Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

STUDIO COMPARATO DEL PROTOCOLLO DI COMUNICAZIONE SCTP

Tesi di Laurea in Reti di Calcolatori

Relatore:
Chiar.mo Prof.
Fabio Panzieri

Presentata da:
Lorenzo Vinci

Sessione 1
2012/2013

Dedicato a tutte le persone che
mi sono state vicine in questi anni:
i miei genitori e mio fratello,
Mauro, per il continuo affetto
e per avermi sostenuto e rialzato nei periodi piú bui,
Marco, amico paziente e generoso
senza il quale non ce l'avrei fatta,
Luca, che crede in me senza alcuna esitazione,
e tutti i miei amici conosciuti in questi anni:
Michele, Davide, Gabriele, Alessandra, Ketty,
Mariagrazia, Remo, Elena.
Ultimi ma non meno importanti, Simone ed Alessio:
coi quali concluso questa avventura.
Grazie ...

Indice

Introduzione

Al giorno d'oggi i principali e piú usati protocolli di trasporto per le reti sono **TCP**, (Transmission Control Protocol), e **UDP**, (User Datagram Protocol). Come sappiamo, in modo particolare il protocollo TCP investe un ruolo di fondamentale importanza nelle comunicazioni tra calcolatori. Basti pensare a tutto il *World Wide Web*. In questo lavoro, mi propongo di analizzare un altro importante protocollo di comunicazione che fornisce un servizio simile al TCP. Esso é lo: **SCTP**, ***Stream Control Transmission Protocol***.

Il livello di trasporto, nell'architetture ISO/OSI della rete, si basa sul sottostante livello Network (dove viene introdotto l'uso dell'inidirizzo IP) per fornire la trasmissione di dati da un processo su una macchina sorgente ad un altro processo su un'altra macchina destinazione, secondo un livello di affidabilitá desiderato, in modo efficace ed indipendente dalle reti fisiche attualmente in uso. Pertanto, tramite il livello di trasporto, di cui fanno parte il protocollo TCP e SCTP di seguito trattati, fornisce tutte le astrazioni necessarie alle applicazioni per utilizzare la rete. Il software per tale livello di trasporto viene implementato all'interno del *kernel* del sistema operativo, fornendo agli utenti le *system calls* apposite per utilizzarlo. Essenzialmente, si puó notare che il servizio di livello di trasporto é molto simile al servizio del sottostante livello di rete: entrambi forniscono due tipi di servizi di rete, orientati alla connessione e non orientati alla connessione, l'indirizzamento e il controllo di flusso. Perció, é ragionevole chiedersi il perché esistano due livelli distinti. Il codice del livello di trasporto viene eseguito nel esclusivamente nel kernel delle macchine utenti, al contrario di quello del livello di rete che é eseguito non solo negli host della rete ma soprattutto nei nei routers, gestiti dall'operatore di telecomunicazioni. Cosa accade se il livello di rete offre un servizio che causa perdita di pacchetti o malfunzionamenti di tanto in tanto? Tali problemi esistono ed ovviamente gli utenti non hanno alcun tipo di controllo sul livello di rete e dunque l'unica possibilitá per risolvere il problema é quello di inserire un altro livello, tra quello applicativo e quello di rete, che migliori la qualitá del servizio e che sia piú affidabile.

Dunque, l'uso di un livello di trasporto introduce delle chiamate di sistema le quali permettono alle applicazioni di essere indipendenti dalle primitive del servizio di rete che cambiano a seconda del tipo di quest'ultima (WiMax, Ethernet, Wireless...). Nascondere il servizio di rete dietro l'insieme delle relative *system calls* ed all'interfaccia delle *socket* a livello kernel, assicura che un cambiamento di rete fisica non richieda alcuna modifica del codice sorgente dell'applicazione. Grazie al livello di trasporto ed al livelo IP, i programmatori di applicazioni possono scrivere codice utilizzando un insieme di primitive standard ed ottenere programmi funzionanti su reti diversi, senza preoccuparsi della gestione delle diverse interfacce. Perció, possiamo affermare che il livello di trasporto svolge la funzione chiave di isolare i livelli superiori dalla tecnologia, dalla struttura e dalle imperfezioni della rete.

Il protocollo a livello di trasporto piú celebre nonché piú usato (basti pensare che tutta Internet si basa su questo) é **TCP**, (*Transmission Control Protocol*). Esso, in quanto orientato alla connessione e non semplicemente *best effort* ed orientato ai messaggi come UDP, é stato progettato appositamente per fornire un flusso di bytes affidabile end-to-end e che consegni i mesaggi nello stesso ordine di invio, basato su un'interconnessione di rete (*internetwork*) inaffidabile. Infatti un'*internetwork* differisce da una singola rete poiché le diverse parti possono avere tipologia, ampiezza di banda, ritardi, dimensioni di pacchetti ed altri parametri completamente diversi tra loro. TCP é stato progettato per adattarsi dinamicamente alle proprietá dell'internetwork e per continuare ad offrire solide prestazioni in presenza di molti tipi di errore. Ogni computer supporta TCP in una parte del *kernel* che si interfaccia con il livello IP. Ogni entitá TCP accetta dai processi locali i flussi dati dell'utente, li suddivide in pezzi di dimensione non superiore a 64 KB ed infine invia ogni pezzo in un datagramma IP autonomo. I datagrammi contenenti i dati TCP che arrivano ad un computer vengono consegnati all'implementazione di TCP nel *kernel* che ricostruisce i flussi di bytes originali. Inoltre, il livello IP non garantisce la consegna senza errori dei datagrammi e quindi é compito di TCP spedirli abbastanza velocemente da utilizzare la capacitá di trasferimento senza creare congestione e ritrasmettere tutti i pacchetti che non sono stati consegnati oppure sono sono stati affetti da errore. Inoltre i datagrammi potrebbero anche arrivare nell'ordine sbagliato, ed é sempre compito di TCP ri-assemblare i messaggi nella giusta sequenza. Le entitá TCP di invio e ricezione si scambiano i dati sotto forma di segmenti. Un segmento TCP consiste in un'intestazione di fissa di 20 bytes piú una parte per i dati da inviare, che puó essere anche nulla durante lo scambio di segmenti di controllo (come durante l'instaurazione di una connessione TCP). TCP puó accumulare in un segmento i dati provenienti da piú invii (chiamate alla system call *send()*) oppure dividere i dati di un invio in piú

segmenti purché ogni segmento, compresa l'intestazione TCP, sia contenuta nel *payload* IP di 65515 bytes.

Come detto nel preambolo, SCTP é un nuovo protocollo di trasporto che perció si trova allo stesso livello di TCP e UDP nell' architettura *ISO/OSI*. Come TCP, esso é un protocollo *unicast* che offre un servizio di trasporto dei pacchetti dei dati, attraverso la rete, che sia ordinato e sicuro. Ricordiamo che *unicast* significa l'invio di pacchetti che sono destinati ad un solo computer nella rete, noto il suo indirizzo IP. Concettualmente una connessione unicast corrisponde ad una connessione punto-punto, ossia una connessione che si viene a creare tra due terminali della rete (non per forza gerarchizzati unicamente come client o server) per lo scambio di informazioni. Vedremo come SCTP offra l'oppurtunitá che un nodo della rete sia rappresentato da piú di un indirizzo IP.

In generale, possiamo definire SCTP come un protocollo ibrido tra UDP e TCP. Infatti, SCTP é orientato ai messaggi come UDP garantendo inoltre sia il controllo della congestione, l'affidabilitá, l'invio/ricezione nella giusta sequenza nonché il riconoscimeno di pacchetti danneggiati, caratteristiche tipiche di TCP.

Oggi SCTP ha due importanti implementazioni: sia come API C dal kernel Unix 2.6 in avanti, sia nelle nuove API *NIO* disponibili dalla versione 7 di Java.

Attualmente, uno dei maggiori usi che si fa di SCTP é soprattutto in ambito di "sicurezza", cosa che possiamo intendere in piú modi. Infatti, l'implementazione del protocollo detta **S-SCTP** (*Secure SCTP*) fornisce un valido meccanismo per il cosiddetto **AAA**: Autorizzazione, Autenticazione e Accounting. Tuttavia, per sicurezza, nel caso di SCTP, possiamo intendere anche la richiesta di una certa robustezza e capacitá di ri-adattamento della trasmissione nel caso l'interfaccia di rete di un endpoint, con cui si sta comunicando, cada. SCTP, mediante il *Multihoming* e *DAR*, offre la possibilitá che la comunicazione continui, spostandola automaticamente ad una delle altre interfacce di rete funzionanti dell'endpoint, purché esso ne abbia a disposizione.

Dobbiamo inoltre ricordare come la tecnologia **VOIP** (Voice Over IP) stia diventando oggi giorno di uso sempre piú comune. Sono molti infatti gli enti (come aziende, hotel) che si appoggiano su tale tecnologia per le loro comunicazioni telefoniche sia interne che esterne. Vedremo in seguito come due importanti caratteristiche di SCTP, il *Multihoming* ed il *Multistreaming*, possano venire utilizzate efficacemente nelle tecnologie real-time ed in modo particolare nel VOIP.

Infine, non dobbiamo dimenticare il motivo iniziale per il quale é stato definito questo nuovo protocollo: SCTP venne reso standard nel 2000, dal-

l'Internet Engineering Task Force (IETF)Signaling Transport (SIGTRAN) come protocollo affidabile per la consegna di messaggi e segnali telefonici SS7 (**System Signaling No. 7**). SS7 altro non é che una raccolta di protocolli utilizzati per istanziare su scala mondiale la maggior parte delle pubbliche comunicazioni telefoniche commutate sulla rete. L'uso di SCTP per questo scopo é raccolto nella suite di protcolli detta *SIGTRAN* (SIGnaling TRANsport).

In seguito, gli argomenti verranno trattati nel seguente ordine di capitoli:

- **Capitolo 1** Confronto del protocollo TCP con SCTP:
  in questo capitolo verranno trattati gli aspetti fondamentali in cui TCP differisce da SCTP, come l'instauramento delle connessioni, la struttura dei pacchetti nonché il controllo di flusso e congestione.

- **Capitolo 2** Il Multihoming:
  questo capitolo tratterá di quella che, forse, é la caratteristica piú innovativa e importante di SCTP: la possibilitá per un host, nel corso di una comunicazione, di passare da un'interfaccia di rete ad un'altra qualora la prima, per un qualsiasi motivo, vada fuori uso.

- **Capitolo 3** Il Multistreaming:
  parleremo qui della seconda importante innovazione di SCTP, la possibilitá che un'associazione SCTP possa essere multiplexata in flussi (*streams*) di dati logicamente diversi per l'inivio in parallelo di piú dati senza dovere creare ulteriori o multiple connessioni.

- **Capitolo 4** SCTP e sicurezza:
  poiché le moderne tecnologie di sicurezza, come IPSec, vengono difficilmente adattate a SCTP, é stata introdotta una versione sicura del protocollo detta S-SCTP che verrá analizzata in questo capitolo.

- **Capitolo 5** SCTP e VOIP:
  tratteremo infine dell'uso di SCTP su tecnologia VOIP vedendo come tale protocollo si sposi molto bene con tale tecnologia fornendo, grazie al Multihoming, ampio supporto alla mobilitá.

- **Capitolo 6** Conclusioni.

Capitolo 1

Confronto del protocollo TCP con SCTP

Prima di procedere all'illustrazione delle modifiche introdotte dal protocollo SCTP, vogliamo analizzare da vicino le sue similitudini e differenze con TCP, a cominciare dalla strutturazione base dei pacchetti usati per la comunicazione.

1.1 I segmenti TCP e SCTP

Vediamo che ci sono molte somiglianze negli header dei due pacchetti. Prima di tutto, sono indicate le porte del mittente e del destinatario, che sono sempre numeri di 16 bit. In entrambi i protocolli questi due campi vengono combinati con i relativi indirizzi IP per identificare univocamente una connessione TCP/associazione SCTP ed inoltre permettono il multiplexing di diverse associazioni SCTP/connessioni TCP sullo stesso indirizzo IP. Dunque la chiave di demultiplexing di SCTP é analoga alla stessa per TCP: *(srcPort, scrIPAddr, dstPort, dstIPAddr)*.

Subito sotto, in SCTP troviamo il **Verification Tag**. Quest'ultimo viene utilizzato dal ricevente per validare il mittente del segmento. Dunque sostanzialmente serve ad indicare che il pacchetto proveniente dal mittente sia corretto. É un valore che viene settato in maniera random al momento della creazione della associazione SCTP tra i due host ed é mantenuto per tutta la durata della stessa. Se viene ricevuto un pacchetto senza tale tag esso viene scartato come metodo di protezione contro attacchi possibili.

Un compito simile, viene mantenuto dal **Sequence number** presente nell'header del pacchetto TCP. Tale numero ha due ruoli diversi:

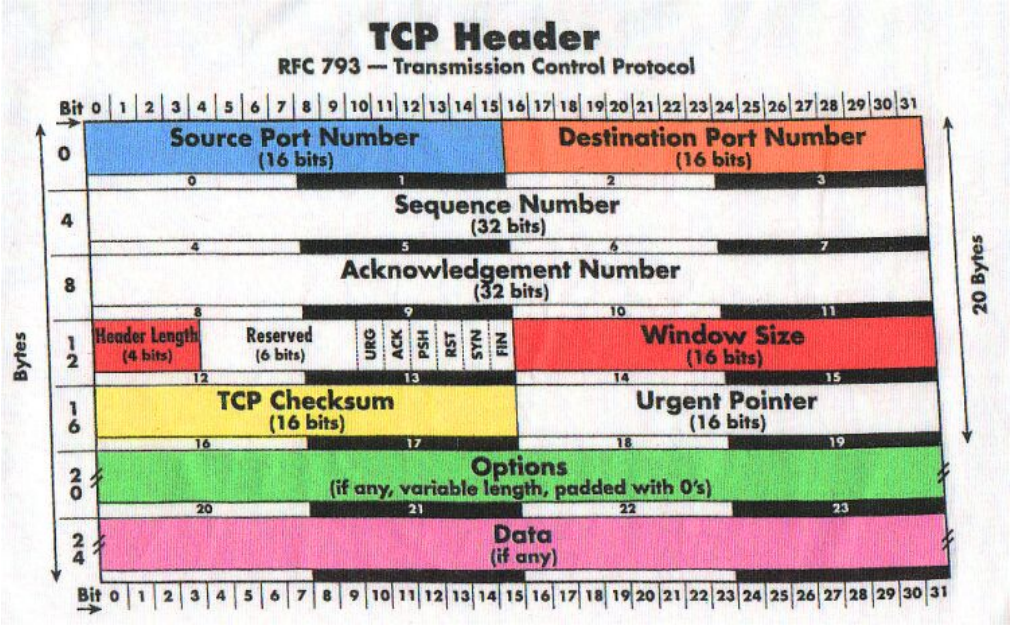


Figura 1.1, il segmento TCP

Bits	Bits 0 - 7	8 - 15	16 - 23	24 - 31
+0	Source port		Destination port	
32	Verification tag			
64	Checksum			
96	Chunk 1 type	Chunk 1 flags	Chunk 1 length	
128	Chunk 1 data			
...	...			
...	Chunk N type	Chunk N flags	Chunk N length	
...	Chunk N data			

Figura 1.2, il segmento SCTP

1. Se il flag SYN é settato a 1 allora ci troviamo nella fase di richiesta inizio connessione e questo numero viene generato casualmente dal mittente e comunicato al destinatario che risponderá a sua volta con un ack ed un suo numero di sequenza.

2. Se, invece, siamo nel corso della comunicazione, *il sequence number* viene incrementato di volta in volta tale che ciascuna delle due entitá

conferma il numero di sequenza ottenuto piú uno. Questo perché il campo **Acknowledgement number** indentifica in ogni momento *il prossimo numero di sequenza che ci si aspetta di ricevere* confermando cosí, implicitamente, di avere ricevuto tutti i numeri di sequenza precedenti.

Entrambi i protocolli, poi, sono muniti di un campo **Checksum** per il controllo dell'integritá sia dei dati che dell'header. L'unica differenza é che in TCP questo valore é di 16 bit, contrariamente a SCTP dove invece ha una dimensione di 32 bit e viene calcolato mediante l'algoritmo CRC32c polinomiale per la rilevazione di errori.

La differenza importante nei due protocolli risiede nel campo **Data**. Nel protocollo TCP tale campo é semplicemente riempito con le informazioni da inviare. Al contrario, in SCTP troviamo un'organizzazione diversa. Infatti dopo ogni intestazione di pacchetto SCTP *Common Header*, troviamo uno o piú **Chunks**.

1.1.1 I Chunks

Ogni chunk ha un formato comune ma il contenuto puó cambiare. Esso puó infatti contenere o informazioni di controllo oppure dati utente, l'uno o l'altro identificato secondo certi campi nell'header.

Di ogni chunk viene fornito il tipo/valore (per indicare cosa esso sta trasportando), alcuni flags (che variano a seconda della natura dei dati trasportati) ed infine la sua lunghezza totale in byte, espressa con un intero a 16 bit. Di solito se tale lunghezza non é un multiplo di 4, il protocollo aggiunge automaticamente degli zeri. É proprio il diverso contenuto di un chunk che ci fa capire se siamo durante la fase di apertura di un'associazione SCTP, se stiamo trasmettendo dati oppure se vi é un errore.

Se i chunks contengono dati utente, ad ognuno di essi viene assegnato un **Transmission Sequence Number** (TSN). Esso é un numero di 32 bit per permettere al ricevente di un'associazione SCTP di mandare un ack per ogni chunk ricevuto correttamente, evitando inoltre di ricevere duplicati.

Chunks multipli, purché contenenti dati utente, possono essere inseriti all'interno dello stesso pacchetto SCTP a condizione che la somma delle loro grandezze resti entro un parametro detto **SCTP MTU** che varia a seconda delle implementazioni.

I tipi di chunk sono indicati dalla tabella sottostante.

Value	Abbreviation	Description
0	DATA	Payload data
1	INIT	Initiation
2	INIT ACK	initiation acknowledgement
3	SACK	Selective acknowledgement
4	HEARTBEAT	Heartbeat request
5	HEARTBEAT ACK	Heartbeat acknowledgement
6	ABORT	Abort
7	SHUTDOWN	Shutdown
8	SHUTDOWN ACK	Shutdown acknowledgement
9	ERROR	Operation error
10	COOKIE ECHO	State cookie
11	COOKIE ACK	Cookie acknowledgement
12	ECNE	Explicit congestion notification echo (reserved)
13	CWR	Congestion window reduced (reserved)
14	SHUTDOWN COMPLETE	Shutdown complete

Tabella 1.1, i tipi di chunk

1.1.2 Frammentazione e riassemblamento dei chunks

Un endpoint coinvolto in un'associazione SCTP puó supportare la frammentazione dei chunks durante la fase di invio (qualora i dati da inviare debbano essere frammentati in piú chunks nello stesso pacchetto). Risulta, ovviamente, obbligatorio che l'endpoint supporti il riassemblamento dei chunks alla ricezione di chunks di tipo dati utente. Nel caso sia supportata anche la frammentazione, l'endpoint mittente deve frammentare l'invio dei dati utente qualora essi siano superiori al parametro MTU sopra indicato. Se questa frammentazione non é supportata e si verifica una quantitá di dati da inviare superiore all'MTU, allora l'endpoint mittente deve generare un errore al livello superiore e non tentare l'invio dei dati comunque.

La frammentazione avviene secondo questi passi:

1. l'endpoint mittente spezza i dati utenti in data chunks in modo che ogni pacchetto SCTP che viene creato (header + data chunks) sia minore del MTU;

2. ad ogni data chunk viene assegnato un TNS in sequenza. Di default sappiamo che invio e ricezione dei pacchetti SCTP sono garantiti nello

stesso ordine. Se a livello utente viene specificato di fare diversamente, per ogni data chunk viene settato un particolare flag, chiamato U, al valore 1;

3. i bit chiami B/E del primo data chunk vengono settati al valore di '10', mentre gli stessi bit dell'ultimo data chunk sono settati a '00'. Si prosegue poi con l'invio.

L'end point ricevente deve dunque essere in grado di riconoscere i frammenti di dati esaminando i bit B/E di ogni chunk i quali vengono poi inseriti in code specifiche e riassemblati. Fatto questo, SCTP ricostruisce i suoi pacchetti raggruppando piú chunks entro l'MTU i quali poi vengono passati all'applicazione utente.

1.1.3 Lo stream di bytes di TCP

A differenza di SCTP (orientato ai messaggi), vediamo come invece l'invio dei dati in TCP possa essere visto come uno *stream* continuo bytes (chiamati *octet* nel RFC). Infatti l'invio di n messaggi a livello utente, non comporta anche l'invio di n pacchetti TCP a livello di trasporto. A differenza di SCTP, il destinatario non riesce a stabilire dove un messaggio inizia e finisce ma é suo proprio compito determinare quando ha letto abbastanza del flusso di bytes perché tale lettura abbia senso per lui e quindi possa ricostruire il pacchetto prima di inoltrarlo all'applicazione. Formalmente, lo *stream* di bytes é un'astrazione, piú precisamente un canale di comunicazione che viene instaurato con una connessione TCP nel quale il viaggio dei dati é bidirezionale. Anche TCP garantisce, mediante il meccanismo dei sequence numbers che vedremo dopo, che l'invio e la ricezione dei dati avvenga nello stesso ordine. Non solo, TCP fa in modo che tale canale sia affidabile mediante la tecnica del *positive acknowledgment with retransmission*.

Dunque, come vengono inviati i dati in TCP?

Generalmente, l'invio di un pacchetto TCP avviene quando il suo bit PSH (PUSH), presente nell'header, viene settato a 1. A livello utente, tale bit puó essere impostato immediatamente a 1 ed in questo caso il pacchetto verrá inoltrato subito al destinatario. In caso contrario, TCP, prima di inviare, aspetta di raccogliere abbastanza dati dall'utente e costruisce i pacchetti (che, ricordiamo, vanno da una dimensione minima di 2 bytes fino a 65535 bytes) dimensionando il campo DATA secondo alcuni parametri. Questo fino a quando non viene segnalato il bit PUSH. A quel punto, si procede con l'invio di tutto quanto si é accumulato.

L'host ricevente dunque, ottiene i pacchetti TCP inviati, mantendenoli in delle code specifiche fino a quando non riceve un pacchetto col flag PUSH

settato. Dunque, il destinatario, smette di aspettare ed inizia ad inoltrare i dati all'applicazione soprastante. Eccezione a questo, é il caso di quando al ricevente arrivano molti pacchetti senza il flag PUSH settato fino ad esaurimento del buffer in entrata. In questo caso, i dati vengono comunque inviati all'applicazione destinaria.

