_{KC} + (\alpha+1) \cdot S_C}, \quad \alpha \geq 0 \quad (2)$$

Da tale relazione si nota che il periodo scelto tiene conto delle richieste dell'applicazione, del burst trasmissivo da inviare in un

periodo e del tempo necessario a transitare lungo la rete.

Calcolo della finestra minima

Affinchè il flusso di tipo pipelining possa avvenire senza che il trasmittente si blocchi, è necessario che la finestra annunciata sia superiore al seguente valore soglia:

$$W_{min} = min\left(MSS \cdot S_{KC}, MSS \cdot S_C \cdot \left(\left\lceil \frac{WCTT + T_C}{T_C} \right\rceil + 1 \right) \right) \quad (3)$$

In tale formula $WCTT + T_C$ rappresenta il tempo necessario prima dell'arrivo dell'ack del burst n-esimo, ovvero l'intervallo di tempo in cui vengono inviati dati prima di ottenerne il riscontro. La divisione di tale valore per T_C rappresenta il numero di periodi in cui il TCP_C tornerà in esecuzione prima di vedersi confermati i dati dell'n-esmo burst (il ceiling impone il vincolo di interezza sul numero di periodi). Viene aggiunta una unità per evitare il blocco (evitare finestra di dimensione zero).

6.3.1 Three Way Handshake

Il three way handshake deve essere quindi esteso per fare in modo che i due TCP si scambino le informazioni necessarie. Nel SYN verrà aggiunta l'opzione QOS_REQUEST dove saranno trasportate le

informazioni di QoS richieste dall'applicazione. Il server può così stabilire la fattibilità della comunicazione e può determinare la finestra iniziale necessaria affinchè il trasferimento dati avvenga senza intoppi. Nel SYN+ACK di risposta verrà aggiunta l'opzione QOS_OFFERED dove saranno indicati i parametri di servizio offerti dal server. In caso di non fattibilità il server potrebbe rispondere con un segmento di reset RST.

Il TCP client con tale SYN+ACK viene a conoscenza della finestra e del RTT. Può così calcolare il suo periodo e il numero di segmenti (burst) da inviare in ogni periodo per soddisfare le richieste dell'apllicazione client. Invia quindi queste informazioni al server con l'opzione SET_PERIOD.

Il server può così istanziare e attivare il task TCP servente che avrà lo stesso periodo del task client. In caso di non fattibilità il server potrebbe rispondere con un segmento di reset RST.

Capitolo 7: Conclusione e sviluppi futuri

In questo capitolo vengono tratte alcune conclusioni e identificati gli sviluppi futuri.

7.1 Conclusioni

E' opinione comune che TCP non sia adatto all'impiego in applicazioni real time, visto che l'obiettivo di tale protocollo è garantire l'affidabilità della comunicazione piuttosto che il rispetto di vincoli temporali. Negli anni sono stati proposti diversi protocolli alternativi al TCP e specifici per applicazioni in tempo reale. Nessuno di essi, però, ha raggiunto la maturità e il grado di accettazione di TCP.

Quando si progettano applicazioni di rete RT spesso si preferisce utilizzare protocolli come UDP per sfruttare il basso overhead imposto da tale protocollo; tale scelta è qui ritenuta erronea in quanto UDP rappresenta solamente una soluzione "di comodo" che non può garantire alcunchè e rimanda tutte le problematiche ai livelli superiori. A nostro avviso invece è il livello di trasporto che dovrebbe farsi carico di determinare i meccanismi per risolvere il problema della comunicazione in tempo reale.

L'analisi effettuata ha voluto mostrare che un sistema di rete real-time può essere definito tale solo se tutti i suoi componenti

collaborano per portare a termine un unico obiettivo generale: garantire le tempistiche sul trasferimento dati. Secondo questa visione non è il protocollo TCP l'anello debole del sistema.

TCP è un protocollo affermato e stabile, implementato nella stragrande maggioranza dei sistemi operativi e largamente accettato nella rete. Implementare un protocollo a parte per applicazioni real-time o, peggio ancora, reinventare la ruota dei servizi TCP utilizzando UDP sembra una scelta poco ingegneristica. Se l'infrastruttura di rete consentisse di riservare la banda e/o permettesse di ricavare un valore di WCTT con piccole fluttuazioni (basso jitter nella spedizione di un pacchetto e ricezione dell'ack), allora a nostro avviso il TCP potrebbe essere opportunamente modificato per essere utilizzato come protocollo di trasporto real time su tale rete.

Si è visto, in prima analisi, come con pochi cambiamenti sarebbe possibile rendere il protocollo ad invio temporizzato senza modificarne la semantica. Così facendo sarebbe possibile rispettare le richieste di QoS delle applicazioni tenendo in considerazione i vincoli temporali richiesti.

7.2 Sviluppi futuri

L'implementazione dello stack di rete TCP/IP sul sistema operativo in tempo reale SHaRK apre la strada a diversi esperimenti sul tema del networking in tempo reale.

Una strada potrebbe essere quella di implementare e testare una serie di estensioni che sono state proposte in letteratura, per verificare la possibilità di aumentare la predicibilità del trasferimento dati anche su una rete non real-time (best-effort).

L'altra strada sarà quella di testare le estensioni al protocollo qui esposte in uno scenario di rete controllata che abbia i requisiti richiesti. Ciò consentirà empiricamente di confermare o confutare la soluzione proposta. Inoltre, in caso di risultati incoraggianti, ciò permetterebbe di migliorare la soluzione (considerando anche l'insieme delle anomalie che possono verificarsi) e valutare tutta una serie di problematiche che qui non sono state indirizzate (come ad esempio il problema della neutralità della rete che si viene a creare in uno scenario in cui la banda sia riservata).

Bibliografia

[SR88] J.Stankovic, K. Ramamritham "Tutorials on Hard Real Time Systems" IEEE Computer Society Press (1988)

[BUT06] Buttazzo Giorgio C. (2006), "Sistemi in tempo reale", Pitagora

[STA88] J.A. Stankovic "Misconceptions about real-time computing" IEEE Computer, October 1988

[GRA76] R.L. Graham "Bounds on performance of scheduling algorithms" in Computer and Job Scheduling Theory, pages 165-227. John Wiley and Sons, 1976

[KUR03] J. Kurose, K. Ross (2003), "Internet e Reti di Calcolatori", McGraw-Hill

[STE94] Stevens Richard (1994), "TCP/IP Illustrated, Volume 1: The Protocols", Addison-Wesley, 1994

[STE95] Stevens Richard (1994), "TCP/IP Illustrated, Volume 2: The Implementation", Addison-Wesley, 1995

[RFC793] J. Postel, "Transmission Control Protocol", RFC-793, September 1981

[RFC1122] Braden R., "Requirements for Internet Hosts - Communication Layers", October 1989

[JAC88] V. Jacobson, M. Karels, "Congestion Avoidance and Control", November 1988

[RFC1071] R.Braden, D.Borman, C.Partridge, "Computing the

Internet Checksum", RFC-1071, September 1988

[RFC2988] V.Paxson, M.Allman, "Computing TCP's Retransmission Timer", RFC-2988, November 2000

[RFC2001] W.Stevens, "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms", RFC-2001, January 1997

[RFC2581] M.Allman, V.Paxson, W.Stevens, "TCP Congestion Control", RFC-2581, April 1999

[POS03] POSIX. IEEE Standard 1003.13-2003, Standard for Information Technology - Standardized Application Environment Profile - POSIX Realtime and Embedded Application Support (AEP). The Institute of Electrical and Electronics Engineers, 2003.

Ringraziamenti

Ringrazio la mia famiglia per il supporto economico, psicologico e morale che ho ricevuto in tutti questi anni. In particolare vorrei ringraziare Sally e Viola, e i nuovi arrivati Gustavo, Lucky e Terry.

Ringrazio i miei colleghi di banco e di studio dell'università; non posso non menzionare i miei tre amici ingegneri specializzati ormai da un anno: Matteo Maccioni, Paolo Magrini e Simone Scaffidi... grazie per avermi aspettato!

Ringrazio i miei amici extra-universitari che hanno riempito i miei momenti di relax con film horror per lo più squallidi, con impervie scalate verso mete mai raggiunte e con pomeriggi estivi passati davanti a forni dalla temperatura minima di 340°.

Ringrazio Chiara, Francesca, Andrea e i ragazzi della Scuola delle Idee che hanno allietato e colorato i pomeriggi di questo mio ultimo anno.

Ringrazio il mio tutor accademico Andrea Claudi per il supporto, per la simpatia, per la competenza e per il suo enorme repertorio di aneddoti!

Ringrazio il prof. Aldo Franco Dragoni per la sua disponibilità, cordialità e per il suo "occhio critico". Inoltre lo ringrazio per la passione e l'entusiasmo che trasmette nelle sue lezioni, le quali mi hanno permesso di apprezzare appieno il fascino del mondo real-time.

Il mio ultimo ringraziamento non può che essere per Alessandra.

Senza di lei non avrei mai potuto essere qui a scrivere quest'ultima pagina: il suo appoggio in questi anni è stato per me fondamentale.