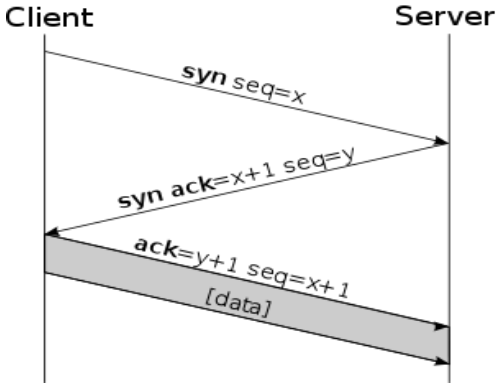
Perció possiamo notare come non vi sia alcuna relazione tra il segnale PUSH e la dimensione dei pacchetti TCP il cui segmento DATA puó derivare da una o piú chiamate di sistema *send()*.

1.2 TCP connection vs. SCTP association

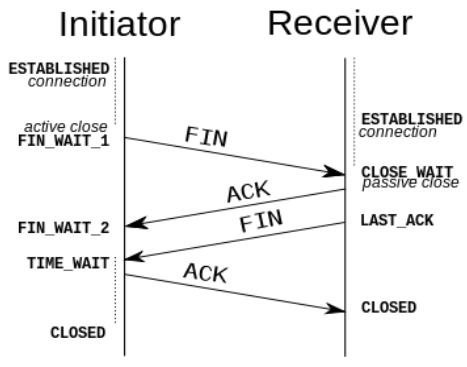
Anche SCTP, come TCP, prevede che prima di poter inviare e ricevere messaggi, sorgente e destinazione debbano prima *accordarsi*. In modo particolare, ad entrambi deve essere nota la porta e l'indirizzo IP su cui comunicheranno e su queste informazioni verrá poi costituito il canale. Gli algoritmi dei due protocolli per stabilire la connessione sono abbastanza simili, sebbene quello per SCTP preveda un ulteriore passaggio di conferma.

1.2.1 L'algoritmo *Three-way handshake* di TCP

Questo algoritmo viene inizializzato da un client che si voglia connettere ad un server. Affinché ció avvenga é necessario che il server abbia prima associato al proprio indirizzo IP una porta sulla quale accettare le richieste di connessione, ció che viene comunemente classificata come *passive open*. A questo punto il client puó effettuare una richiesta di connessione (*active open*) la quale avviene, dal nome dell'algoritmo, secondo 3 passi.



(a) Figura 1.3: Attivazione connessione TCP



(b) Figura 1.4: Shutdown di una connessione TCP

1. **SYN**: il client che vuole connettersi prepara un pacchetto TCP con il flag SYN (usato per indicare richieste di connessione) settato a 1; inoltre, viene generato random un sequence number A col quale inizierá la trasmissione ed anch'esso viene indicato al server.

2. **SYN-ACK**: in risposta, il server risponde con un pacchetto TCP sempre col flag SYN settato e riconoscendo con un ack che ha ricevuto la richiesta di connessione. Questo ack é dato dal sequence number fornito dal client incrementato di 1 (A+1). Inoltre il server genera, per questo pacchetto di risposta, un altro sequence number B. Il server si aspetta che la prossima volta che il client gli manderá un pacchetto TCP esso abbia sequence number (A+1).

3. **ACK**: il client risponde con un ack finale (impostato a B+1) e sequence number (A+1).

I passi 1 e 2 stabiliscono i parametri di connessione (in modo particolare il sequence number) nella direzione client-server, confermati da ack. Viceversa, i passi 2 e 3 stabiliscono i parametri di connessione nel verso server-client. Dunque, tramite questo meccanismo, abbiamo ottenuto una comunicazione full-duplex.

1.2.2 L'algoritmo *Four-way handshake* di SCTP

La creazione di un'associazione SCTP é una delle sue fondamentali caratteristiche in ambito di sicurezza poiché tale associazione viene creata usando anche un meccanismo di cookies. Pertanto, un'associzione SCTP avviene attraverso 4 fasi.

1. **INIT**: il client prepara un chunk di INIT che manda al server. In esso, il client deve settare il proprio *Verification Tag*, un intero positivo a 32bit generato in maniera random. Inoltre é presente anche il campo *Advertised Receiver Window Credit*, (a-rwnd) che indica quanto é grande la finestra in ricezione del destinatario. Dopo l'invio di tale pacchetto, il client setta un timer ed entra in uno stato di attesa detto COOKIE-WAIT.

2. **INIT-ACK**: quando il server riceve il pacchetto sopra descritto, esso deve rispondere immediatamente con un altro pacchetto di risposta INIT-ACK nel quale deve impostare il proprio *Verification Tag* confermando poi quello del client. In questa fase, la cosa piú importante é che il server deve generare uno **State Cookie** secondo una funzione

hash ed in modalitá protetta. Tale cookie deve essere inizializzato al *time-of-day* ed essere dotato di un *lifespan*, quindi la quantitá di tempo durante il quale puó essere considerato valido nella rete. Il pacchetto cosí creato viene inviato al client. A differenza di TCP, il server per ora non alloca nessuna risorsa (in modo particolare nessuna struttura dati di tipo **TCB**, Transmission Control Block) per tale richiesta di associazione rimanendo invece in attesa della prossima fase ad opera del client.

3. **COOKIE ECHO**: il client esce dunque dallo stato di attesa COOKIE-WAIT e semplicemente rimanda indietro il cookie ricevuto nel pacchetto generato nella fase precedente dal server. Ció fatto, imposta un altro timeout ed entra nella fase COOKIE-ECHOED-WAIT.
   N.B.: giá durante questa terza fase il client puó iniziare ad inviare chunks di dati a condizione peró che il cookie rimandato al server sia il PRIMO chunk di tutti. Una volta inviato tale pacchetto ed eventuali chunks di dati, il client deve restare in attesa della conferma finale del server e per il momento non puó inviare altri dati.

4. **COOKIE ACK**: alla ricezione del medesimo cookie inviato prima, il server stavolta costruisce un TCB per tale associazione passando in modalitá EFFETTUATA (*Established*). Dunque il server procede con l'estrarre eventuali dati che il client aveva giá inviato, e risponde con un pacchetto detto COOKIE-ACK. Esso puó contenere anche eventuali dati di risposta a condizione che, come sopra, il cookie ack sia il primo di tutti i chunks. Quando il serer riceve il cookie ack, esce dallo stato COOKIE-ECHOED-WAIT passando anch'esso in quello *Established*.

   Notiamo dunque che i pacchetti *INIT* e *INIT ACK* sono gli unici che non possono contenere chunks di dati.

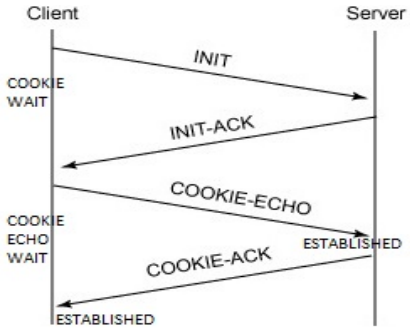


Figura 1.5, 4-ways handshake di SCTP

1.2.3 Osservazioni

Per come é strutturato l'algoritmo di richiesta associazione SCTP, esso costituisce uno strumento potente contro gli attacchi *Syn Flooding*, un tipo di attacco della famiglia Denial Of Service. Infatti, dopo la richiesta di inizio connessione TCP da parte del client, il server TCP, in attesa dell'ack finale del client, alloca risorse per tale connessione rimanendo poi in attesa dell'ack finale che, in caso di attacco, non arriverá mai. L'attacco infatti *Syn Flooding* consiste nell'inviare in quantitá massiccia di pacchetti TCP durante la fase SYN (facendoli sembrare provenienti da indirizzi IP fasulli) facendo in modo che il server allochi risorse per ognuno di essi, fino ad esaurimento delle stesse.

Questa situazione, non puó verificarsi con SCTP poiché il server non alloca risorse fino a quando il client non ha risposto re-inoltrando il corretto cookie da lui generato. In questo modo, qualora si dovesse verificare un attacco massiccio di richieste associazioni SCTP, il server invierebbe i cookies che genera ad inidirizzi IP falsi (generati ed inseriti nel pacchetto INIT dall'host che attacca) i cui veri host ignorerebbero tale cookie in quanto non hanno richiesto alcuna associazioni SCTP. Perció, non verificandosi alcun echo del cookie, il server rimane protetto e non é soggetto a Denial Of Service per sovraccarico.

1.2.4 Chiusura delle connessioni/associazioni

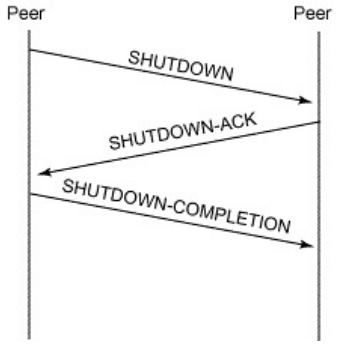


Figura 1.6, Shutdown di un'associazione SCTP

Nel caso della chiusura associazione SCTP, la situazione si rovescia rispetto a TCP poiché quest'ultimo usa un algoritmo in 4 fasi (Figura 1.4) contro quello a 3 fasi di SCTP (Figura 1.6).

Lo shutdown di una connessione TCP prevede lo scambio tra client e server di una coppia data da:

- un pacchetto FIN, ossia un pacchetto TCP dove l'omonimo flag dell'header é settato a 1;

- un ack della corretta ricezione del pacchetto FIN.

Dunque, il primo endpoint che vuole chiudere la connessione, invia il pacchetto FIN all'altro il quale, appena ricevutolo, risponde con un ack e, a sua volta, con un suo pacchetto FIN. Il primo endpoint che voleva chiudere la connessione invia a sua volta un ultimo ack all'altro e prima di chiudere il canale definitivamente aspetta un certo timeout, durante il quale la porta locale rimane non si puó riutilizzare in quanto potrebbero arrivare ancora pacchetti ritardatari riservati all'ormai chiusa connessione. N.B.: questo puó causare le cosiddette connessioni *half-open* nella quale solo una delle due estremitá ha chiuso la connessione mentre l'altra sta ancora leggendo o scrivendo. L'endpoint che ha terminato la connessione dunque non puó piú inviare e nemmeno ricevere mentre l'altro sí. Questo non capita invece quando si termina un'associazione SCTP.

La chiusura di un'associazione SCTP viene fatta in modo sicuro solo quando il programma utente la richiede esplicitamente, attraverso quindi un apposita primitiva/system call. Questo fa in modo che eventuali pacchetti SCTP rimasti in sospeso vengano inviati subito al client prima che l'associazione sia distrutta. Quando tale primitiva viene invocata, l'host entra in uno stato detto *SHUTDOWN-PENDING* durante il quale rimane in attesa degli ultimi pacchetti (finché tutti non sono stati segnalati da ack) e non accetta piú dati dal livello applicativo soprastante. Quando questa attesa termina, si passa alla chiusura vera e propria dell'associazione. La procedura prevede lo scambio di tre pacchetti con un chunk vuoto, dove troviamo solo dei valori particolari nel campo *Chunk Type*. Il procedimento é il seguente:

1. il client che vuole terminare l'associazione invia il pacchetto con il campo Chunk-Type settato al valore 7 (SHUTDOWN). Questo pacchetto indica al server che il client é pronto a chiudere la connessione. Inoltre tale pacchetto contiene l'ultimo TSN dell'ultimo chunk ricevuto dal client. Il punto forte di tale procedura, é che dopo avere inviato tale pacchetto, il client entra nello stato *SHUTDOWN-SENT* facendo partire un timeout. Se tale timeout finisce senza avere ricevuto il pacchetto di risposta dal server (punto 2. seguente) il pacchetto é re-inviato ed il timeout riparte evitando cosí connessioni half-open. Dunque non vi é alcuna chiusura non sincronizzata col server. Tuttavia, questo tentativo di ritrasmissione del pacchetto SHUTDOWN puó essere fatto un numero limitato di volte superato il quale SCTP segnala a livello applicativo che il server non é raggiungibile.

2. il server riceve la richiesta di fine associazione dal client ed entra nello stato di *SHUTDOWN-RECEIVED*. Dunque, smette di accettare nuovi dati dal client e verifica tramite il TSN che tutti i pacchetti siano stati ricevuti dal client. Nel caso, invece, vi siano pacchetti in sospeso il server continuerá la normale trasmissione dei dati fino a che tutti i chunks dati siano correttamente notificati.

   Nel caso capiti quando appena detto, il client (che, ricordiamo, si trova nello stato di *SHUTDOWN-SENT*) deve immediatamente rispondere con ack per ogni chunk dati appartenente ad ogni pacchetto ritardatario ricevuto, re-inviando poi il pacchetto SHUTDOWN e facendo ripartire il timeout.

   Quando non ci sono piú pacchetti in sospeso, il server infine risponde inviando un pacchetto col campo Chunk-Type settato al valore 8 (SHUTDOWN ACK) ed entrando nello stato di *SHUTDOWN-ACK-SENT*;

3. il client che lo riceve termina la procedura non impostando piú alcun timeout ed inviando un ultimo pacchetto col campo Chunk-Type settato a 14 (SHUTDOWN COMPLETE). Infine vengono rimossi tutti i record e strutture dati dell'associazione. A lato server, quando viene ricevuto il pacchetto SHUTDOWN COMPLETE si controlla di essere nello stato SHUTDOWN-ACK-SENT per quella associazione. In caso affermativo, le strutture dati dell'associazione vengono distrutte ed essa é definitavamente chiusa; in caso contrario, il pacchetto é ignorato.

1.3 Consegna affidabile e in ordine dei pacchetti

Abbiamo detto che una delle piú notevoli somiglianze di SCTP con TCP, é il fatto che anch'esso gestisce una consegna in ordine dei pacchetti assicurandosi anche che non contengano errori attraverso meccanismi di *damage data detection* in seguito al quale i pacchetti vengono eventualmente re-inviati. In questo contento anche in SCTP entra in gioco il concetto di *stream*. Tuttavia, per *stream SCTP* dobbiamo intendere una sequenza di messaggi utente che devono essere consegnati al livello applicativo in ordine rispetto ad altri messaggi facenti parte dello stesso *stream*. (vedremo piú avanti come sia possibile avere piú *streams* in contemporanea grazie all'innovativo *Multistreaming SCTP*).

1.3.1 Ordine e affidabilitá in SCTP

Ad ogni messaggio utente dello *stream* passatogli dall'applicazione, il protocollo SCTP assegna un **Stream Sequence Number** (SSN) prima di essere incapsulato dentro un chunk e quindi inviato. É proprio tramite questo SSN che SCTP assicura che, a lato ricevente, i messaggi di uno *stream* vengano consegnati in sequenza.

Spostandoci ad un livello superiore, a livello di pacchetto SCTP, ad ognuno di essi il protocollo assegna un **Transmission Sequence Number** (TSN) e tale numero é indipendente rispetto a qualsiasi SSN assegnato a livello di stream. L'host destinazione deve dunque notificare tutti i TNS ricevuti con un ack, anche se vi sono gap nella sequenza in modo che il servizio di consegna affidabile sia mantenuto funzionalmente separato dal concetto di stream. L'importanza dell'ack sta alla base, come in TCP, della trasmissione affidabile poiché, se entro un timeout il mittente non ha ricevuto tale ack, il pacchetto viene re-inviato. Tale ack puó essere o il numero cumulativo TNS dell'ultimo data chunk ricevuto, oppure un ack speciale detto **SACK**.

L'aknowledgement SACK

SACK, é un pacchetto speciale SCTP il cui tipo di data chunk ha il valore di 3. Questo chunk é inviato ad un endpoint per notificarlo che ha ricevuto i vari DATA chunk o per informarlo che ci sono dei *gap* nella sequenza di DATA chunk ricevuto, come indicato dai loro TSN. Dunque il pacchetto SACK deve contenere il l'ack cumulativo TSN nonché l'a-rwnd. Per definizione l'ack cumulativo TSN, (chiamato anche *CUMACK*), é l'ultimo numero di sequenza TSN ricevuto prima che si sia verificata una rottura nella sequenza dei TSN ricevuti. Il seguente TSN non é ancora stato ricevuto dall'endpoint che ha inviato il SACK.

La figura sopra ci mostra come é composto un pacchetto SCTP il cui chunk é una notifica SACK. Notiamo che ogni pacchetto di questo tipo deve contenere l'ack cumulativo dell'ultimo TSN ricevuto prima che si sia verificata una rottura della sequenza. Dunque il (Cumulative-TSN-ack+1) é il numero del primo pacchetto mancante. In seguito troviamo numero dei pacchetti mancanti (*Number of gap blocks*) nonché dei duplicati. Ognuno dei blocchi indicato nel Number of gap blocks, notifica una sotto-sequenza di TSN ricevuti che seguono la rottura avvenuta. Per definizione quindi il numero contenuto in tale campo é maggiore uguale al *Cumulative TSN Ack*. Vi é poi il *Number of Duplicate TSNs* che indica il numero di TNSs duplicati che il destinatario ha ricevuto. Ogni TSN duplicato é inserito sotto come una lista di chunks contenenti tale numero. Infine notiamo un'altra lista, la

+	Bits 0 - 7	8 - 15	16 - 31
0	Chunk type = 3	Chunk flags	Chunk length
32	Cumulative TSN ACK		
64	Advertised receiver window credit		
96	Number of gap ACK blocks = N		Number of duplicate TSNs = X
128	Gap ACK block #1 start		Gap ACK block #1 end
...	...		...
96+N*32	Gap ACK block #N start		Gap ACK block #N end
128+N*32	Duplicate TSN #1		
...	...		
96+N*32+X*32	Duplicate TSN #X		

Figura 1.7, Struttura del SACK packet

Gap Ack Blocks che contiene gli omonimi blocchi identificati dal loro TNS, incrementando via via il cumulative TSN ack. Tale lista é ovviamente lunga tanto quando indicato nel Number of gap blocks.

Indichiamo infine il *Duplicate TSN*, un numero a 32 bit indicante il numero di volte che un TSN é stato ricevuto dopo l'ultimo SACK inviato. Ogni volta che l'host destinatario riceve un duplicato TSN, esso lo inserisce nella lista dei duplicati prima di inviare il SACK, dopo il quale il contatore dei duplicati viene re-inizializzato a zero.

Esempio

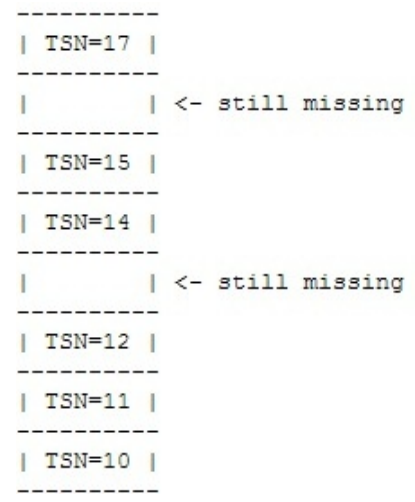


Figura 1.8, Esempio di ricezione con chunks mancanti

Supponiamo che l'host destinatario abbia ricevuto un pacchetto SCTP i cui data chunks sono come in Figura 1.8. Analizzandolo, l'host destinatario invierá un sack al mittente di questa forma:

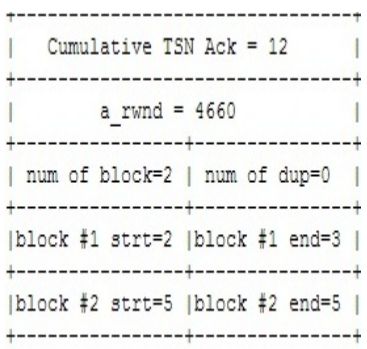


Figura 1.9, SACK inviato in seguito alla situazione in Figura 1.8

In linea con quanto detto sopra, in tale chunk SACK é indicato:

- il TSN dell'ultimo chunk ricevuto prima della rottura: 12;
- il numero di TSN mancanti, che sono due: il 13 ed il 16;
- chunk per chunk é indicato l'offset di dove inizia e finisce il gap per questa sequenza di chunks in modo che si possa calcolare il TSN dei pacchetti mancanti per poi poterli ritrasmettere.

1.3.2 Ordine e affidabilitá in TCP

TCP offre gli stessi servizi di consegna in ordine ed affidabile di SCTP. La differenza sostanziale sta nel fatto che TCP utilizza un sequence number associato non ad un segmento di dati ma ad ogni byte dello *streaming* in modo che l'informazione, al ricevente, possa essere ricostruita in ordine a discapito della frammentazione, disordine durante la trasmissione e perdita di pacchetti. Dunque per ogni byte trasmesso nel payload (il campo DATA del pacchetto TCP), il sequence number, generato durante lo *Three-way handshake*, viene incrementato.

Anche TCP utilizza uno schema di *ack cumulativo*, tramite il quale il destinatario dichiara di aver ricevuto tutti i dati precedenti al sequence number notificato. Dunque, il mittente setta il sequence number in funzione del primo byte del payload ed il destinatario dovrá rispondere con il prossimo sequence number che si aspetta di ricevere. Se questo avviene, tutti i dati sono stati ricevuti. Per esempio, supponendo che il mittente invii un payload di 4 bytes il cui primo byte ha sequence number 100, se il destinatario

manda un ack con sequence number uguale a 104 significa che ha ricevuto correttamente tutti i 4 bytes (100, 101, 102, 103).

Dopo che TCP ha trasmesso un pacchetto, di quest'ultimo viene inserita una copia in una *lista di ritrasmissione* e viene avviato un timeout. Quando arriva l'ack per quel pacchetto, esso viene rimosso dalla lista. Se l'ack non é ricevuto prima dello scadere del timeout, il pacchetto é re-inviato.

Tuttavia, per motivi di efficienza, anche TCP prevede un SACK, impiegato nel caso di perdita di uno o piú pacchetti facenti parte di una grossa quantitá di dati. Infatti esso evita di ritrasmettere l'intero messaggio e i suoi pacchetti (cosa che avverrebbe con un solo sistema di ack accumulativo, in quanto non é in grado di dichiarare la parte che é stata correttamente ricevuta) ma solo il/i pacchetto/i mancante/i. Il SACK informa il mittente che non sono stati ricevuti dati in modo continuo ma si sono verificati dei gap. Il contenuto di un SACK é una lista di *SACK blocks*. Ogni blocco di questo tipo é delimitato dal sequence number di inizio e di fine di un range dello *stream* che il destinatario ha correttamente ricevuto.

Ad esempio, se si ha una quantitá di dati di 1000 byte che vengono inviati con 10 pacchetti TCP, supponiamo che il primo pacchetto venga perso. Il destinatario allora manderá un SACK ack con sequence numbers 100 e 999, indicando che dal byte di sequence number 100 in avanti ha ricevuto correttamente. Il mittente, ottenendo questo tipo di ack, provvederá quindi al solo re-invio del primo pacchetto, dal byte 0 al 99.

1.4 Message Boundary

Abbiamo giá detto che SCTP é un protocollo orientato ai messaggi (come UDP), TCP che invece é orientato allo *stream continuo* di bytes. Questo ha un notevole impatto sul mantenimento o meno del *Message Boundary*, (lett. Confine del messaggio). Infatti, in TCP i limiti del messaggio utente non sono preservati, nel momento in cui essi viaggiano tra sorgente e destinazione.

Ad esempio, se il mittente TCP invia due messaggi AAA e BBB, non c'é alcuna garanzia che i messaggi arrivino in maniera distinta, ma possono essere inviati frammentati o anche l'uno mischiato con l'altro: ad esempio in 3 pacchetti come A, AAB, BB. Dunque, questo significa che é richiesto a livello utente di dare un delineamento dei messaggi, ossia specificare al ricevente la lunghezza del messaggio e quanto leggere. Dunque occorre implementare una scrittura di messaggi che abbiano sempre la stessa lunghezza. In questo modo il programma TCP destinatario puó riconoscere senza alcun dubbio quando l'intero messaggio é stato ricevuto.

In contrasto, SCTP offre un'esplicita demarcazione dei messaggi. Ogni messaggio infatti é consegnato con un'unica operazione di scrittura/lettura, alleggerendo di tale compito l'applicazione utente. Pertanto, due diversi messaggi utente non verranno mai inseriti nello stesso pacchetto SCTP.

1.5 Segnale di *Keep alive*

Generalmente un messaggio (o segnale) di *keep alive* é un meccanismo, sostenuto da uno dei due end point, che si occupa di sondare la connessione verso l'altro end point qualora essa risulti in stato inattivo. Riguardo TCP, attualmente é oggetto di controversia la decisione se tale servizio sia da implementare a livello applicativo invece che a quello di trasporto poiché in quest'ultimo caso esso sprecherebbe inutilmente una notevole quantitá di banda. Contrariamente gli sviluppatori di SCTP sostengono che la verifica continua di raggiungibilitá di un end point che partecipa all'associazione sia di fondamentale importanza, in modo particolare quando SCTP viene impiegato in una *Rete telefonica SS7* dove é richiesto che un eventualmente problema di non raggiungibilitá sia risolto subito. Quando parliamo di keep alive, introduciamo il concetto di *heartbeat*, (lett. *battito vitale*) che altro non é che un timeout scaduto il quale, nel caso non si abbiano avuti ulteriori riscontri, un end point si preoccupa di stabilire se la connessione con l'altro é ancora funzionante.

1.5.1 Segnale *heartbeat* nei due protocolli

Riguardo al protocollo SCTP, di default il segnale di keep alive é impostato con un timeout di 30 secondi dopo il quale l'associazione tra i due end points é considerata inutilizzata e perció un end point manda all'altro, con cui é associato, un particolare pacchetto SCTP il cui data chunk é di tipo *HEARTBEAT* (tipo n. 4) in cui inserisce il tempo attuale nel quale tale chunk viene inviato. Se il mittente di tale segnale non riceve un ack specifico (data chunk di tipo n. 5, *HEARTBEAT ACK*) entro un tempo ragionevole, (che viene calcolato sulla base del Round Trip Time impiegato, per esempio, durante le fasi del *Four Way Handshake*), allora viene incrementato un certo parametro e poi viene ritrasmesso il pacchetto heartback. Se queste ritrasmissioni superano un numero di volte stabilito durante l'implementazione di SCTP, l'associazione viene dichiarata come inattiva a livello applicativo e quindi chiusa.

Il protocollo TCP, a differenza del precedente, non prevede un keep alive heartbeat obbligatorio, lasciandolo invece come opzionale. Infatti il docu-

mento ufficiale RFC di TCP non menziona alcun tipo di segnale keep alive il quale viene invece discusso nel RFC 1122 che tratta degli standard di comunicazione tra host nella comunitá Internet. Dunque, anche TCP puó prevedere l'invio di pacchetti keep-alive allo scadere di un timeout di libera scelta, il quale peró deve essere maggiore uguale di 2 ore. Dunque le implementazioni TCP che prevedono tale servizio, per verificare se una connessione non utilizzata é ancora attiva, inviano un particolare segmento sonda TCP senza campo DATI il quale fa in modo che il ricevente invii un ack confermando cosí che la connessione é ancora in vita. Se, invece, il mittente riceve un pacchetto TCP col flag RST settato a 1, esso significa che la connessione é stata annullata e nel campo DATA, eventualmente, é possibile ricavarne le cause.

1.6 Controllo della congestione

L'ultimo argomento che mi propongo di trattare, analizzando le principali differenze e similitudini dei due protocolli, é il controllo della congestione. Vedremo che questa é la caratteristica piú comune tra TCP e SCTP. Quest'ultimo infatti viene detto *TCP-friendly* in merito a questa caratteristica proprio per il fatto che i suoi meccanismi di controllo congestione sono molto simili a quelli di TCP.

1.6.1 Cosa si intende per controllo delle congestione di rete?

Uno dei compiti principali della rete, consiste nella corretta allocazione delle risorse tra tutti gli host che ne fanno parte. Essenzialmente le risorse che tutti gli elaboratori condividono sono due:

- l'ampiezza di banda delle linee di collegamento
- i buffer dei routers/switch dove i pacchetti sono accodati in attesa di essere trasmessi in una linea di collegamento.

Capita peró che troppi pacchetti competano per la medesima linea fino ad esaurimento banda e/o dei buffer dei routers. Perció l'unica soluzione é il liberare le risorse di forza bruta scartando ed eliminando pacchetti. Se tale fenomeno diventa molto frequente significa che la rete é *congestionata*. Fortunatamente la maggior parte delle reti prevede certi meccanismi di controllo della congestione, sia da parte degli host che dagli stessi apparati di rete (nei routers). Tale controllo influisce sulla velocitá con cui le sorgenti possono

inviare pacchetti nel tentativo, in primo luogo, di evitare che la congestione si verifichi o di eventualmente aiutare ad eliminarla. Ed é proprio in questo senso che entrano in scena i meccanismi implementati nei due protocolli.

1.6.2 Controllo di congestione in TCP

Il protocollo TCP ipotizza che tutti i router della rete usino una politica FIFO (*First In, First Out*, la piú comunemente usata) per l'instradamento ed invio dei pacchetti che arrivano nelle loro code. La strategia di TCP consiste nel cominciare ad inviare pacchetti secondo certe quantitá e, in seguito, reagire agli eventi osservabili che accadono, diminuendo o incrementando la quantitá di dati inviati. In parole povere, il protocollo TCP affida ad ogni sorgente il compito di determinare quanta sia l'ampiezza di banda disponibile e regolarsi di conseguenza con l'invio dei pacchetti. TCP basa i suoi meccanismi di controllo congestione sugli ack poiché la mancata ricezione di un di un ack per un pacchetto significa quasi sempre che esso é stato eliminato dalla rete in quanto congestionata. Dato che TCP usa gli ack per controllare la velocitá di trasmissione dei pacchetti, si dice che esso é *self-clocking* (autotemporizzato). La congestione in TCP é monitorata tramite tre meccanismi.

Aumento additivo/diminuzione moltiplicativa

Prima di spiegare in cosa consiste questo metodo, dobbiamo avere chiaro il concetto di *Advertised Window*. Quando due elaboratori instaurano una connessione TCP, il destinatario comunica al mittente una certa dimensione di dati che esso é in grado di ricevere (utilizzando quindi un algoritmo di *sliding window* a dimensione variabile, inserendola di volta in volta nell'header del pacchetto TCP nell'apposito campo *AdvertisedWindow*. Di conseguenza, il mittente si deve limitare fino al punto di avere, in ogni momento, un valore massimo di bytes, non confermati da ack, minore o uguale al valore di *AdvertisedWindow*. Generalmente questo algoritmo viene detto di *Congestion Avoidance*, ossi é un procedimento che cerca, letteralmente, di evitare la congestione qualora essa venga percepita. Questo si differenzia rispetto ai due metodi seguenti poiché essi, al contrario, cercano di aiutare a eliminare o contenere ae congestione.

Il protocollo TCP gestisce, per ogni connessione, una nuova variabile di stato chiamata *Congestion Window* che verrá usata dalla sorgente per limitare la quantitá dei propri dati in viaggio nella rete in un certo istante. Tale finestra gioca un ruolo simile all' *Advertised Window* nel controllo della congestione. Infatti TCP pone che il massimo numero di byte non ancora

confermati sia il valore minimo tra l'*Advertised Window* e la *Congestion Window*. In questo modo la sorgente non puó spedire piú velocemente di quanto sia accettabile per il destinatario. Tuttavia, come puó la sorgente determinare il giusto valore di questa nuova variabile? Essa deve essere impostata basandosi sul livello di congestione della rete percepito dalla sorgente, diminuendone il valore quando la congestione é alta e viceversa. La sorgente riesce a determinare che la rete é congestionata proprio per il fatto che si iniziano a verificare scadenze delle temporizzazioni per la ricezione degli ack (come spiegato nel paragrafo 1.6.2) fenomeno interpretato da TCP come mancata consegna di pacchetti poiché scartati in seguito a congestione. TCP si regola in modo che il valore della *Congestion Window* venga di volta in volta dimezzato ogni volta che scade una temporizzazione. Tuttavia, é necessario anche che il valore della *Congestion Window* venga anche re-aumentato per sfruttare le nuove risorse liberate nella rete.

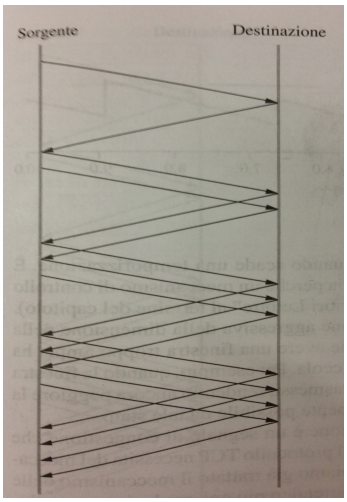


Figura 1.10, Fase dell'aumento additivo

Come ci suggerisce la Figura 1.10, ogni volta che la sorgente invia con successo (poiché confermata da ack) una quantitá di pacchetti pari al valore di *CongestionWindow*, allora quest'ultimo valore viene incrementato di un numero di bytes per l'equivalente di un pacchetto. Pertanto questo andamento, fatto di continui aumenti e diminuzioni della finestra di congestione, continua per tutta la durata della connessione.

Partenza lenta

Il meccanismo sopra descritto risulta efficace quando la sorgente si trova ad operare in condizioni prossime alla capacitá disponibile della rete ma

richiede troppo tempo per fare decollare la trasmissione dei dati. Il protocollo TCP quindi usa un secondo meccanismo chiamato ironicamente *partenza ritardata/lenta* che viene usato per aumentare la finestra di congestione esponenzialmente anziché linearmente.

La sorgente inizia impostando il valore della finestra di congestione ad un solo pacchetto. Quando arriva la conferma ack per tale pacchetto, TCP incrementa di 1 il valore *CongestionWindow* ed invia 2 pacchetti. Appena ricevuti i due ack, TCP incrementa ancora il valore della finestra di congestione di 1 per ogni ack ricevuto. Il risultato finale é che TCP, in pratica, raddoppia il numero di pacchetti in viaggio a ogni intervallo di tempo RTT (*Round Trip Time*, il tempo impiegato da un pacchetto a raggiungere la destinazione piú quello dell'arrivo dell'ack).

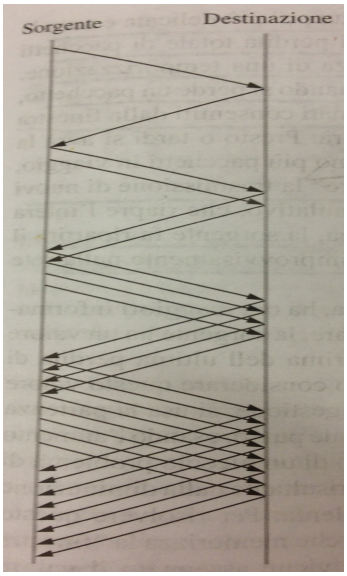


Figura 1.11, Pacchetti in viaggio durante la partenza lenta

Il nome di questo metodo deriva dal fatto che se il mittente, appena stabilita la connessione, inviasse subito una quantitá di pacchetti pari all'*Advertised Window*, sebbene ci possa essere un'ampiezza di banda disponibile, potrebbe accadere che nei buffer dei routers non vi sia spazio abbastanza per assorbire questa improvvisa mole di pacchetti. Perció si inizia con il trasmettere un pacchetto, ed incrementarne poi via via la quantitá, piuttosto che inviare all'improvviso una quantitá di pacchetti che riempia l'intera finestra. Ci sono due situazioni nel quale si fa uso di tale meccanismo:

1. all'inizio di una connessione, quando la sorgente non ha ancora alcuna idea in merito a quanti pacchetti si possa permettere di mantenere in

viaggio nella rete;

2. quando si verifica una situazione di perdita totale dei pacchetti, detta *dead connection*, mentre la sorgente é in attesa della scadenza di una temporizzazione.

In particolare, per quanto riguarda il punto 2, dobbiamo ricordare come funziona l'algoritmo di *sliding window* di TCP: quando si perde un pacchetto prima o poi la sorgente arriva ad un punto in cui ha inviato tutti i dati consentiti dalla finestra e si blocca in attesa di un ack che non arriverá mai e questo provoca una non stimolazione per la trasmissione di nuovi pacchetti. Perció la sorgente fa ripartire il flusso di dati usando la partenza lenta piuttosto che scaricare improvvisamente nella rete una quantitá di dati da riempire di nuovo la finestra.

Ritrasmissione e recupero veloce

La grossolana implementazione delle temporizzazioni in TCP puó provare lunghi periodi di tempo durante i quali la connessione va incontro a fenomeni di stagnazione, in attesa che scada una temporizzazione. Perció é stato aggiunto al protocollo TCP un nuovo meccanismo chiamato *ritrasmissione veloce* che innesca la ritrasmissione di un pacchetto perduto prima che scada la normale temporizzazione. L'idea di fondo é la seguente: ogni volta che un pacchetto dati arriva a destinazione, il ricevente risponde con una conferma anche se quest'ultima porta un numero di frequenza che é giá stato confermato. In questo modo quando arriva un pacchetto fuori sequenza, che quindi non puó essere confermato da TCP poiché non sono ancora arrivati i pacchetti precedenti, allora il ricevente rispedisce la stessa conferma spedita la volta precedente: questa seconda trasmissione della conferma si chiama *ACK Duplicato*. Quando il mittente vede un ack duplicato, sa che la destinazione ha ricevuto un pacchetto fuori sequenza, da cui si deduce che un pacchetto precedente é andato perduto. Tuttavia, dato che tale pacchetto potrebbe anche solo essere stato ritardato dalla rete, la sorgente aspetta di ricevere un numero ack duplicati (in TCP questo numero é 3) prima di rispedire il pacchetto mancante. Per farci un'idea migliore, osserviamo la figura 1.12 sottostante:

N.B.: per semplificare l'esempio, ragioniamo in termini di pacchetti e non di singoli numeri di sequenza per ogni singolo byte. Osserviamo che la sorgente invia subito i pacchetti 1,2,3 e 4 ma il terzo viene scartato dalla rete. Di conseguenza, quando il destinatario vede arrivare il pacchetto 4 senza prima avere ricevuto il 3, allora invia un ack duplicato per il pacchetto 2 e lo fará di nuovo con i pacchetti 5 e 6. Solo quando la sorgente vedrá che l'ack per

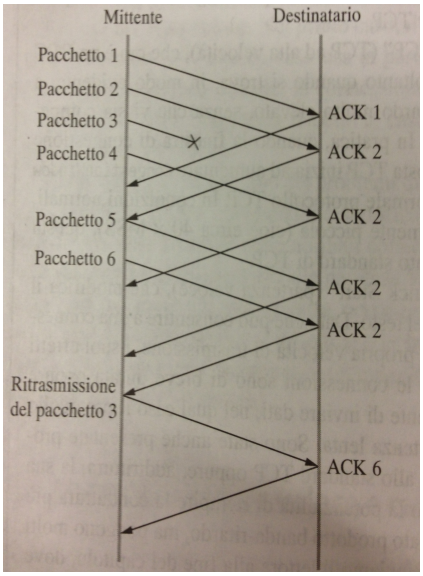


Figura 1.12, Ritrasmissione veloce basata su ack duplicati

il pacchetto 2 si é ripetuto 3 volte, allora provvederá alla ritrasmissione del pacchetto 3. Notiamo che poi la destinazione risponderá inviando un unico ack cumulativo per tutto quanto ricevuto fino al pacchetto 6. In questo modo, quindi, spariscono i lunghi periodi di tempo durante i quali la finestra di congestione rimaneva costante e non venivano spediti pacchetti. Questo porta ad un miglioramento del *throughtput* di TCP del 20%, eliminando circa la metá delle scadenza delle temporizzazioni di una tipica connessione TCP.

Infine, in questo meccanismo é stato implementato un miglioramento notevole detto *recupero veloce*. Quando la ritrasmissione veloce segnala una congestione, invece di riportare la finestra di congestione al valore unitario e quindi riprendere con la partenza lenta, si possono usare i pacchetti di conferma ack, che si trovano ancora in viaggio, per pianificare l'invio di pacchetti. Questo accorgimento elimina la fase di partenza lenta che avviene nel momento in cui la ritrasmissione veloce rileva la perdita di un pacchetto e quello in cui inizia l'aumento additivo. In altre parole, la partenza lenta viene usata solo all'inizio della connessione oppure quando si ha la scadenza di una temporizzazione; in tutte le altre occasioni, la finestra di congestione segue lo schema dell'aumento additivo e diminuzione moltiplicativa.

1.6.3 Controllo di congestione in SCTP

Come TCP, anche SCTP possiede 3 variabili di controllo stato per regolarsi con l'inivio di dati durante un'associazione. Esse sono:

- *receiver advertised window size*, (rwnd, in bytes), la quale viene settata dal destinatario sulla base della sua disponibilitá in buffer in ricezione;

- *congestion control window*, (cwnd, in bytes), la quale viene di volta in volta settata dalla sorgente sulla base di come percepisce le condizioni di rete;

- *slow-start threshold*, (ssthresh, in bytes), che viene usata dal mittente per distinguere tra una fase di partenza lenta ed una di prevenzione di congestione.

Tuttavia SCTP richiede un'ulteriore variabile di stato, detta *partial-bytes-acked* (pba), la quale viene usata durante la prevenzione della congestione per ottenere un miglior settaggio possibile della cwnd sopra indicata. Come vedremo in seguito, un'importante innovazione di SCTP é il *Multihoming*, ossia la possibilitá che un host sia raggiungibile sulla rete da piú di un indirizzo IP, uno per ogni sua interfaccia. Dunque, SCTP prevede che venga mantenuta una tripletta composta da {*cwnd, ssthresh e partial-bytes-acked*} per ogni indirizzo IP destinazione, se la destinazione é *multihomed*. Al contrario, viene mantenuta una sola rwnd per ogni associazione.

Partenza lenta

L'iniziare la trasmissione di dati in una rete di cui non sa ancora le condizioni (o eventualmente dopo un periodo abbastanza lungo di inattivitá), richiede che SCTP sondi prima la rete per determinarne la capacitá disponibile. L'algoritmo della partenza serve proprio a questo scopo, sia durante l'inizio della trasmissione sia per correre ai ripari quando viene identificata la perdita di pacchetti. Questo a differenza di TCP, il quale potrebbe iniziare anche utilizzando l' "Aumento additivo/diminuzione moltiplicativa". Vengono eseguite le seguenti fasi:

- il valore iniziale della cwnd, prima di iniziare a trasmettere o dopo un lungo periodo di inattivitá, viene settato a 2*MTU (ricordiamo che MTU sta per *Maximum Transmission Unit* ed indica la dimensione massima di un pacchetto SCTP);

- allo scadere di una temporizzazione, la cwnd viene settata al valore di 1 MTU prima della ritrasmissione;

- il valore iniziale di ssthresh puó essere arbitrariamente grande e di solito viene settato alla stessa dimensione dell'advertised window del ricevente;

- ogniqualvolta il valore di cwnd é maggiore di zero, ad un endpoint é permesso avere al massimo il valore di cwnd bytes in sospeso nella trasmissione;

- quando il valore di cwnd é minore o uguale al ssthresh, un endpoint su SCTP deve usare l'algoritmo di partenza lenta (a differenza di TCP) per potere poi incrementare il valore della cwnd. Quest'ultima viene incrementata se e solo se valgono due condizioni:

  1. l'ack entrante ha un valore maggiore dell'ack cumulativo (quindi del valore totale di dei data chunks appena notificati precedentemente in sospeso);
  2. l'intero valore di cwnd era in uso prima dell'arrivo dell'ack.

  Dunque, la cwnd viene incrementata del valore minimo tra l'MTU del destinatario e quello della grandezza dei dati appena notificati. Questo, a differenza di TCP, permette di fare in modo che con SCTP si evitino attacchi di tipo *ACK-Splitting* che si verifica quando un endpoint destinatario invia ack al mittente non per ogni messaggio ricevuto ma per ogni piccola porzione di esso, provocando un aumento spropositato della cwnd del mittente (che l'aumenta per ogni ack ricevuto) e facendogli cosí violare il controllo di congestione.

- Infine, quando un endpoint non trasmette dati su un dato indirizzo IP, la cwnd per quell'indirizzo deve essere re-settata col valore massimo tra (cwnd/2) e (2*MTU) per ogni RTO (*Retransmission Time-Out*). In sostanza la perdita di un pacchetto causa il dimezzamento di cwnd.

Prevenzione della congestione

SCTP utilizza il metodo della prevenzione della congestione quando il valore di cwnd é maggiore del valore ssthresh. L'idea di fondo di questo metodo é che il valore di cwnd sia incrementato di 1 MTU per ogni RTT (*Round-Trip Time*) ed é proprio per facilitare questo meccanismo che viene usata la variabile di stato pba. Vengono quindi eseguiti i seguenti passi:

- la variabile pba é settata a 0;

- all'arrivo di ogni SACK, che sorpassa il valore cumulativo del TSN Ack Point (CUMACK), allora viene incrementata la pba di un valore pari al numero totale dei bytes dei chunks appena notificati da tale SACK. La cwnd é incrementata di 1 MTU quando valgono due condizioni:

1. il valore di pba é maggiore o uguale di cwnd;
2. l'intera cwnd era in uso prima dell'arrivo del SACK (ossia il mittente aveva un numero di bytes in sospeso maggiore o uguale di cwnd.

Il pba viene poi re-settato a (pba - cwnd).

- Quando poi tutti i dati inviati dal mittente sono stati notificati dal ricevente, allora il pba é resettato a 0.

Rilevamento della perdita pacchetti

Come in TCP, anche SCTP utilizza due metodi per rilevare la perdita dei pacchetti ma con qualche piccola differenza.

1. Algoritmo di ritrasmissione veloce: la differenza con TCP sta nel fatto che tale procedura é avviata quando si verifica una sequenza di 4 SACK indicanti un TSN mancante (e non dopo 3 ack ripetuti come in TCP). Infatti, quando la destinazione rileva che vi é un buco nella sequenza, comincia a mandare SACK col TSN perduto fino a quando il buco non é riempito. Dunque la sorgente si comporta nel modo seguente:
   - marca i vari chunks da ritrasmettere;
   - aggiusta il valore della cwnd dell'indirizzo IP del destinario (se multi-homed) in cui é avvenuta la perdita, secondo la formula cwnd = max((cwnd/2), (2*MTU));
   - determina quanti dei DATA chunks marcati per la ritrasmissione possano entrare dentro ad un pacchetto SCTP singolo secondo il valore di MTU della destinazione e lo invia;
   - fa partire un timer solo se tale endpoint sta ri-trasmettendo di nuovo alla destinazione il primo DATA chunk in sospeso.

2. Retransmission Time-Out (RTO): la regola generale per questo tipo di meccanismo é che per ogni destinazione che ha dei data chunks in sospeso (ancora non notificati), deve essere mantenuto un timer di retrasmissione (RTO, calcolato in proporzione al RTT). Quando tutti i dati, rispediti alla destinazione vengono notificati, allora il time-out é fermato. Nel caso invece il time-out giunga a scadenza, ne viene impostato uno nuovo raddoppiato, i dati in sospeso per quella destinazione vengono di nuovo rimarcati come da rispedire e se per caso il ricevente é multi-homed, viene selezionato un indirizzo IP alternativo a cui inviare di nuovo i DATA chunks. Si procede dunque con l'inviare di nuovo quei DATA chunks. Il nuovo valore della cwnd é drasticamente ridotto a 1 MTU.

Capitolo 2

Il Multihoming

Dopo avere analizzato in cosa il protocollo SCTP é comune e differisce dal TCP, iniziamo adesso a dare uno sguardo a quelle che sono le caratteristiche innovative di tale protocollo a cominciare dall'ormai piú volte citato ***Multihoming***.

2.1 Introduzione al Multihoming

Semplicemente, possiamo dire che il Multihoming consiste nel fatto che un personal computer possieda diverse interfacce di rete (Ethernet, Wireless etc.), ognuna con un proprio indirizzo IP e che tale host possa utilizzare tutte queste interfacce assieme per essere indirizzato e raggiungibile nella rete. Quando uno, o entrambi gli endpoint che partecipano all'associazione SCTP, possiedono diversi indirizzi IP, allora ogni host stabilisce tra questi un indirizzo primario da usare per la trasmissione di DATA chunks. Inoltre, per tale scopo, viene scelto anche un secondo indirizzo di riserva in caso ritrasmissione di DATA chunks a causa di qualche problema della rete nel percorso per raggiungere l'indirizzo IP primario (ad esempio, potrebbe esserci congestione). Altri ed eventuali indirizzi IP, verranno utilizzati per scopo di ridondanza.

La figura 2.1 ci mostra quale sia l'idea di fondo del Multihoming. Notiamo quindi che entrambi gli host che partecipano all'associazione SCTP, devono fare un *bind()* della stessa porta con due o piú indirizzi IP diversi. Questo é il motivo principale per il quale in SCTP non si parla di connessioni, quanto piuttosto di associazioni. In TCP, infatti, una connessione é un unico canale di comunicazione tra due endpoints (una socket, tra una sola interfaccia di rete sorgente ed una destinazione) mentre un'associazione SCTP é un modo di raggruppare assieme in un tutt'uno diversi percorsi di raggiungibilitá per

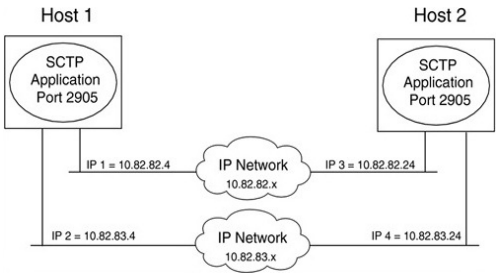


Figura 2.1, configurazione base del Multihoming

entrambi gli host che vi partecipano. Tutti questi indirizzi IP sono scambiati e testati durante la fase di INIT dell'associazione ed ognuno di essi é considerato un percorso alternativo al relativo endpoint.

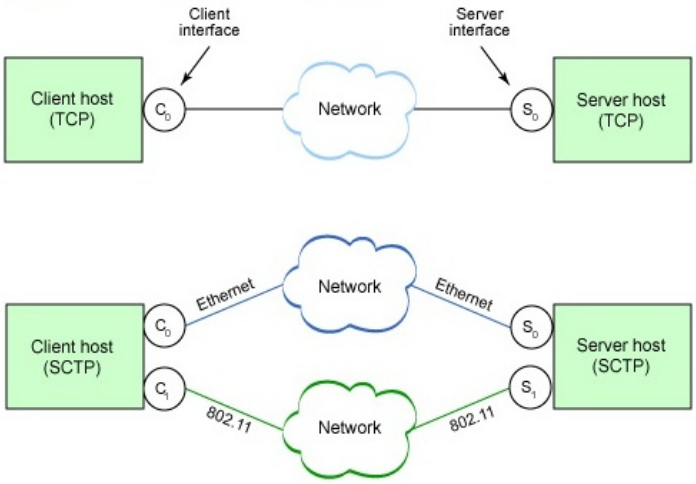


Figura 2.2, connessione TCP contro associazione multi-interfaccia di SCTP

2.2 Vantaggi del Multihoming

Il scopo maggiore del Multihoming é il fatto che riesce a dare una possibilitá di sopravvivenza dell'associazione molto maggiore, in presenza di problemi e/o errori di rete. Infatti, il Multihoming é stato introdotto per lo scopo principale di aumentare la robustezza e la sicurezza di applicazioni che richiedono alta disponibilitá di rete. Differentemente, in una sessione convenzionale (single-homed), un problema di accesso ad una LAN locale puó portare al completo isolamento di un end-system mentre un vero e proprio problema nel cuore della rete puó causare l'impossibilitá momentanea di tra-

sportare pacchetti, almeno fino a quando i routers non si sono riconfigurati con un percorso alternativo. Usando SCTP in multihomed, invece, le varie LANs in ridondanza configurate possono essere utilizzate per rinforzare l'accesso locale. Infatti, usando indirizzi diversi, si puó forzare il processo di routing affinché passi attraverso linee diverse. Tuttavia, in questa forma, il Multihoming di SCTP non é stato pensato per *load sharing* (bilanciamento della quantit'ị dati da trasportare) tra i vari indirizzi IP, ma solo per scopi di sicurezza e ridondanza. Dunque, il continuo fallimento di inviare DATA chunks verso l'indirizzo primario, porta SCTP a considerare valido solo il secondario almeno fino a quando un segnale di *HEARTBEAT* notifica di nuovo il corretto funzionamento dell'indirizzo IP primario. La ritrasmissione verso un indirizzo IP secondario non influenza peró il conteggio totale dei DATA chunks in sospeso. Infatti é previsto che ci sia un contatore di messaggi in sospeso per ogni indirizzo IP usato nell'associazione ed entrambi vengono correttamente aggiornati ogni volta che é necessario ritrasmettere i messaggi per pacchetti rimasti in sospeso. Inoltre, quando il percorso primario diventa inattivo ma l'utente cerca comunque di forzare l'invio dei DATA chunks all'indirizzo IP primario, SCTP, oltre a riportare un errore a livello applicazione, é tenuto comunque a provare l'invio utilizzando uno degli altri percorsi, se essi esistono.

2.3 Supporto al Multihoming

Per supportare il Multihoming, i due endpoint SCTP si scambiano tutti i loro indirizzi IP disponibili durante la fase iniziale, inserendoli all'interno quindi del chunk INIT. Ogni endpoint deve essere continuamente in grado di ricevere dati da tutti gli indirizzi IP associati e la raggiungibilit'a di questi ultimi é verficata tramite l'invio periodico di segnali *HEARTBEAT*. Inoltre, é anche possibile che certi sistemi operativi utilizzino gli indirizzi disponibili in modalitá round-robin, nel qual caso quindi la ricezione di messaggi da interfacce diverse risulta il comportamento standard. In ogni caso, quando un server é associato con un client multihomed, esso dovrebbe provare a ritrasmettere chunks di prova a tutti gli altri indirizzi IP usati dal client, per verificarne la corretta accessibilitá. Tuttavia, di default, un mittente SCTP dovrebbe inviare i DATA chunks all'indirizzo IP primario a meno che non venga specificato diversamente a livello utente dell'applicazione.

Inoltre, é obbligatorio che certi chunks di risposta particolari vengano inviati al solo indirizzo IP del mittente che li ha scatenati. Per esempio, quando un server SCTP riceve un chunk INIT da un client per iniziare un associazione, allora il server deve rispondere col chunk INIT ACK inviandolo all'in-

dirizzo IP primario che si trovava nell'header del pacchetto INIT del client. Altri chunks che devono seguire questa regola sono SACK o l'HEARTBEAT SACK.

Infine, é importante notare che quando un endpoint destinazione, connesso ad un server multihomed, riceve DATA chunks duplicati, allora sarebbe utile cambiare l'indirizzo di invio del SACK utilizzando uno degli altri indirizzi del server (diverso quindi dall'indirizzo IP mittente primario contenuto nel DATA chunk inviato dal server). La ragione di questo é che ricevere un duplicato da un endpoint server potrebbe indicare che il percorso di ritorno per il SACK non é disponibile e quindi bisogna comunicare i duplicati al server per vie alternative.

2.4 Dynamic Address Reconfiguration

Un'estensione importante di SCTP nel contesto del Multihoming é il cosiddetto **DAR** (*Dynamic Address Reconfiguration*). Essenzialmente il DAR rende dinamicamente possibile l'aggiunta o l'eliminazione di indirizzi IP e la richiesta di un cambiamento del percorso primario durante una sessione SCTP attiva.

Al giorno d'oggi é possibile estendere le interfacce di rete di un computer molto facilmente (per esempio inserendo gli ormai noti modem/chiavetta usb forniti da operatori di telefonia mobile). Inoltre, sappiamo che i provider possono ri-enumerare la rete dinamicamente, associando al nostro host un altro indirizzo IP. Per fare in modo che il sistema riesca ad interagire con la nuova configurazione, spesso l'associazione a livello di trasporto deve essere riavviata. Possiamo immaginare come per molte applicazioni tutto questo significhi in una temporanea interruzione la quale, ovviamente, non é gradita. Dunque, tale estensione DAR permette di:

- aggiungere dinamicamente un indirizzo IP all'associazione;
- eliminare dinamicamente dall'associazione un indirizzo IP non piú usabile;
- richiedere all'altro endpoint associato che cambi il suo percorso primario all'indirizzo IP appena aggiunto.

Il tutto, ovviamente, senza alcuna modifica o riavvio dell'associazione la quale, dunque, continua a funzionare nonostante modifiche di rete dell'host su cui si trova. Quindi, distinguiamo due procedimenti diversi a seconda se si deve aggiungere o eliminare un indirizzo IP dall'associazione.

2.4.1 Aggiunta ed eliminazione dinamica di un indirizzo IP all'associazione

Per implementare queste nuove funzionalitá, sono stati introdotti nuovi tipi di chunks di controllo:

- un pacchetto SCTP per comunicare le modifiche degli indirizzi IP;
- un pacchetto SCTP che funge da ack rispetto al precedente.

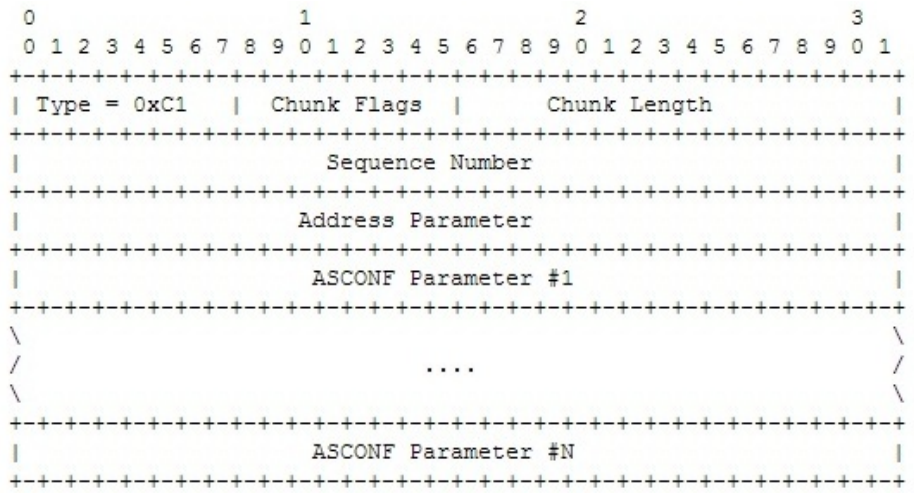


Figura 2.3, chunk di richiesta di modifica nell'associazione

La figura 2.3 ci mostra il chunk di controllo che un endpoint manda all'altro (per la precisione all'indirizzo IP primario attuale) per comunicargli che vuole effettuare delle modifiche agli indirizzi IP coinvolti nell'associazione. Come sempre, vi troviamo un *Sequence Number TSN* il cui valore é compreso tra 0 e $(2^{32}-1)$ mentre nell'*Address Parameter* troviamo l'indirizzo IP dal quale proviene tale chunk, indirizzp IP che deve appartenere all'associazione in corso e quindi prima dichiarato. Infine, nei campi denominati *ASCONF Parameter* troviamo i vari chunks con le informazioni di controllo che trasportano. I chunks sono della forma in Figura 2.4 e, a seconda del messaggio di configurazione che portano, avranno un *Parameter Type* tra quelli presenti in Tabella 2.1.

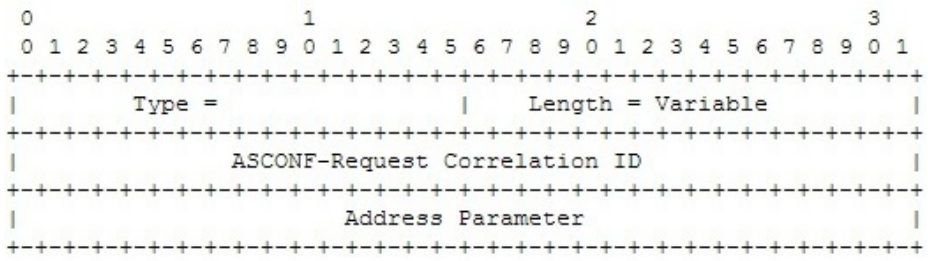


Figura 2.4, struttura di un chunk per modifiche degli indirizzi IP per un'associazione

Address Configuration Parameters	**Parameter Type**
Add IP Address	0xC001
Delete IP Address	0xC002
Set Primary Address	0xC004
Error Cause Indication	0xC003
Success Indication	0xC005

Tabella 2.1, i tipi di Address Parameters in DAR

Perció distinguiamo in 3 casi:

1. per ogni indirizzo IP che si vuole **aggiungere**, ci sará un chunk di tipo *Add IP Address* e nel cui campo *Address Parameter* sará scritto l'indirizzo IP da aggiungere all'associazione;

2. per ogni indirizzo IP che si vuole **eliminare** dall'associazione, ci sará un chunk di tipo *Delete IP Address* nel cui campo *Address Parameter* sará inserito l'indirizzo IP da dis-associare;

3. se, infine, si vuole **settare un altro indirizzo IP come primario**, verrá inserito un chunk di tipo *Set Primary Address* nel cui campo *Address Parameter* troviamo il nuovo indirizzo da settare come *primary path*.

Affinché tutte le operazioni sopra descritte vadano a buon fine, il ricevente di un pacchetto che ha apportato modifiche, deve poi inviare un ack al mittente, detto *Address Configuration Acknowledgment Chunk*, il cui header é uguale a quello del pacchetto in Figura 2.3 (se non per il campo Type che per l'ack ha valore 0x80). Invece, tale pacchetto ack trasporta tanti chunks quanti quelli inviati dal pacchetto di configurazione, ognuno che si riferisce al rispettivo chunk del pacchetto di configurazione. Ogni chunk avrá *Parameter*

Type settato a *Success Indication* se la corrispondente operazione é andata a buon fine oppure, viceversa, *Error*. In quest'ultimo caso troviamo il campo *Error Cause Indication* dove é riportato l'errore verificatosi.

Se tale ack, non dovesse giungere al mittente che richiedeva le modifiche, allora quest'ultimo non potrá apportarne di ulteriori fino a quando non riceve l'ack ma dovrá invece impostare un time-out, al termine del quale rimanderá il pacchetto iniziale.

2.5 Uno sguardo alle primitive per il Multihoming

Vogliamo ora presentare una piccola panoramica di uso del Multihoming fornendo i passi principali di una piccola applicazione client-server, sfruttando l'implementazione del protocollo SCTP disponibile nel Kernel Linux (¿= 2.6) e le relative API.

Cominciamo con osservare l'applicazione a lato Server. Prima di tutto é necessario, come sempre, creare una socket *listening*:

```
int sock = socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
```

dove, il parametro SOCK_SEQPACKET indica che la nostra socket non é di tipo SOCK_STREAM (per TCP) e nemmeno SOCK_DGRAM (per UDP) ma proprio una socket che gode delle proprietá di entrambe, come sappiamo essere SCTP. Il terzo parametro poi, IPPROTO_SCTP, indica che usiamo questa socket per il nuovo protocollo SCTP. Come sempre poi ci serviamo di una struttura dati di tipo **sockaddr_in** dove indichiamo la porta su cui il server sará in ascolto ma soprattutto specifichiamo che vogliamo ricevere da TUTTE le nostre interfacce di rete tramite il parametro INADDR_ANY. Diremo in questo caso, infatti, che il server SCTP che stiamo creando é **one-to-many**, proprio per il fatto che possiede multiple interfacce di rete con cui comunicare, a differenza del classico stile **one-to-one** per cui abbiamo il parametro *SOCK_SEQPACKET*.

```
struct sockaddr_in addr;
addr.sin_family = AF_INET /* famiglia di indirizzi IPv4 */
addr.sin_addr.s_addr = htonl(INADDR_ANY);
addr.sin_port = NOSTRA_PORTA;
```

Come detto precedentemente, il Multihoming si avvale di un pacchetto HEARTBEAT, per verificare che ognuna delle altre interfacce di rete sia attiva e funzionante, e di un time-out RTO allo scadere del quale, se non si é ricevuto un ack dall'interfaccia principale del destinatario, si passa ala

secondaria (questo nel caso ci connettessimo ad un client anch'esso *multi-homed*. Questi parametri li impostiamo con due strutture dati apposite, rispettivamente **sctp_paddrparams** e **sctp_rtoinfo**.

```
struct sctp_paddparams heartbeat;
struct sctp_rtoinfo rtoinfo;

heartbeat.spp_flags = SPP_HB_ENABLE; /***/

/* Ogni 5000 millisecondi vengono effettuati i controlli
di ogni altra interfaccia */
heartbeat.spp_hbinterval = 5000;

/* se dopo 2000 millisecondi scade il time-out,
si passa alla seconda interfaccia */
rtoinfo.srto_max = 2000;
```

N.B. alla linea segnata con /***/: occorre settare questo flag altrimenti l'impostazione di ogni quanti millisecondi mandare l'HEARTBEACK non andrá a buon fine. Infatti, per fare in modo che la socket listening creata abbia tutte queste caratteristiche, eseguiamo una serie di chiamate alla system call *setsockopt()*.

```
/* Impostiamo l'heartbeat */
setsockopt(sock, SOL_SCTP, SCTP_PEER_ADDR_PARAMS, &heartbeat,
           sizeof(heartbeat));

/* Impostiamo l'RTO massimo */
setsockopt(sock, SOL_SCTP, SCTP_RTOINFO, &rtoinfo, sizeof(rtoinfo));
```

Seguiranno poi le solite chiamate alle system calls *bind()* e *listen()* e prima di iniziare a ricevere messaggi possiamo verificare che il multi bind é avvenuto su tutte le interfacce, mediante un array di puntatori alla **struct sockaddr_in** e passandolo, assieme alla socket listening, alla primitiva di protocollo **sctp_getladdrs()**. Quest'ultima funzione, come possiamo intuire, restituisce nell'array tutti gli indirizzi IP attivi del host locale che possiamo poi stampare a schermo con un *ciclo for*.

```
struct sockaddr_in *laddr[6]
int addr_count = 0;
int i

addr_count = sctp_getladdrs(sock, 0, (struct sockaddr**)laddr);
```

```
for(i=0; i<addr_count; i++) {
     printf("Address number %d: %s:%d\n",
                i,
                inet_ntoa((*laddr)[i].sin_addr),
                (*laddr)[i].sin_port
           );
}
```

Rimane solo un'ultimissima considerazione, prima di iniziare a ricevere messaggi. A livello di programmazione, troviamo anche la struttura dati **struct sctp_event_subscribe**. Per verificare il corretto funzionamento del Multihoming, dobbiamo attivare un'opzione della socket che ci permetta di catturare i cosiddetti *SCTP EVENTS*. Questi ultimi possono essere considerati come segnali di un avvenuto cambiamento nell'associazione SCTP che possono essere notificati e gestiti con un adeguato *event_handler()*. Nell'esempio mostrato vedremo come un client invii messaggi ad un server il quale é avviato con interfaccia primaria di tipo Wireless e come, in seguito alla disattivazione di quest'ultima, la ricezione dei dati continui ma passando in modalitá cablata. Dunque per prima cosa, attiviamo la *cattura degli eventi* (la quale ha bisogno di una struttura dati apposita detta **struct sctp_event_subscribe**) sulla socket con:

```
struct sctp_event_subscribe event;
setsockopt(sock, IPPROTO_SCTP, SCTP_EVENTS, &event,
            sizeof(struct sctp_event_subscribe)
           );
```

Quando si verifica un evento, la sua notifica viene spedita assieme ai chunks di messaggi. Quindi, ogni volta che riceviamo un messaggio occorre verificare se esso contenga notifiche di eventi. A questo punto il nostro server é pronto a ricevere e dunque procediamo con:

```
while(1) {
 int flags = 0;
 int res = sctp_recvmsg(sock, buffer, BUFFER_SIZE,
                        NULL, 0, NULL, &flags
                       );
 if(flags & MSG_NOTIFICATION)
        handle_event(buffer);
}
```

dove *sctp_recvmsg()* é una primitiva di SCTP che permette di sfruttare le sue nuove caratteristiche. Per controllare se si é verificato un evento,

basta confrontare *flags* con la costante *MSG_NOTIFICATION*. Se il risultato é positivo, allora si é verificato un evento ed occorre passare il *buffer* ad un apposito gestore da noi implementato che, convertendo il buffer in una struttura dati di tipo **sctp_notification**, ne estrae il campo che determina il tipo di evento. Nel nostro caso, se l'evento si rileva di tipo **SCTP_PEER_ADDR_CHANGE** allora significa che vi é stata una variazione dell'interfaccia di rete su cui avviene la comunicazione.

Nelle istantanee seguenti, durante la trasmissione del client al server, viene disattiva l'interfaccia Wireless il cui evento é rilevato dal client che lo comunica al server facendo cosí continuare la trasmissione via cavo Ethernet.

Per prima cosa, il server ci mostra all'avvio un listato delle sue interfacce disponibili, in ordine rispettivamente: Local, Ethernet e Wireless, tutte in ascolto sulla stessa porta.

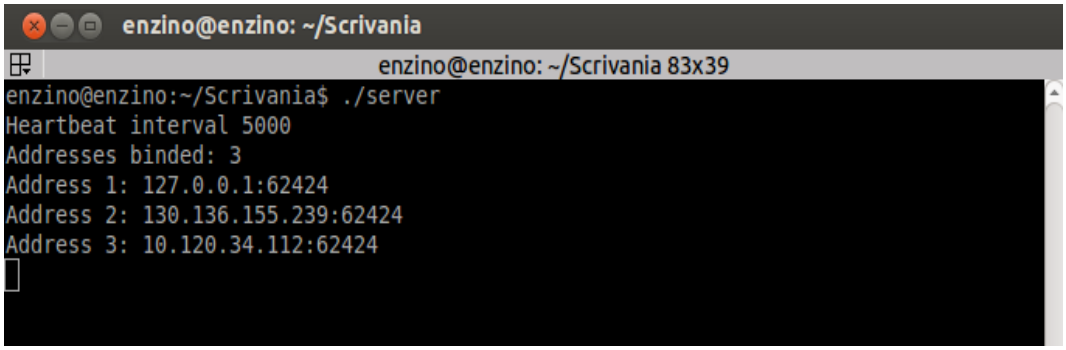


Figura 2.5, avvio Server SCTP

Subito dopo lanciamo il client, dandogli come indirizzo primario per contattare il server quello wireless. Anche il client ci mostra un listato delle interfacce disponibili del server correttamente rilevate esclusa, ovviamente, quella locale. Il client puó fare questo nel modo seguente:

1. creare una socket con indirizzo IP primario del server;
2. chiamando la system call *connect()* su tale socket;
3. a connessione accettata, chiamare la primitiva SCTP *sctp_getpaddrs()* la quale preso in input un array di puntatori a **struct sockaddr_in** poi lo riempie con tutte le informazioni riguardanti gli indirizzi IP del server partecipanti all'associazione. Tale funzione restituisce il numero di questi indirizzi IP;
4. di ogni **struct sockaddr_in** cosí allocata stamparne poi il campo *sin_addr* con un *ciclo for*.

E qui di seguito, il codice:

```
#define PORT_SERVER 62424

char[16] address;

/* Prendiamo l'IP primario del server in input */
strncpy(address, argv[1], 15);


int sockCli;

/* Array dove salvare le interfacce di rete del server*/
struct sockaddr_in *paddrs[6];


/* Strttura dati per l'indirizzo primario del server */
struct sockaddr_in addrServPrime;
addrServPrime.sin_family = AF_INET;
inet_aton(address, &(addrServPrime.sin_addr));
addrServPrime.sin_port = PORT_SERVER;

/* Connessione al server */
connect(sockCli, (struct sockaddr*)&addrServPrime,
        sizeof(struct sockaddr)
       );

/* Rilevamento delle interfacce di rete del server */
int addr_count = sctp_getpaddrs(sockCli, 0,
                                (struct sockaddr**)paddrs);

/* Listato degli indirizzi IP del server */
int i;
for(i = 0; i < addr_count; i++) {
        printf("Address number %d: %s:%d\n",
                i +1, inet_ntoa((*paddrs)[i].sin_addr),
                (*paddrs)[i].sin_port
              );
}
```

N.B.: prima di procedere con queste istruzioni, il client deve avere creato

una socket, impostato l'heartbeat e l'RTO esattamente come lato server. Solo in questo modo, infatti, il client puó rilevare il cambiamento di interfaccia a lato server ed inviargli la notifica sotto forma di *event* e cominciare la trasmissione via cavo.

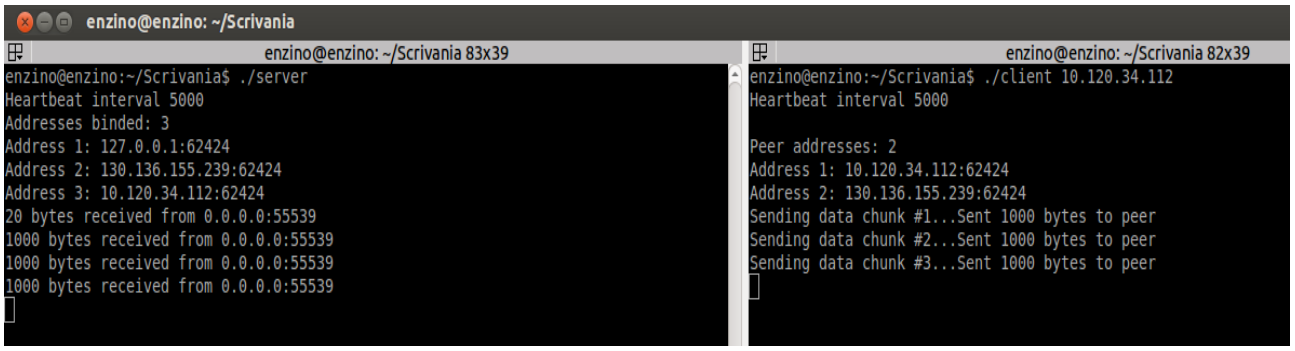


Figura 2.6, avvio Client SCTP ed inizio trasmissione

Dopo aver lasciato transitare alcuni pacchetti, disattiviamo la connessione Wireless (Figura 2.7): vediamo che il client notifica subito al server che cambierá interfaccia di comunicazione (avviso contenuto nel pacchetto da 148 Bytes, event di tipo **SCTP_PEER_ADDR_CHANGE**). Anche se dalle istantanee non é possibile vederlo, la ricezione a lato server, al momento del cambio da Wireless a Ethernet, si interrompe momentaneamente per poi recuperare tutti insieme all'improvviso i pacchetti rimasti in sospeso. La trasmissione poi continua poi come se niente fosse successo (Figura 2.7).

Figura 2.6, disattivazione scheda wi-fi

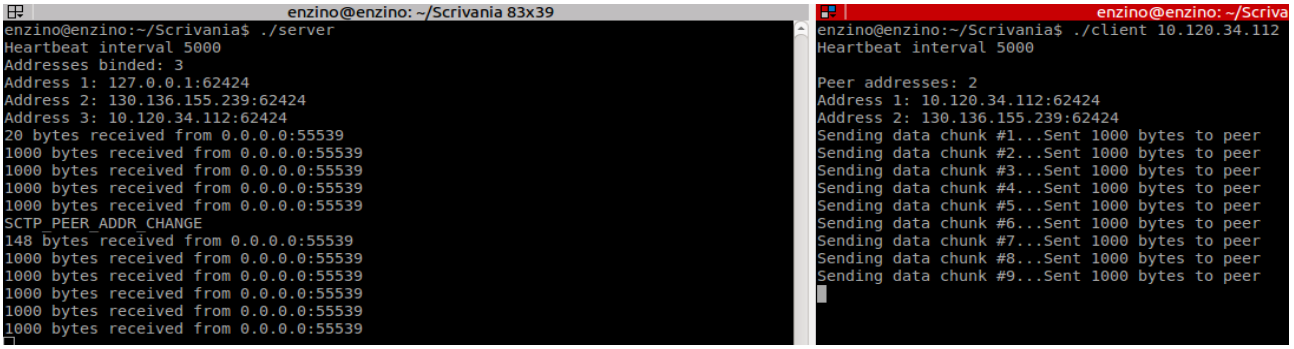


Figura 2.7, cambio interfaccia e proseguimento della comunicazione

Infine, quest'ultimo grafico sperimentale ci mostra come varia il tempo a seconda che la trasmissione prosegua normalmente indisturbata (notiamo in questo caso che il tempo é cira costante), oppure che si verifichi un cambiamento di interfaccia di rete del server dopo lo *shut down* di quella predefinita. Vediamo che il tempo impiegato da quest'ultima trasmissione cresce in proporzione non lineare all'RTO settato (il campo *srto_max* della struttura dati *sctp_rtoinfo*) ossia il timeout entro il quale il client cerca di connettersi ad un'altra interfaccia di rete del server, qualora presente. Dunque, maggiore é RTO settato, maggiore é il tempo che il client aspetta prima di cambiare interfaccia sebbene maggiorare l'RTO di n millisecondi non significhi un'attesa piú lunga di n millisecondi esatti.

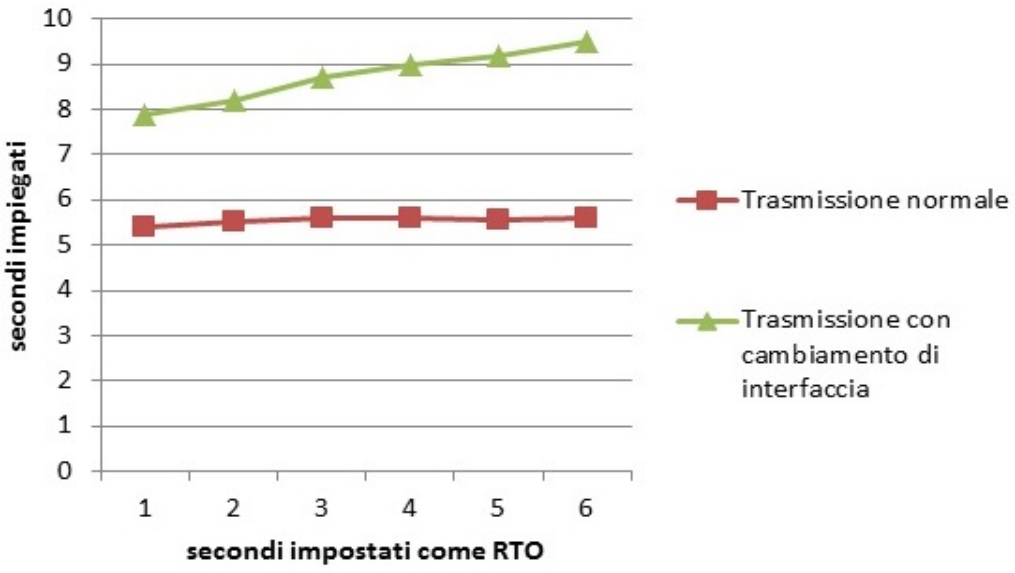


Figura 2.8, trasmissione normale vs. cambio d'interfaccia di rete

Capitolo 3

Il Multistreaming

3.1 Introduzione al Multistreaming

Il termine Multistreaming si riferisce alla capacitá di SCTP di inviare multipli *streams* indipendenti di chunks in parallelo. Dunque, in un'associazione SCTP uno *stream* é un canale logico UNIDIREZIONALE stabilito tra i due endpoints che fanno parte dell'associazione, all'interno del quale tutti i messaggi utente sono consegnati in sequenza (in ordine). Per esempio, nella normale procedura di caricamento di una pagina web, ne viene prima trasmesso l'HTML seguito poi da eventuali immagini o altri contenuti, il tutto quindi sequenzialmente. Invece, SCTP ci offre la possibilitá che tutti i contenuti vengano inviati assieme in parallelo in modo indipendente in un'unica associazione e poi re-assemblati in un unico documento. Pertanto un'applicazione non deve piú aprire multiple connessioni *end-tp-end* con lo stesso host semplicemente per indicare ed usare differenti flussi logici. Il numero di *streams* richiesti tra due SCTP endpoints é dichiarato durante la fase di instaurazione dell'associazione ed essi sono validi per tutta la durata della stessa.

Questo sicuramente é un grande vantaggio nel trasferimento di files. Infatti, usando TCP e dovendo trasmettere un certo numero x di files occorrerá inviarli o in sequenza uno dopo l'altro, oppure avviare in parallelo x connessioni diverse, una per ogni file da trasmettere. Usando SCTP, invece, tutti questi per ognuno di questi files viene avviato uno *stream* e quindi i dati da inviare sono *multiplexati* in un unico canale (associazione) come mostrato in Figura 3.2. Ogni *stream* é indipendente dall'altro ed ognuno si occupa della consegna in ordine ed affidabile dei propri DATA chunks.

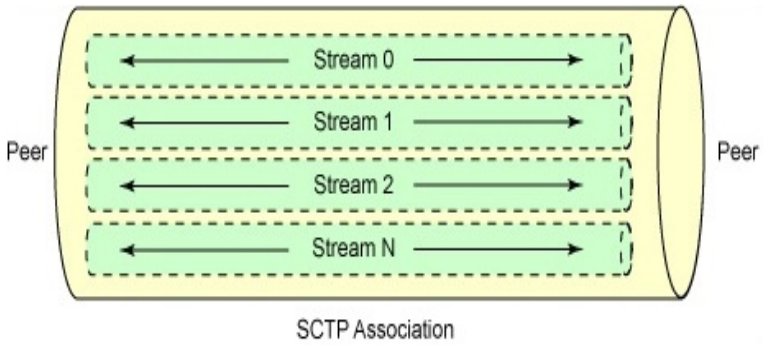


Figura 3.1, il concetto di Multistreaming

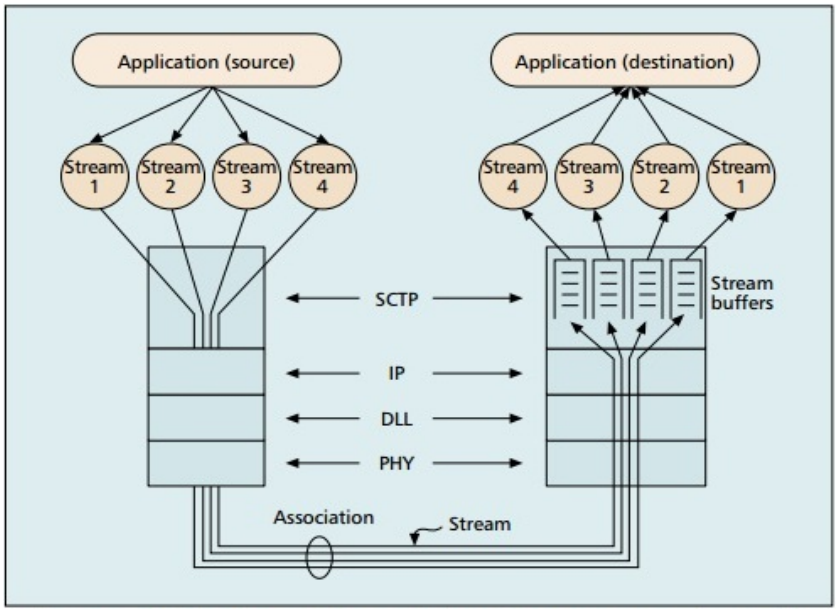


Figura 3.2, trasmissione in Multistreaming

3.2 Gestione e vantaggi del Multistreaming

Poiché i vari *streams* sono indipendenti da loro, la perdita di un DATA chunk appartenente ad uno *stream* non influenza la trasmissione degli altri i quali vengono normalmente consegnati a livello applicativo. Invece, i DATA chunks di uno *stream* seguenti a quello perduto vengono posti in attesa nell'appropriato *stream buffer* del ricevente fino a quando il DATA chunk mancante non é ritrasmesso e ricevuto.

Tale caratteristica fa in modo che SCTP sia esente dal fenomeno detto ***Head-of-line blocking*** **(HOL blocking)** il quale invece affligge TCP. Infatti, poiché in TCP vi é un singolo *stream*, l'invio di messaggi indipendenti

attraverso una connessione TCP fa in modo che la consegna di alcuni messaggi inviati dopo sia ritardata (sebbene corretta e completa) fino a quando un messaggio mandato prima di loro ma perduto (o a causa di ricezione di pacchetti fuori sequenza), non é ritrasmesso e consegnato (o la sequenza sia ripristinata correttamente). In altre parole se vengono inviati piú segmenti TCP ed il primo di questi é perso, gli altri devono attendere nella coda del ricevente finché il primo segmento non sia ritrasmesso e ricevuto correttamente. Tale effetto bloccante dell'HOL ritarda la consegna dei dati all'applicazione il che nei segnali telefonici ed in alcune applicazioni multimediali é inaccettabile.

Questo invece non accade in SCTP. Per esempio, come vediamo in Figura 3.3, il pacchetto 12 del primo *stream* viene subito consegnato a livello applicativo mentre i pacchetti degli altri *streams* sono posti in attesa fino a quando non arriva il DATA chunk mancante (rispettivamente il 9, il 4 ed il 22) ed il fenomeno dell'*HOL blocking* affligge solo questi 3 *streams* e non l'intera associazione.

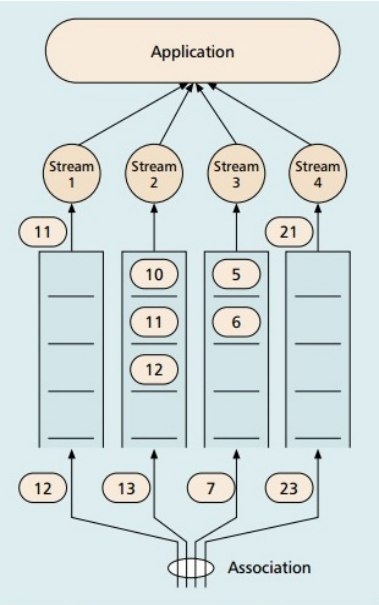


Figura 3.3, HOL blocking confinato solo ad alcuni streams

Infatti, per un grosso numero di applicazioni, la caratteristica di una rigida conservazione della sequenza non é veramente necessaria. Per esempio, poiché sappiamo che SCTP é nato per la trasmissione dei segnali telefonici (*SS7 Signaling*), in una telefonata é ragionevolmente giusto mantenere la sequenza dei messaggi che fanno parte della stessa risorsa (la stessa telefonata quindi) e non di tutte le telefonate assieme.

Poiché il trasporto dei vari *streams* é effettuato tramite un'unica associazione, essi sono soggetti ad un controllo del flusso e della congestione in comune per quell'associazione riducendo cosí anche gli *overhead* richiesti a livello di trasporto.

Inoltre, SCTP compie il Multistreaming creando un'indipendenza tra la trasmissione dei dati (*data transmission*) e la loro consegna (*data delivery*). In particolare, ogni *payload* dei DATA chunks utilizza due *sequence numbers*:

- **Transmission Sequence Number (SSN)**: il quale controlla la trasmissione dei messaggi occupandosi anche del rilevamento della loro eventuale perdita. Ogni segmento appartenente ad uno *stream* ha un SSN il quale é unico all'interno dello *stream* stesso;

- una coppia detta **Stream ID/Stream Sequence Number**: che viene usata per determinare la sequenza di consegna dei dati ricevuti per quello Stream ID.

L'indipendenza tra questi meccanismi permette al ricevente di determinare immediatamente sia quando si verifica un *gap* nella sequenza di trasmissione (dovuto, per esempio, alla perdita di un messaggio), sia se i messaggi che seguono il *gap* fanno parte dello *stream* colpito dalla perdita. Se un messaggio ricevuto fa parte dello *stream* in cui si é verificata la perdita, allora ci sará un *gap* corrispondente nello *Stream Sequence Number* mentre i messaggi di altri *streams* non mostreranno alcun *gap*. Perció, il ricevente puó continuare a ritirare messaggi dagli *streams* non affetti da perdita mentre continuerá la memorizzazione temporanea dello *stream* affetto da perdita fino a quando non si verifica la ritrasmissione di quanto manca.

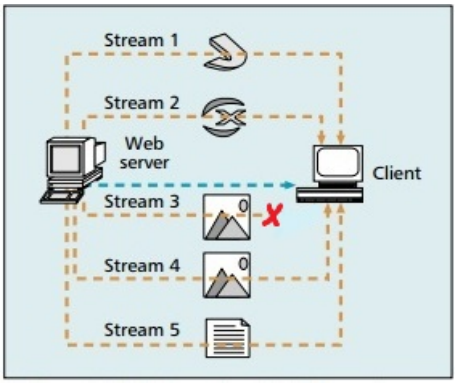


Figura 3.4, applicazione web multistreamed che sfrutta SCTP

La Figura 3.4 ci mostra un'esempio di *Web Browsing Application* che sfrutta il Multistreaming di SCTP. Vediamo che il client richiede una pa-

gina HTML che contiene diversi elementi multimediali oltre che al codice. Pertanto, tale pagina viene scissa in 5 oggetti:

1. Java Applet;
2. un controllo ActiveX;
3. un'immagine;
4. una seconda immagine;
5. semplice textitplain text.

Dunque, invece di creare una connessione separata per ogni oggetto (come avverrebbe in TCP), SCTP fa uso del Multistreaming per velocizzare la trasmissione della pagina HTML, avviando in parallelo uno *stream* per ogni singolo oggetto eliminando in questo modo l'effetto *HOL* tra i vari oggetti: se un oggetto, o parte di esso, (ad esempio una delle due immagini) viene perso durante la trasmissione gli altri vengono consegnati comunque subito al *web browser client* mentre si attende la ritrasmissione di quello mancante dal *web server*. Questo porta anche ad un incremento maggiore della *Congestion Window Size (cwnd)* basato sul numero di acks ricevuti, in questo caso 4 su 5 (a differenza invece di TCP in cui un solo mancato ack avrebbe ridotto la cwnd molto di piú).

3.3 Uno sguardo alle primitive per il Multistreaming

Anche per il Multistreaming, vogliamo dare uno sguardo generale al modo in cui si programmano due semplici applicazioni client-server affinché possano comunicare usando diversi canali nella loro associazione.

3.3.1 Multistreaming lato server

Lavorando col multistreaming, questa volta il client verrá impostato come modello **one-to-one** in quanto, in questo esempio riguardo al solo Multistreaming), non ci interessa che sia associato a multipli indirizzi IP. Dunque, creiamo una semplice socket listening per il server, la cui unica differenza é che va specificato esplicitamente che deve essere di protocollo SCTP.

```
int sockServ = socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP);
```

In seguito poi il server deve dichiarare quanti *streams* in uscita e quanti in entrata puó concedere ad un client quando vi si associa. Questo viene dichiarato mediante un'apposita struttura dati

il cui nome ci indica che tale struttura verrá utilizzata durante la fase di *INIT* dell'associazione, per prendere accordi di quanti *streams* aprire. L'esempio che mi propongo di presentare, mostra come si comporta un'associazione *multistreamed* per lo scambio di 3 files, ognuno su stream diverso, e al crescere della loro dimensione. Quindi, il server dichiara che puó concedere tale massimo numero di *streams* e tale opzione che va settata mediante la solita chiamata alla *setsockopt()*.

```
memset( &initmsg, 0, sizeof(initmsg) );
initmsg.sinit_num_ostreams = 3; /* uscenti */
initmsg.sinit_max_instreams = 3; /* entranti */
initmsg.sinit_max_attempts = 2;
setsockopt(sockServ, IPPROTO_SCTP,
           SCTP_INITMSG, &initmsg, sizeof(initmsg) );
```

Quando viene stabilita l'associazione, il numero di *streams* concessi é il minimo tra quelli voluti dal client e quelli che il server puó stabilire. Pertanto, in questo esempio, se il nostro client richiedesse 6 *streams* gliele verrebbero concessi solo la met'Ṡeguono poi le solite chiamate di sistema a *bind(), listen()* e *accept()*. Appena il client si connette al nostro server, quest'ultimo entra in un *ciclo for* dove ad ogni iterazione spedisce un file sullo *stream* i-esimo, il tutto sempre all'interno della stessa associazione, rimanendo quindi sempre connesso/associato al client. Questa, é una notevole differenza rispetto a TCP che richiederebbe l'instaurazione di 3 connessioni sequenziali introducendo un non trascurabile *overhead*. Vedremo nella prossima sezione, un piccolo confronti di efficienza tra SCTP e TCP a seconda di come vengono utilizzati, mostrando il loro grafico. La primitiva per inviare su uno *stream* é la *sctp_sendmsg()* il cui ottavo parametro é il numero dello *stream* su cui inviare. Nel nostro caso quindi abbiamo:

```
sctp_sendmsg(sockCli, (void *)buffer, (size_t)strlen(buffer),
        (struct sockaddr*)&client, (socklen_t)sizeof(client),
         0, 0, i /* stream number */, 0, 0 );
```

Come ultimo parametro di tale primitiva, é anche possibile passare un *time-to-live* scaduto il quale, se non é stato possibile effettuare l'invio, verrá restituito un messaggio di errore.

3.3.2 Multistreaming lato client

Il client per potere usufruire del *Multistreaming* ha bisogno di 4 strutture dati apposite, introdotte appositamente con l'implementazione del nuovo protocollo. Esse sono:

- **struct sctp_sndrcvinfo sndrcvinfo;**: per capire su quale stream é arrivato il messaggio;
- **struct sctp_initmsg initmsg;**: per richiedere l'apertura del numero di *streams* desiderato;
- **struct sctp_event_subscribe events;**: per abilitare la notifica di eventi;
- **struct sctp_status status;**: per sapere quanti *streams* sono stati concessi al client.

Dunque, il client a sua volta crea la propria socket ed imposta la variabile *initmsg* col numero di *streams* in entrata ed in uscita che vuole. In questo caso, trattandosi del trasferimento di 3 files, tale settaggio sará uguale a quello del server, seguito sempre dalla chiamata alla *setsockopt()* per abilitarlo. Affinché il client possa capire da quale *stream* é entrato il messaggio, occorre abilitare la notifica degli eventi per l'I/O di trasmissione, cosí che la variabile *sndrcvinfo* venga di volta in volta settata.

```
memset(&events, 0, sizeof(events));

/* Abilitiamo la notifica eventi per I/O dei dati */
events.sctp_data_io_event = 1;

res = setsockopt(sockCli, SOL_SCTP, SCTP_EVENTS,
                 (const void *)&events, sizeof(events) );
```

Dopo, il client esegue la solita chiamata *connect()* per associarsi al server e, prima di mettersi in ricezione, verifica quanti streams ha ottenuto, in entrata ed in uscita, accedendo rispettivamente ai campi *sstat_instrms* e *sstat_outstrms* della variabile *status*. Affinché tali informazioni siano scritte nella variabile *status* occorre, ad associazione effettuata, chiamare la *getsockopt()*. Dunque:

```
int lenStatus = sizeof(status);
getsockopt(sockCli, IPPROTO_SCTP, SCTP_STATUS, &status, &lenStatus);

printf("Client gained %d stream for input
```

```
        and %d streams for output\n",
        status.sstat_instrms, status.sstat_outstrms);
```

A questo punto il client entra in un "lungo" *ciclo while* chiamando di volta in volta la *sctp_recvmsg()* la quale riceve i dati in entrata da uno *streams* e scrive nella variabile sndrcvinfo da quale *stream* i dati sono arrivati. Pertanto, dopo la chiamata a tale funzione, occorre fare uno *switch* sul campo *sndrcvinfo.sinfo_stream* e a seconda dello stream su cui sono arrivati i dati, scrivo sul file appropriato.

```
while(1) {

 int received;
 int flags;

 received = sctp_recvmsg(sockCli, (void*)buffer, sizeof(buffer),
                         (struct sockaddr*)&server,
                         (socklen_t*)&servSize,
                        &sndrcvinfo, &flags);

 if (received == 0) {
        /* Trasmissione terminata per ogni stream,
           esco da ciclo while */
        break;
 }

 else {

        switch(sndrcvinfo.sinfo_stream) {
                case 0:
                   /* scrivi su file 0 */;
                case 1:
                   /* scrivi sul file 1 */;
                case 2:
                   /* scrivi sul file 2 */;
        }

 }

}
```

3.4 Studio grafico-sperimentale delle performance di TCP vs. SCTP

La Figura 3.5 soprastante ci mostra le differenti tempistiche di SCTP e TCP nell'ambito del trasferimento di tre files che di volta in volta crescono in dimensioni. Il grafico é stato realizzando ripetendo l'inivio dei 3 files per 10 volte per protocollo e per ogni nuova dimensione e poi calcolandone il tempo medio. Possiamo notare come SCTP, per dimensioni di files non troppo grosse, sia molto piú efficiente e veloce di TCP e questo sicuramente sarebbe un grosso vantaggio se applicato alle pagine web per la navigazione in Internet poiché pagine web molto complesse (con molto codice html e scripts, come quella di Facebook) sono dell'ordine di centinaia di KB e quindi SCTP potrebbe velocizzare il loro caricamento sul browser, a maggior ragione se queste contengono anche files multimediali. Man mano che i files crescono in dimensioni, il comportamento di SCTP si avvicina sempre di piú a quello di TCP per poi probabilmente coincidere o quasi.

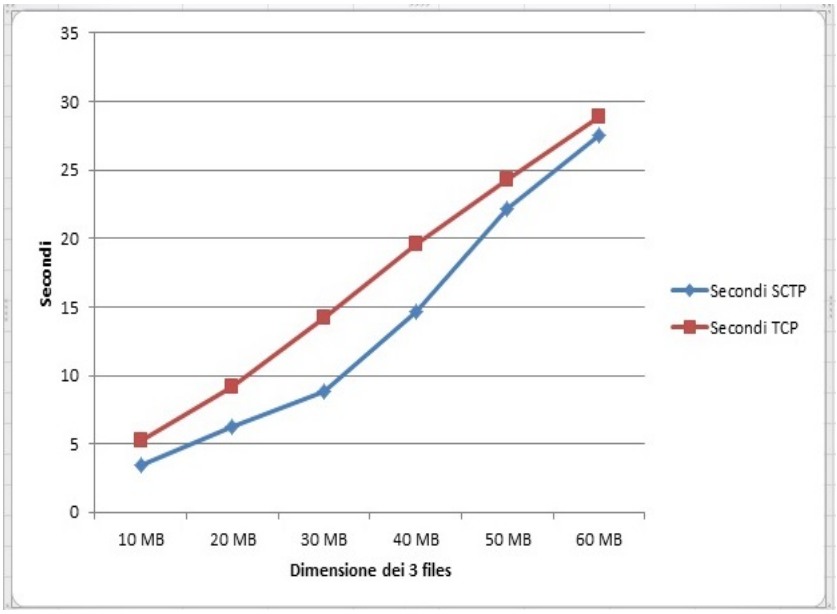


Figura 3.5, SCTP 3-streamed vs. TPC 3-connections

Un altro importante confronto da fare, é quello di comunicare col server TCP non con 3 connessioni sequenziali ma parallele. Quindi, per ogni file da inviare, il server crea un processo (*fork()*) che invia un file al corrispondente processo ricevente sul client. Sebbene in questo modo la trasmissione via TCP guadagna tempo, essa risulta tuttavia ancora piuttosto inferiore, come

prestazioni e velocitá, rispetto a SCTP. La Figura 3.6 mostra questo risultato, dove per ogni numero di files (sulle ascisse) viene creato uno stream (SCTP) ed un processo (TCP) per inviarli. Il test é stato effettuato inviando in parallelo un numero via via crescente di files di dimensione fissata (10 MB).

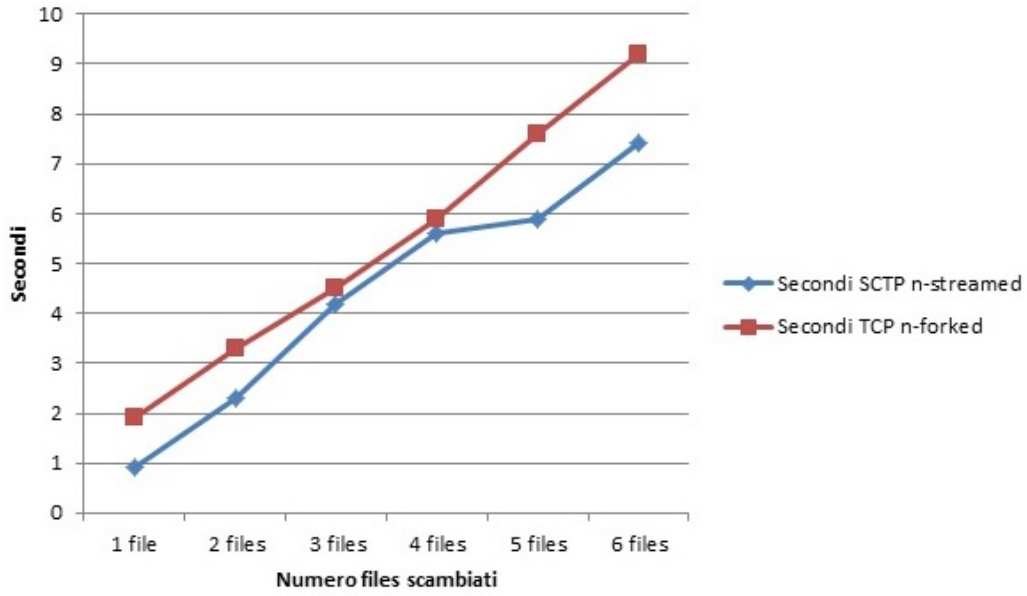


Figura 3.6, SCTP n-streamed vs. TPC n-forked

Infine, l'ultimo esperimento mette a diretto confronto SCTP e TCP a livello di singola associazione/connessione (Figura 3.7). Dunque vengono scambiati un numero crescente di files di dimensione fissata (sempre di 10 MB) creando n processi paralleli, un processo per ogni file da inviare in una connessione/associazione. Perció questa volta non si fa uso del Multistreaming di SCTP ma lo si usa come se fosse TCP, senza quindi le caratteristiche avanzate. Al contrario dei due esempi precedenti, qui abbiamo il risultato che la singola connessione TCP sembra poco piú efficiente della singola associazione SCTP. Questo probabilmente é dovuto al fatto che per instaurare un'associazione singola SCTP serve piú tempo, trattandosi di un *FOUR-way handshake* contro la *THREE-way handshake* di TCP. Le performance tuttavia diventano quasi identiche al crescere del numero di files da scambiare.

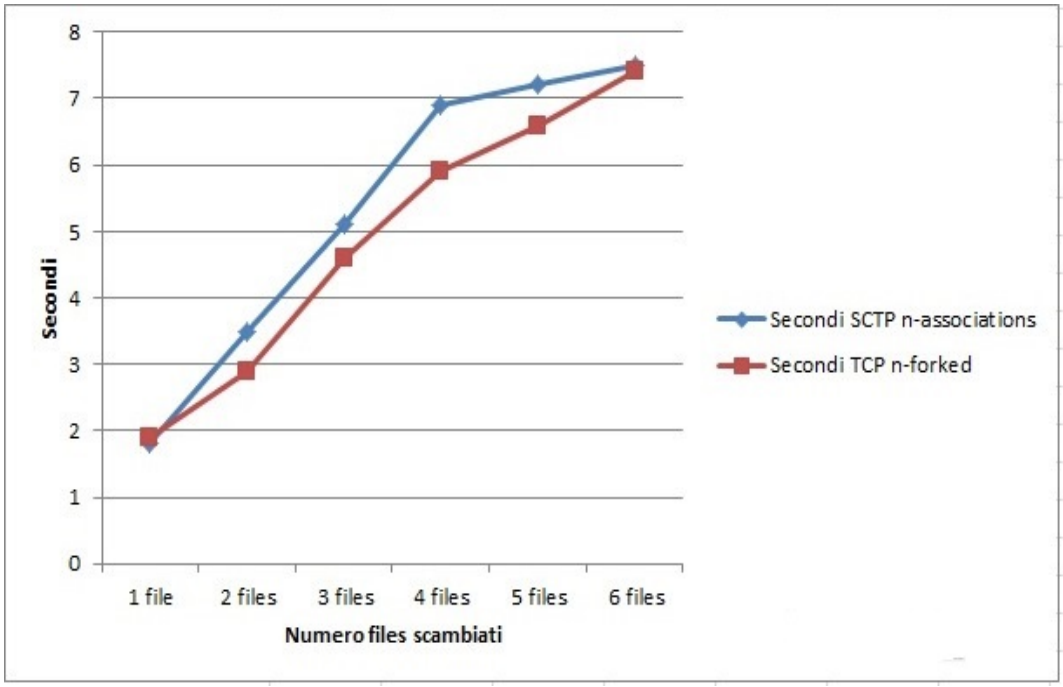


Figura 3.7, SCTP n-forked vs. TPC n-forked

3.5 Multiple FTP over SCTP

3.5.1 Perché passare a FTP over SCTP

Possiamo ragionevolmente stabilire che il Multistreaming SCTP si presta molto bene al trasferimento di diversi files in simultanea. Pertanto, si puó pensare di riadattare il protocollo FTP, *File Transfert*, in modo che possa sfruttare il Multistreaming. Fino ad oggi, FTP é sempre stato costruito utilizzando il protocollo di trasporto TCP, il quale peró, come dimostrano i risultati precedenti, introduce parecchi overheads, soprattutto per il fatto che in FTP su TCP occorrerebbe fare connessioni per il controllo ed una per ogni file da trasferire. Pertanto, si potrebbe fornire un servizio di *file transfert* in 3 modi diversi, sfruttando SCTP:

- rimpiazzare semplicemente TCP con SCTP, mantenendo un'associazione per il controllo ed un'associazione per ogni file da trasferire;
- utilizzare un'unica associazione SCTP *multistreamed* assegnando uno degli *streams* per il controllo, e gli altri per i dati;
- unire all'implementazione precedente dei comandi di "pipelining".

Nell'ultimo decennio, con lo sviluppo sempre piú veloce di Internet, vi é stato un grosso aumento dell'uso di HTTP con conseguente caduta in disuso di FTP, a causa della natura inflessibile della sua interfaccia nonché della sua inefficienza riguardo alla latenza e gestione dei ritardi textitend-to-end. Tuttavia é chiaro che un protocollo per trasferimento di files debba essere affidabile e sicuro, pertanto la scelta di appoggiarsi a TCP era sottintesa. Per aumentare il *throughput*, FTP ha poi iniziato a fare uso di connessioni TCP parallele. Quest'ultima scelta, tuttavia, si é rivelata *TCP-unfriendly* poiché non permette all'applicazione di avere una condivisione equa di banda causando anche un flusso maggiore di rete ed una maggiore instabilitá della stessa. Dobbiamo anche ricordare che connessioni parallele TCP consumano molte piú risorse di sistema di quanto sia necessario. Pertanto, sicuramente FTP puó beneficiare del Multistreaming SCTP, riducendo cosí gli *overhead*, soprattutto in trasferimento multiplo di files.

3.5.2 Stato attuale di Multiple FTP over TCP

Una sessione FTP attuale consiste in varie connessioni TCP: una, la prima, per informazioni di controllo, seguita poi dalle altre per il trasferimento dei dati. La connessione di controllo é utilizzata per lo scambio di comandi e relative risposte in semplici caratteri ASCII che vengono innescati dalla richieste utente. In seguito, viene stabilita un un'unica connessione per ogni trasferimento di dati oppure di *directory listing*, che viene poi chiusa a trasferimento completato. La chiusura di una connessione avviene quando il mittente legge il carattere di fine file EOF *End Of File* dal file che stava inviando. Pertanto, il numero di connessioni in una sessione di FTP é equivalente al numero di trasferimenti richiesto. Ogni connessione TCP facente parte di una sessione FTP puó essere di due tipi:

1. attiva: il client manda al server un comando PORT indicandogli l'indirizzo IP e la porta al quale il server dovrá stabilire le connessioni per il trasferimento dati.

2. passiva: il client apre direttamente la connessione col server. Tale modalitá permette di risolvere particolari problemi di interazione di FTP con NATs e FIREWALLs.

Comandi di uso comune in RETR (per ricevere un file), LIST (per *directory listing*), SIZE (per ottenere la dimensione in byte di un file) e STOR (per inviare un file). In modo particolare FTP supporta anche il recupero di di multipli files, basato su espressioni date a livello di comando dall'utente (per esempio digitando da terminale qualcosa di simile a *mget* * che fa trasferire

tutti i files di quella cartella). In questo caso i files vengono trasferiti indipendentemente e non vi é alcuna forma di informazione di connessione tra i trasferimenti dei vari files. Ognuno di questi trasferimenti é quindi composto dalla sequenza di comandi: PORT, SIZE e RETR. Dunque, per il trasferimento di n files otteniamo (n+1) connessioni: la prima di queste serve per il *directory listing*. La fine del trasferimento del file (EOF, *End Of File*) é segnalata dalla chiusura di una connessione per i dati di quel file.

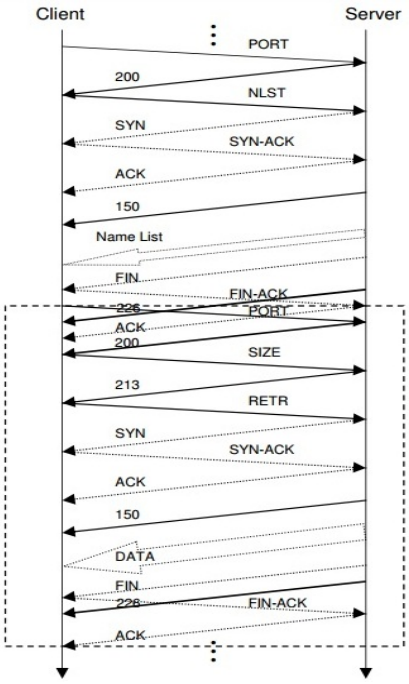


Figura 3.8, meccanismo di FTP over TCP

La Figura 3.8 mostra una *timeline* per la consegna di multipli files, dal server al client. Vengono mostrati i comandi e le risposte scambiate in fase iniziale di instauramento connessione e richiesta di *directory listing* seguita poi da una zona contornata dalla linea tratteggiata la quale viene ripetuta per ogni file da inviare.

Inefficienze di FTP over TCP

Il disegn corrente di FTP su TCP presenta alcune problematiche legate all'efficienza causate da:

- connessione di controllo separata da connessione per trasferimento dati;
- connessione dati non continuativa.

Per sua natura FTP, usando un approccio di segnali di controllo *out-of-band*, ha conseguenze sulla latenza end-to-end. Il traffico sulla connessione di controllo é per sua natura periodico perció tale connessione rimane tipicamente nella fase di inizio lento del controllo congestione TCP. Ma la connessione di controllo é vulnerabile a timeout a causa della natura *send-and-wait* dei comandi di controllo. Dunque un'operazione sará soggetta a timeout nel caso si verifichi una perdita o del comando o della sua risposta. Inoltre, siccome i flussi di conrollo e dati avvengono su connessioni distinte, questo provoca un exra *overhead* di 1,5 RTT (*Round Trip Time*) di cui 1 per stabilire la connessione e 0,5 per la chiusura della stessa. Inoltre l'host server deve mantenere ben 2 TCB (*Transmissione Control Block*) per sessione FTP la quale cose puó essere molto negativa su server molto usati e che ricevono molte connessioni, provocando una diminuzione del *throughput* dovuta all'attesa del liberarsi di blocchi di memoria per allocare le nuove risorse. Inoltre, poiché indirizzo IP e porta sono inviati in *plain text* durante il comando PORT in modo da facilitare il server a creare le connessioni per i dati, questo causa una mancanza in ambito di sicurezza ed inoltre la trasmissione di questi dati nel *payload* del protocollo causa problemi anche a livello di NAT (Network Addresses Translators).

Riguardo al secondo punto, la non permanenza della connessione dati causa un *overhead* di 1 RTT ogni volta che un file deve essere trasferito o per ogni operazione di *directory listing* il che potrebbe essere aumentato molto a causa di ritardi accodati. Infine, ogni connessione per i dati provoca un nuovo sondaggio sulla finestra di congestione durante la fase iniziale di avvio lento. Ogni connessione inizia sondando l'ampiezza di banda disponibile prima di raggiungere lo stato continuo e regolare della cwnd. La perdita di un pacchetto nella fase di avvio lento, prima che la cwnd sia larga abbastanza per la ritrasmissione veloce, risulterá in un timeout al server. Possiamo quindi concludere che tra il trasferimento di un file e l'altro, passa un ragionevole intervallo di tempo, impiegato per chiudere la connessione precedente, settare quella nuova per il prossimo file ed inviare il comando, prima che la trasmissione del file seguente abbia inizio.

3.5.3 Simple Multiple FTP over SCTP

Abbiamo detto che il Multistreaming, all'interno di un'unica associazione SCTP, separa i flussi di dati logicamente differenti in *streams* indipendenti. Dunque, non é piú necessario che un'applicazione FTP apra n connessioni end-to-end allo stesso host solo per indicare flussi semanticamente differenti. Pertanto, sfruttando questa caratteristica, FTP, per inviare n files, userebbe una sola associazione SCTP, per tutta la durata della sessione FTP, con n+1

streams di cui il primo sarebbe usato per il controllo e gli altri per i dati. In generale, un primo funzionamento sarebbe circa il seguente:

1. durante l'inizializzazione sono aperti due *streams* per ogni direzione;

2. client e server si scambiano comandi e relative risposte sullo *stream 0*;

3. lo *stream 1* viene utilizzato per i dati ed il *directory listing*.

Questo primo approccio mantiene quindi una semantica analoga a quella di TCP, usando due streams (su un'associazione persistente) invece che due connessioni. Nel caso venga emanata una richiesta di files multipli, il client manda via via le richieste sullo *stream 0* mettendosi poi in ricezione dei file sullo *stream 1* in modo iterativo/sequenziale.

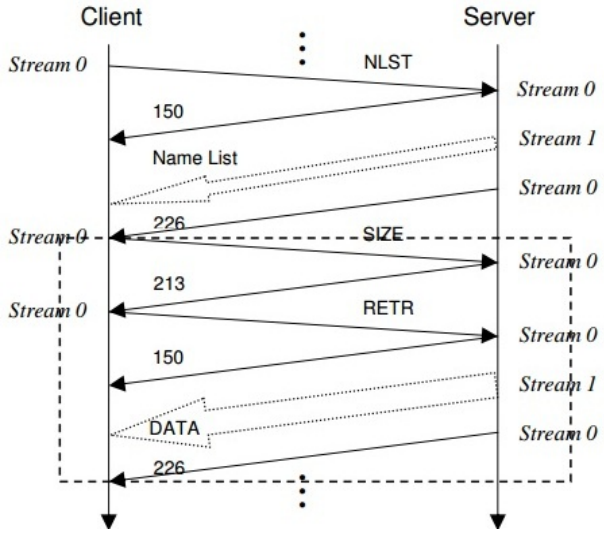


Figura 3.9, meccanismo di FTP over SCTP 2-streamed

La Figura 3.9 mostra quanto appena discusso sopra dove la zona tratteggiata indica ció che va ripetuto sequenzialmente per ogni file da inviare. Tale approccio riduce molto gli overhead discussi prima nel caso di TCP. Ed infatti il numero di RTT é ridotto poiché abbiamo un'unica associazione SCTP per tutta la durata della sessione FTP e dunque non occorrono multipli avvi e chiusure di connessione nonché di comandi PORT. Anche la memoria occupata lato server é minore poiché alloca al piú la metá di TCBs rispetto a quanto richiesto con TCP.

FTP over SCTP Multistreaming e comandi di *pipelining*

Quest'ultima considerazione evita una riduzione non necessaria della cwnd per l'inivio multiplo di files. Nella sezione precedente poiché i comandi SIZE e RETR, per il prossimo file da trasferire, sono inviati solo dopo che il file precedente ha completato il trasferimento, questo causa una riduzione della cwnd. Infatti, ogni nuovo invio di file non puó la banda disponibile sondata precedentemente e si riparte sempre da capo. Vorremmo, invece, che se l'inivio del file *i-esimo* ha sondato una buona ampiezza di banda, allora l'iivio del file (i+1)-esimo iniziasse subito con tale ampiezza sondata. I comandi di pipelining assicurano un flusso continuo di dati dal server al client durante tutta l'esecuzione di un trasferimento multiplo di files.

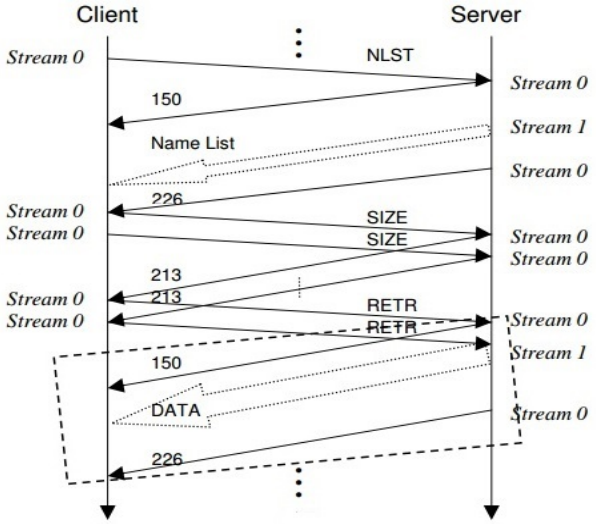


Figura 3.10, meccanismo di FTP over SCTP 2-streamed con comandi di pipelining

Come ci mostra la Figura 3.10, dove aver analizzato la lista di tutti i files richiesti, il client manda subito il comando SIZE per tutti i files. Non appena viene ricevuta una risposta per ognuno di tali comandi SIZE, il client invia subito il comando RETR per esso. Poiché il controllo del flusso é ordinato, avremo che le risposte per il comando SIZE e l'invio del relativo RETR avverranno nella stessa sequenza dei comandi SIZE stessi. In questo modo FTP vede il trasferimento multiplo dei files come un unico ciclo di dati il che risulta in un miglior uso della cwnd e quindi in uso piú efficiente dell'ampiezza di banda.

Capitolo 4

SCTP e sicurezza

In merito alla sicurezza, abbiamo giá discusso come SCTP fornisca, grazie al suo *four-way handshake* per stabilire le associazioni, un'importante metodo di protezione verso il *SYN flooding*, un tipo di attacco della famiglia *Denial Of Service* (DoS). In questo ambito, infatti, SCTP é stato implementato tenendo ben in mente le varie problematiche di TCP. Per rendere il piú difficoltoso possibile l'iniezione di pacchetti SCTP "contraffatti" nel corso di un'associazione tra due end-point (un tipo di attacco conosciuto come *man-in-the-middle attack*), entrambi gli end-point utilizzano un valore a 32 bit chiamato *Verification Tag* per assicurarsi che i datagrammi SCTP, contenenti dati, che si stanno scambiando, appartengano davvero ad un'associazione esistente (si veda Figura 1.2). Perció, affinché un pacchetto SCTP dati sia accettato dal ricevente, deve avere un *Verification Tag* valido. Tale tag viene concordato durante la fase di inizializzazione dell'associazione ed é compito del mittente inserirlo ogni volta in un pacchetto che sta per inviare. Infatti, quando un client vuole connettersi ad un server via SCTP, esso setta il *Verification Tag* dell'*INIT Chunk* a 0 e lo inivia al server. Se il server riceve un datagramma SCTP il cui *Verification Tag* é 0, allora deve verificare che tale pacchetto contenga un solo chunk del tipo INIT. Se cosí non é, il server scarta il pacchetto. Altrimenti il tag viene concordato durante le fasi successive della costruzione dell'associazione. Durante l'inivio di dati, se la destinazione riceve un datagramma SCTP il *Verification Tag* non é quello concordato, allora essa lo scarta silenziosamente senza comunicare alcunché a nessuno e continua. Dunque, l'unica possibilitá di portare a buon fine un attacco sta nella probabilitá di indovinare o il valore contenuto nel COOKIE durante la fase di avvio dell'associazione, o quella di indovinare tale *Verifcation Tag*. Perció, sebbene lo scopo principale di SCTP sia quello di fornire maggiore disponibilitá, robustezza e velocitá alla rete (grazie al *Multihoming* ed al *Multistreaming*, esso ci fornisce anche qualche meccanismo base di pre-

venzione contro certi attacchi. Tuttavia, qualora sia richiesto autenticazione, integritá e confidenza, esso deve appoggiarsi a ***IPSec***. Infatti, poiché SCTP é stato progettato anche per la mobilitá, questo apre la possibilitá al verificarsi di attacchi di tipo *ridirezione del traffico* dove colui che svolge l'attacco dichiara che il suo indirizzo IP dovrebbe essere aggiunto ad una sessione SCTP tra due end-points ed essere usata per le future comunicazioni. In questo modo il traffico tra sorgente e destinazione puó essere visto dall'attaccante pertanto l'aggiunta di un indirizzo IP ad un'associazione dovrebbe essere autenticata.

4.1 SCTP e IPSec

IPSec é stato progettato per fornire uno strato di sicurezza a livello di rete, sia per IPv4 che per IPv6, basato sulla crittografia e sull'autentiazione dei pacchetti IP. Dunque IPSec fornisce servizi di sicurezza a livello IP permettendo ad un end-point di selezionare i protocolli di sicurezza desiderati, di determinare gli algoritmi da usare e di scambiarsi le chiavi crittografiche richieste per fornire i servizi sopra citati.In alternativa, per la sicurezza, sono utilizzati TLS e SSL tuttavia il nostro interesse é per IPSec poiché é su questo che si sta iniziando ad integrare con SCTP. IPSec é composto in realtá da due sotto-protocolli:

- **Encapsulating Security Payload** (ESP): fornisce integritá dei dati, autenticazione e confidenzialitá ;
- **Authentication Header** (AH): meno complesso del precedente, ne fornisce solo i primi due servizi.

La Figura 4.1 sottostante mostra il posizionamento del protocollo IPSec (composto dai due suoi sotto-protocolli) nello stack ISO/OSI.

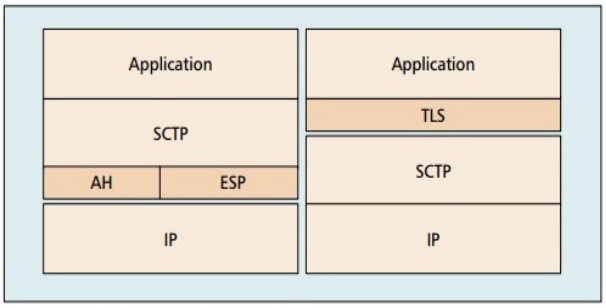


Figura 4.1, IPSec

Innanzitutto, utilizzando AH e ESP per fornire sicurezza ai pacchetti SCTP, essi sono trattati senza distinzione come pacchetti provenienti da un qualsiasi altro protocollo a livello di trasposto (come TCP o UDP). Per stabilire un'associazione IPSec sicura (SAs) é richiesto la negoziazione di una chiave tra client e server; in modo particolare, questo é ottenibile mediante l'uso di **IKE**, (Internet Key Exchange, RFC 2409 e RFC 6071). La gestione di una SAs é abbastanza complessa giá in TCP e lo risulta ancora di piú in SCTP a causa del *Multihoming* poiché di base essa richiederebbe multiple SAs, una per ogni indirizzo IP. Ricordiamo che quando indicato di seguito é ancora in fase sperimentale e di ricerca e quindi non ancora ben implementato nelle attuali versioni di SCTP. Tuttavia, per ogni aggiornamento che include l'aggiunta di un indirizzo ad un'associazion, IKE dovrebbe validare tutti gli indirizzi facenti parte di un end-point SCTP usando un qualche metodo *out-of-band*.

4.1.1 Breve panoramica su IKE

Una sessione IKE per lo scambio di chiavi crittografiche consiste in due fasi:

1. lo scopo della prima fase é quello di stabilire un canale di comunicazione sicuro ed autenticato generando una chiave segreta IKE condivisa, che verrá utilizzzata per le comunicazioni successive, oppure usando una chiave pre-condivisa giá nota ai due peers;

2. nella seconda fase, detta *Quick Mode*, i due peers usano il canale precedentemente creato per negoziare associazioni sicure sulle quali si basano altri servizi/protocolli di sicurezza, nel nostro caso di IPSec.

4.1.2 SCTP e IKE

Essenzialmente, vi sono due questioni che riguardano l'uso di IKE quando si negozia la protezione per il traffico SCTP:

1. poiché SCTP ammette molteplici indirizzi IP sia lato sorgente che lato destinazione, allora dovrebbe essere possibile per IKE negoziarli efficientemente durante la fase di *Quick Mode* nella quale vengono scambiate le informazioni per concordare le chiavi dopo avere creato un canale sicuro di comunicazione. L'approccio piú diretto é quello di negoziare una coppia di IPSec SAs per ogni combinazione di indirizzi sorgente e destinazione. Questo puó risultare in un numero grande e non necessario di SAs provocando quindi uno spreco di tempo e memoria. Attualmente, comunque, l'implemetazione di IKE supporta tale possibilitá.

2. per ogni SA creata, IKE utilizza dei selettori per riferirsi ad essa e nel caso di multipli indirizzi IP occorre ricevere abbastanza informazioni dalla fase uno per creare il relativo selettore.

Una SA individua una comunicazione unidirezionale. Quindi per avere un canale *full duplex* occorre crearne una per ogni verso di comunicazione. Al fine di semplificare la gestione delle SAs viene usato un database detto SAD (*Security Association Database*) che tiene traccia della SAs attive. Ogni selettore all'interno di tale database individua una SA secondo i parametri seguenti:

- indirizzi IP (sorgente e destinazione) dei peer coinvolti;
- protocollo utilizzato per il tunnel (AH o ESP);
- tecniche di cifratura usate e relative chiavi;
- Un intero a 32 bit detto SPI, (*Security Parameter Index* che serve nel caso esistano differenti sessioni IKE con uno stesso host.

Attualmente, IKE supporta solo il semplice scenario di due hosts che comunicano utilizzando solo un indirizzo IP sorgente ed uno destinazione; dunque occorrono piú selettori se per caso la comunicazione fosse *multihomed*. Non vi é un diretto supporto quindi ad SCTP. Ogni selettore creato deve essere validato e pertanto deve essere reso noto sia all'iniziatore della sessione che al ricevente che possono ricevere informazioni da tutti gli indirizzi che sono stati raccolti dil selettori in fase due. Attualmente questo é configurabile

solo mediante meccanismi *out-of-band*. Per esempio, puó essere impostato manualmente dall'amministratore di rete.

Inoltre, ogni end-point IP che implementa IPSec mantiene anche una seconda base di dati, la SPD (*Security Policy Database*) che indica, secondo delle regole SP (*Security Policy*), quale sia il traffico, ed il tuo tipo, che debba essere instradato in modo sicuro attraverso il tunnel IPSec. Per fornire supporto al *Multihoming* si é suggerito di cambiare leggermente i records dell'SPD, creandone uno per ogni indirizzo IP degli host coinvolti nella sessione IPSec e facendo in modo, con l'introduzione di nuove tabelle, che tali records facciano riferimento ad una sola specifica SA. Questo a condizione che anche i protocolli per la gestione delle chiavi permettano di associare le stesse a piú indirizzi di uno stesso host. Sebbene IKE possa stabilire delle *policy* che possono coprire un'intera sottorete (*subnet*), ció non é sufficiente per SCTP poiché gli indirizzi IP di un host possono anche appartenere a sottoreti diverse. Una soluzione possibile sarebbe quella di configurare le SA con liste di indirizzi IP invece che farle basare su singoli scambi in fase due di IKE, dove ogni scambio supporta solo un'indirizzo IP per host.

Dunque, per potere ottenere gli stessi vantaggi dello scenario standard di IPSec (un indirizzo IP per host) anche nel caso del *Multihoming*, occorrerá fare in modo che:

- si possa utilizzare la stessa chiave pre-condivisa per convalidare tutti gli indirizzi IP di un host che vanno notificati durante la fase uno;

- oppure utilizzare uno o piú certificati. Nel primo caso, il certificato contiene tutte le informazioni degli indirizzi IP oppure, nel secondo caso, occorre avere un certificato per ogni indirizzo dell'host.

Poiché SCTP é stato inizialmente progettato per fornire mobilitá (in particolare per il traffico telefonico su Internet), questo apre le possibilitá per attacchi di redirezionamento del traffico, dove un intruso dichiara che il suo indirizzo IP deve essere aggiunto ad'associazione SCTP in corso tra due hosts, ed essere usabile per future comunicazioni. In questo modo il traffico tra i due hosts puó essere spiato dall'intruso o anche modificato, ed é per questo che ogni aggiunta di un indirizzo durante la comunicazione andrebbe autenticato. Pertanto, IKE deve validare tutti gli indirizzi facenti parte di un endpoint, sia attraverso certificati presentati durante la fase uno, o tramite i soliti metodi *out-of-band*. Il ricevitore, durante la fase due di IKE, deve dunque verificare l'autoritá, per l'iniziatore della sessione IPSec di ricevere ed inviare traffico per tutti gli indirizzi presenti nei selettori di fase due. Se questo non fosse fatto, permetterebbe a qualsiasi altro peer di vedere il traffi-

co tra iniziatore e ricevitore, a condizione che il peer intruso riesca a portare a compimento la fase 1 di IKE per una SA.

4.2 Secure SCTP

Come abbiamo appena mostrato, ci sono limitazioni significative nell'uso di SCTP assieme ai protocolli standard di sicurezza. Perció é stata proposta un'estensione di SCTP che integra alcune funzionalitá di sicurezza all'interno del protocollo stesso. Tale estensione é chiamata ***Secure SCTP*** (S-SCTP), creata appositamente per evitare i vari svantaggi che si otterrebbero con una versione non integrata di SCTP e IPSec.

4.2.1 Gli scopi di S-SCTP

SCTP é stato progettato con i seguenti criteri:

- *Sicurezza*:
  fornisce integritá, confidenzailitá ed autenticazione del mittente non solo per i dati trasportati ma anche per le informazioni di controllo peer-to-peer. Perció le varie vulnerabilitá che si avrebbero nel contesto del DAR (*Dynamic Address Reconfiguration*) possono essere evitate. Tutti i *chunks* di un pacchetto SCTP soggetti a criptazione sono raggruppati e criptati insieme per limitare *overheads*.

- *Performance e scalabilitá*:
  integrando la sicurezza direttamente all'interno del protocollo, quando questo viene usato come metodo di trasporto sicuro, i vari *overheads* possono essere ottimizzati secondo i bisogni dell'applicazione. Sia nel caso di piú *streams* che in quello di multiple combinazioni di indirizzi possibili tra *endpoints*, S-SCTP prevede una soluzione scalabile garantita dallo stabilire una sola sessione sicura per associazione SCTP. Comparato a IPSec nel caso di multiple combinazioni di indirizzi tra endpoints, il costo in questo caso é minimizzato poiché non c'é bisogno di stabilire multiple associazioni di sicurezza.

- *Facilitá d'uso*:
  Affinché gli utenti traggano beneficio dalla piena funzionalitá di S-SCTP, senza costo di configurazione eccessivi, sono stati definiti diversi livelli:

1. Livello di Sicurezza 0: a questo livello, S-SCTP non usa nessuna delle nuove funzioni di sicurezza ed é pienamente compatibile con SCTP standard.
2. Livello di Sicurezza 1: a questo livello tutti i *chunks* e le intestazioni di tutti i pacchetti di un'associazione sono autenticati e ne viene controllata l'integrotá tramite l'algoritmo basato su funzione *hash* detto HMAC.
3. Livello di Sicurezza 2: solo alcuni *chunks* selezionati dall'applicazione (tramite un flag aggiuntivo) sono criptati.
4. Livello di Sicurezza 3: tutti i *chunks* sono criptati mentre il pacchetto completo é autenticato e ne viene controllata l'integritá.

All'interno di una sessione S-SCTP, i due endpoint possono usare diversi livelli di sicurezza e perfino durante la vita di un'associazione SCTP l'applicazione SCTP puó scegliere di cambiare il livello di sicurezza.

4.2.2 Concetti base di S-SCTP

Ogni associazione SCTP puó essere protetta da una sessione S-SCTP, indipendentemente dal numero di *streams* usati o dal numero di indirizzi IP della destinazione.

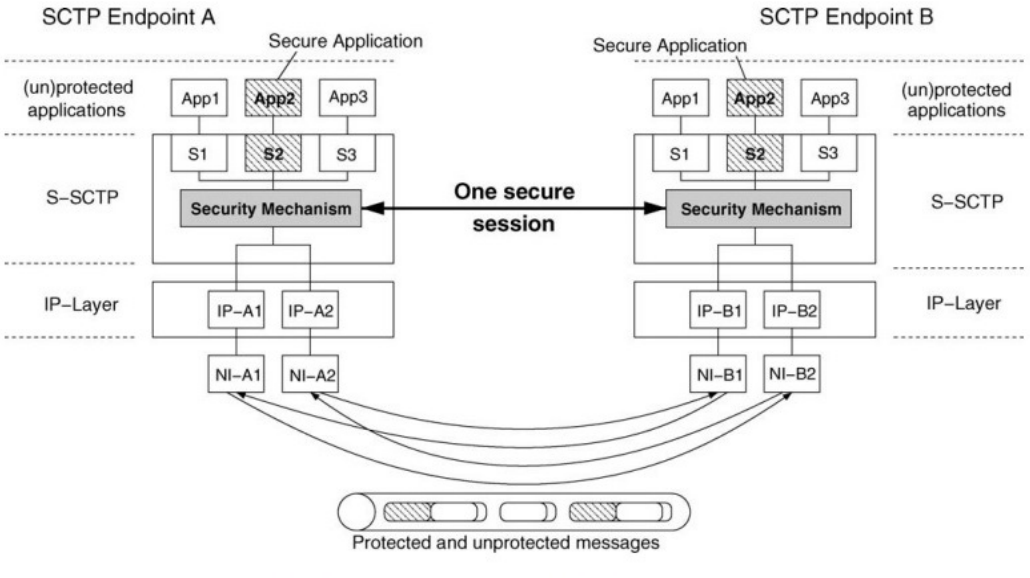


Figura 4.2, S-SCTP connessione tra 2 endpoints multihomed

La Figura 4.2 mostra che il nucleo centrale della sicurezza di S-SCTP é logicamente situato tra due blocchi funzionali: il blocco superiore si occupa di raggruppare i DATA *chunks* mentre il livello inferiore sceglie i percorsi di rete, scegliendo l'indirizzo destinazione a cui inviare i pacchetti. S-SCTP usa un meccanismo di cifratura ibrido: un algoritmo di cifratura asimmetrico crea una chiave segreta condivisa tra i due endpoints mentre un algoritmo a chiave simmetrica si occupa si criptare i dati utente. Un'associazione SCTP é divisa per fasi:

1. instaurazione di una normale associazione SCTP;

2. quando uno dei peer setta una livello di sicurezza maggiore di 0, allora si entra nella fase di *iniziazione della sessione sicura*. Durante tale fase i peers si scambiano messaggi per stabilire reciprocamente un *common master secret*. Finché tale fase non é terminata non é possibile usare alcuna *feature* di S-SCTP;

3. la terza fase é chiamata *fase di sessione sicura* e si possono usare tutte le *features* del livello di sicurezza scelto;

4. quando l'applicazione termina il proprio compito si entra nella fase di *shut down* della sessione sicura. Il peer che chiude la sessione non puó piú utilizzare alcuna *feature* di S-SCTP ma deve comunque aspettarsi altri dati criptati o autenticati, rimasti indietro, fino a quando non arriva l'*aknowledgement* anche dell'altro peer della chiusura sessione. Se l'associazione SCTP ancora é in corso, eventualmente si riparte dalla fase 1.

4.2.3 Nuovi tipi di DATA chunks per S-SCTP

Affinché S-SCTP possa lavorare compatibilmente con SCTP, sono stati definiti nuovi tipi di DATA *chunks* che sono usati per lo stabilimento iniziale e chiusura della sessione sicura, per lo scambio delle chiavi, per l'autetnticazione dei pacchetti e per il trasporto di dati criptati. Questi nuovi tipi sono indicati in Tabella 4.1.

Questi nuovi *chunks* sono stati introdotti affinché un'implementazione semplice di SCTP, che non supporta quindi l'estensione a S-SCTP, riporti un'eccezione a livello applicativo qualora dovesse rilevare un *chunk* per stabilire una sessione S-SCTP, facendo poi continuare la trasmissione come una normale associazione non sicura. Solo nel caso in cui venga "incontrato" uno degli ultimi tre tipi di pacchetto listati in tabella, allora in questo caso

Table 2
List of new chunk types for S-SCTP.

Chunk type	Chunk name
0xD0	Secure session open request
0xD1	Secure session certificate
0xD2	Secure session acknowledge
0xD3	Secure session server key
0xD4	Secure session client key
0xD5	Secure session open complete
0xD6	Secure session close
0xD7	Secure session close acknowledge
0x10	Encrypted data chunk
0x11	Authentication chunk
0x12	Padding chunk

Tabella 4.1, nuovi DATA chunks per S-SCTP

l'associazione viene chiusa. L'*Encrypted DATA chunk* serve per trasportare un *payload* composto da crittogrammi (Figura 4.3).

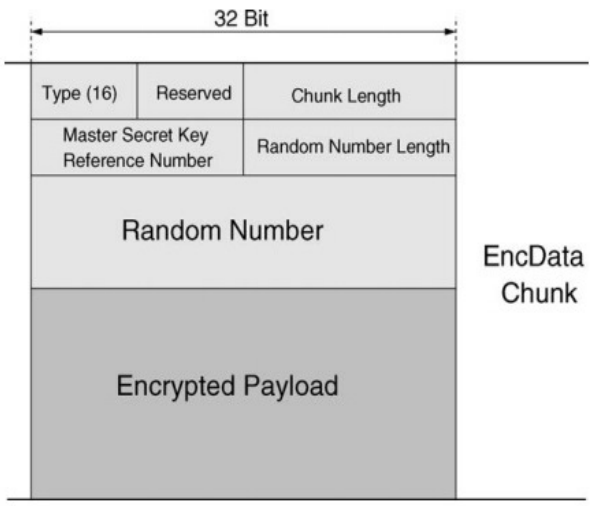


Figura 4.2, Encrypted DATA chunk

Tale pacchetto contiene un'intestazione in *clear text* composta da il tipo e la lunghezza del pacchetto stesso, un numero generato a random e la sua lunghezza, un riferimento ad una chiave segreta (*Master Secret Key*) seguito poi dai campo dati che contiene dei *chunks* criptati concatenati. Prima della criptazione tale campo deve essere riempito di una lunghezza mininima di 64 bits cosé da evitare attacchi che analizzano se vi sono dei *patterns* nel traffico oppure per raggiungere il numero di bit necessari affinché possa venire applicata la Cifratura a blocchi, un algoritmo a chiave simmetrica. Grazie al riferimento alla chiave segreta, chi riceve un *EncDATA chunk* é in grado di decidere quale *set* di chiavi é stato usato for la criptazione e l'autenticazione, in quanto durante la vita della sessione sicura le chiavi tra client e server possono essere aggiornate o cambiate varie volte.

4.2.4 Gestioni delle sessioni sicure S-SCTP

Abbiamo detto che una sessione sicura S-SCTP inizia con una normale associazione tra client e server SCTP, con il *4-way handshake*. Una sessione sicura S-SCTP puó essere innescata in qualsiasi momento e piú volte nel corso dell'associazione. Anche il "negoziamento" di tale sessione é composta da uno scambio di quattro messaggi composti dai nuoti tipi di *chunks* sopra indicati la cui ritrasmissione é garantita nel caso di perdita o di scadenza di un timeout. Dunque, vediamo le varie fasi:

1. un client manda un *Secure Session Open Request chunk* al server. Il tipo di questo *chunk* é scelto in modo tale che un endpoint che non

supporta S-SCTP, qualora lo dovesse ricevere, si limita a scartarlo. In tal caso verrá verrá riportato un errore a livello di trasporto a lato client conservando peró la normale associazione SCTP creata prima; Se quanto indicato sopra va a buon fine, il server riceve col pacchetto anche una lista di algoritmi di cifratura supportati dal client. Di solito tale lista comprende un algoritmo per scambio chiavi, uno a chiave simmetrica ed uno a chiave asimmetrica. Eventualmente il client puó anche presentare un certificato, qualora ne sia in possesso;

2. a questo punto il server risponde con un *Secure Session Open Acknowledgement* indicando al client quali algoritmi di cifratura ha scelto. Inoltre il server invia anche un *chunk* con il proprio certificato, in modo da inoltrare al client le proprie credenziali sicure, ed anche un *Secure Session Server Key chunk* che contiene materiale per la gestione e computazione delle chiavi degli algoritmi concordati sopra;

3. il client risponde a tutto questo con il solito ack, assieme ad altro materiale sempre usato per le chiavi;

4. infine vengono scambiati due *Secure Session Open Complete chunks* i quali contengono un HMAC (*keyed-hash message authentication code* di tutti i messaggi scambiati precedentemente cosí che sia client che server possano verificare che tutti i messaggi sono stati trasmessi correttamente e non manomessi.

Lo scenario di tale comunicazione é illustrato in Figura 4.3

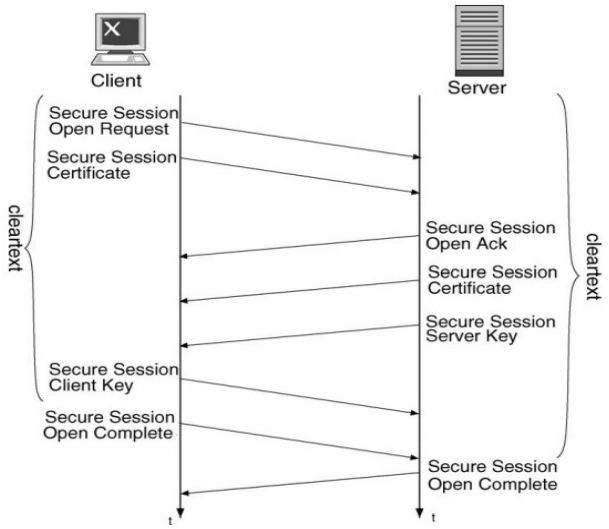


Figura 4.3, 4-way handshake SCTP Secure Session

Infine, quando si vuole chiudere la sessione sicura, client e server si scambiano *chunks* indicati in Figura 4.4. Il primo *chunk* inviato da client comprende un TSN cumulativo dei DATA *chunks* criptati ed inviati ed eventualmente anche il TSN piú grande di tutti, che deve ancora arrivare, nel caso vi siano ancora *chunks* in sospeso; in caso contrario é settato un opportuno. Il server risponderá con un ack ma solo quando tutti i pacchetti in sospeso sono stati ricevuti, nel caso ce ne fossero. Da questo punto in avanti quindi la trasmissione di *chunks* criptati non é piú permessa.

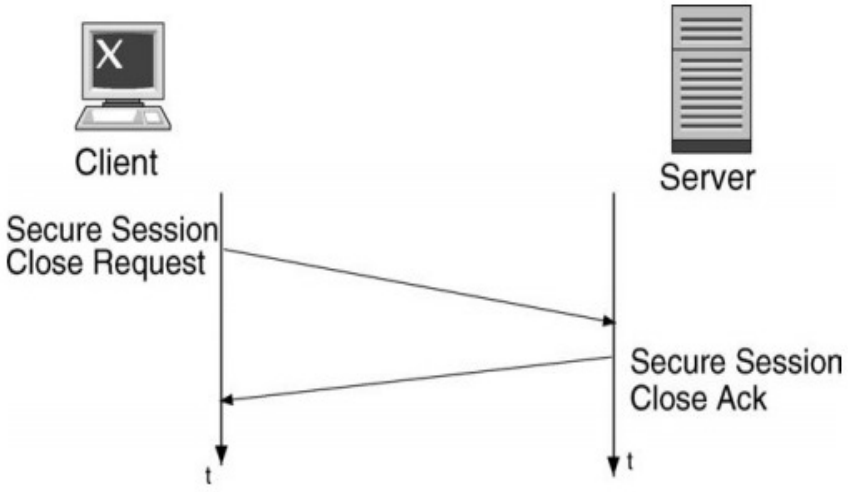


Figura 4.4, Shut down di una SCTP Secure Session

4.2.5 Trasporto sicuro dei dati

Infine, osserviamo la Figura 4.5 che mostra piú in dettaglio il processo di assemblamento dei *chunks* per il livello due, qualora essi siano criptati selettivamente. I *chunks* pronti da inviare e che devono essere criptati (in figura sono quelli piú scuri) sono raggruppati assieme per creare un *EncDATA chunk* a cui puó servire un *chunk* di riempimento per arrivare alla grandezza del blocco di cifratura. A questo punto i vari dati sono criptati al *EncDATA chunk* é aggiunto l'header dell' *EncDATA* in *clear text* . Se nel pacchetto cosí risultante é rimasto dello spazio libero, possono eventualmente essere inseriti anche altri DATA *chunks* che non devono essere criptati. A tutto questo viene aggiunto il *common header* di un normale pacchetto SCTP e sul pacchetto finale cosí ottenuto viene computato un HMAC che viene poi appeso in fondo ad esso in un *Authentication chunk* prima di computare il *checksum* del pacchetto che andrá inserito poi nel *common header*. Come risultato, l'*Authentication chunk* non puó proteggere il checksum CRC32 dell'intestazione ma ne protegge tutti gli altri campi.

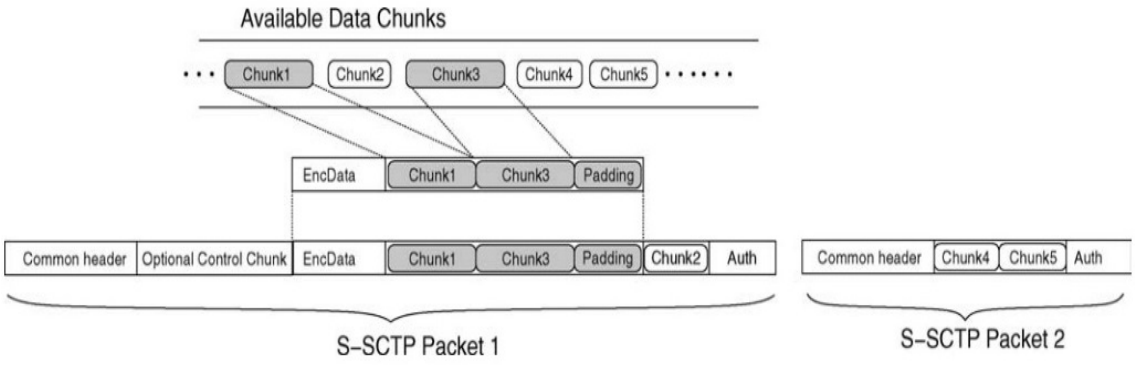


Figura 4.5, assemblamento chunks criptati e non

Come ultima osservazione, é importante sottolineare che i pacchetti S-SCTP sono *self-contained*, il che significa che tutte le informazioni richieste per decriptarli sono incluse nel pacchetto stesso. Inoltre l'allocazione delle funzioni crittografiche al modulo di assemblamento e disassemblamento dei pacchetti evita problemi legati a multipli textitstreams ed ordine di consegna e alla riconfigurazione dinamica degli indirizzi nel caso di multipli indirizzi IP.

Capitolo 5

SCTP e VoIP

Voice over Internet Protocol (VoIP) é una delle piú importanti applicazioni multipmediali al giorno d'oggi. Per nodi della rete mobili, in quest'ambito, é importante tenere in considerazione le varie questione legate all'*handoff* ossia dal cambiamento di canale di comunicazione di un nodo *wireless* che mantenga peró attiva la comunicazione. Percó SCTP e la sua estensione con DAR (*Dynamic Address Reconfiguration*) si sposa benissimo con le reti mobili *wireless* che fanno uso di VoIP. Tale tecnologia, infatti, con l'enorme sviluppo degli ultimissimi anni della tecnologia dei dispositivi mobili, é diventato un servizio molto importante e richiesto, soprattutto in ambito aziendale ed economico. Solitamente i servizi real-time (come VoIP) erano basati su TCP ma soprattutto su UDP il quale, grazie alla sua trasmissione orientata ai messaggi e senza connessione, consentiva una comunicazione di buona qualitá. Introducendo SCTP nella tecnologia VoIP sarebbe possibile migliorare questo tipo di qualitá, grazie al *Multihoming*, in modo particolare in ambienti *wireless* dove la congestione e la perdita di pacchetti sono parecchio frequenti (ricordiamo che SCTP, a differenza di UPD, sebbene orientato ai messaggi offre affidabilitá e ordine di trasmissione/ricezione), come é frequente la possibilitá di passare da un *access point* all'altro di reti eterogenee. Infatti, nel caso si verificasse, durante la comunicazione, il cambiamento dell'indirizzo IP di un dispositivo mobile, allora tale comunicazione verrebbe interrota e la chiamata perduta, qualora VoIP si basi su UDP (o TCP). Perció l'uso di SCTP combinato con DAR non provoca la chiusura della comunicazione/associazione qualora si verifichi un fenomeno di *handoff*.

5.1 SCTP e SIP

Essendo in ambito VoIP, é inveitabile riferirsi al *Session Initiation Protocol* (SIP), il protocollo di rete basato su IP che viene usato per la telefonia *over Internet*. SIP é un protocollo che permette la gestione di una sessione interativa di comunicazione tra due o piú entitá, ossia fornisce meccanismi per instaurare, modificare e terminare una sessione. Attraverso SIP possono essere trasferiti dati di diverso tipo (messaggi testuali, immagini ed audio-visivi) ed é quindi un protocollo per eccellenza dei servizi *real-time* e multimediali. In quanto protocollo applicativo SIP deve appoggiarsi su un protocollo di trasporto. Illustreremo di seguito i vari vantaggi di SIP su protocollo SCTP rispetto agli attuali TCP e UDP. Infatti, si é notato che molte delle caratteristiche di SCTP per i segnali telefonici SS7 si sono rivelate utili anche per il trasporto di SIP il quale comunque é indipendente rispetto al tipo di trasporto poiché funziona su qualsiasi tipo di trasmissione, affidabile o non. Di seguito mostriamo i vari vantaggi che SCTP dimostra di avere su VoIP rispetto a TCP e UDP.

UDP : SCTP puó determinare velocemente la perdita di pacchetti grazie al suo meccanismo di messaggi SACK i quali vengono inviati piú velocemente quando sono individuate perdite. Dunque le perdite di messaggi SIP possono essere individuate molto piú facilmente con SCTP rispetto a UDP il quale non ha controllo della congestione, e dunque non puó essere garantita la qualitá della trasmissione della voce. Pertanto, é compito dell'applicazione SIP implementare il proprio meccanismo di ritrasmissione per garantire affidabilitá. Per questo SIP, quando eseguito su UDP, offre un controllo di congestione poco efficiente. Questo si ha anche perché lo stato della congestione é misurato su una base di transazione in transazione, piuttosto che fra tutte le transazioni. Utilizzando SCTP invece. tale protocollo mantiene il controllo congestione per la durata dell'intera associazione. Per SIP ció significa che aggregazioni di messagg fra due entitá possono essere controllate. Infine, vi é il discorso della frammentazione del messaggio in pacchetti. In SCTP ció avviene a livello di trasporto mentre con UDP questo si verifica a livello IP il che aumenta la probabilitá che si verifichi perdita di pacchetti e di rendere piú difficile l'alltraversamento di NAT e *firewalls*. Questo aspetto é da tenere molto in considerazione nel caso in cui la dimensione dei pacchetti SIP aumenti considerevolmente. Perció sarebbe meglio offrire una frammentazione a livello di trasporto (offerta da SCTP ma anche da TCP).

TCP : SCTP é orientato ai messaggi, a differenza di TCP che é basato sullo *stream*. Questo permette a SCTP di separare i diversi messaggi di segnale telefonico (per le diverse destinazioni) subito al livello di trasporto, mentre TCP "capisce" solo flussi di bytes i quali devono essere assemblati dal destinatario applicazione telefonica. Da questo deduciamo che SCTP puó consegnare direttamente i messaggi all'applicazione senza ulteriori processi. In modo particolare, facendo uso del *Multistreaming*, in seguito a perdita pacchetti solo quelo *stream* resterá bloccato (per esempio se abbiamo uno *stream* per video ed uno per audio, la perdita di pacchetti interesserá solo uno dei due mentre l'altro continuerá indisturbato), a differenza del flusso continuo di bytes di TCP che si bloccherebbe completamente fino alla ritrasmissione di tutti i bytes mancanti (fenomeno dell'*Head of Blocking* sebbene altri bytes logicamente diversi (appartenenti a flussi di dati diversi) siano giá arrivati a destinazione, cosa che si verifica quando vi sono multiple connessioni ad alto livello multiplexate in un'unica connessione TCP. Una transazione SIP puó essere considerata come una connessione a livello applicativo. Poiché tra proxies vi sono molte transazioni in corso, la perdita di un messaggio di una transazione non dovrebbe avere effetti sfavorevoli riguardo alla possibilitá di altre transazioni di inviare messaggi. Perció, se SIP é eseguito tra entitá con molte transazioni in parallelo, SCTP puó fornire performance migliori rispetto a TCP. facendo proprio uso di diversi *streams*. In particolare, di default, due entitá SIP dovrebbero scambiarsi messaggi di controllo sullo *stream* 0, ed usare gli altri per inviare e ricevere segnali telefonici e voce. Infine, un altro vantaggio di SCTP rispetto a TCP é che, essendo basato su messaggi, consente un migliore *parsing* degli stessi a livello applicativo, senza bisogno quindi di stabilire confini tra un messaggio e l'altro.

N.B.: é importante notare che la maggior parte dei benifici di SCTP per SIP si hanno in condizioni di perdita di pacchetti. In un ambiente senza perdita pacchetti, le prestazioni di SIP su TCP/UDP sono le stesse che su SCTP. Quindi, la ricerca, attualmente, si sta focalizzando sul valutare sotto quali tipi di eventi di perdita tali benefici siano effettivamente misurabili e utili.

5.2 SIP e Multihoming

UDP, sebbene i suoi ridotti *overhead* ed efficienza, é considerato un protocollo arretrato per il trasporto in *real time* poiché i *payloads* multimediali sono inviati e ricevuti fuori sequenza. SCTP, tuttavia, ha la capacitá di

inviare i messaggi in sequenza, ma anche fuori sequenza (impostanto un parametro apposito a livello applicativo tramite la *system call sctp_sendmsg()*) mantenendo comunque l'affidabilitá all'interno degli *streams*. Ció é molto adatto per protocolli per contenuto multimediale tra diversi *devices* (device audio, device video...). SCTP usa dunque il suo meccanismo di HEARTBEAT per determinare ulteriori *path* dei *device* in modo da avere cosí una buona ridondanza sulla rete, in accordo con la logica dell'applicazione, cosí da potere essere in grado di continuare la sessione multimediale invece di invocare una chiamata di fallimento dovuta al rilevamento dell'inattivitá di un *device*.

Secondo gli ultimi studi, il *Multihoming* risulta essere molto utile quando si combina il VoIP assieme a due tipi diverse di reti *wireless*: il WiFi e il WiMax (nei paesi in cui é disponibile). Infatti, si é osservato che questioni legate a ritardo e a sovraccarico della rete sono meno influenti sulla comunicazione VoIP quando il device é connesso simultaneamente sia il WiFi che al WiMax. Infatti, connessioni concorrenti tra due diverse tecnologie wireless permettono ai dispositivi mobili di alternarsi tra questi due tipi di segnale e di sfruttare il WiFi, privilegiandolo ovunque possibile poiché offre trasmissioni e performance migliori rispetto al WiMax. Mantenendo un device connesso ad entrambi questi tipi di rete, si nota che il *throughput* rimane circa costante, qualora anche uno dei due canali fosse sovraccarico.

Nel corso della ricerca tra VoIP e SCTP sono state proposte diverse architetture di rete possibili per supportare tale servizio. Di seguito mi propongo di parlarne di una in particolare che integra SIP assieme ad un altro protocollo detto MPLS.

5.3 SCTP e MPLS nell'architettura VoIP

Come in tutte le reti di telefonia, anche nella VoIP bisogna fornire una certa qualitá del servizio voce (QoS, *Quality of Service*) secondo certi standard. Usare certi servizi integrati, come MPLS, possono fornire tale qualitá richiesta a VoIP. Il protocollo *MultiProtocol Label Switching* (MPLS) é un meccanismo ad alta performance, basata su reti IP, per le telecomunicazioni. Esso dirige i dati da un nodo della rete all'altro basandosi non sugli indirizzi di rete ma su delle etichette assegnate a coppie di *routers* adiacenti e semplici operazioni sulle etichette stesse, evitando cosí ricerche complesse nella tabella di *routing*. Le etichette infatti identificano link virtuali (*paths*) tra nodi della rete, piuttosto che tra endpoints. Usando MPLS l'*overhead* per la gestione dell'ottimizzazione delle risorse di rete e per il recupero veloce puó essere ottimizzato, fornendo anche una rigida QoS.

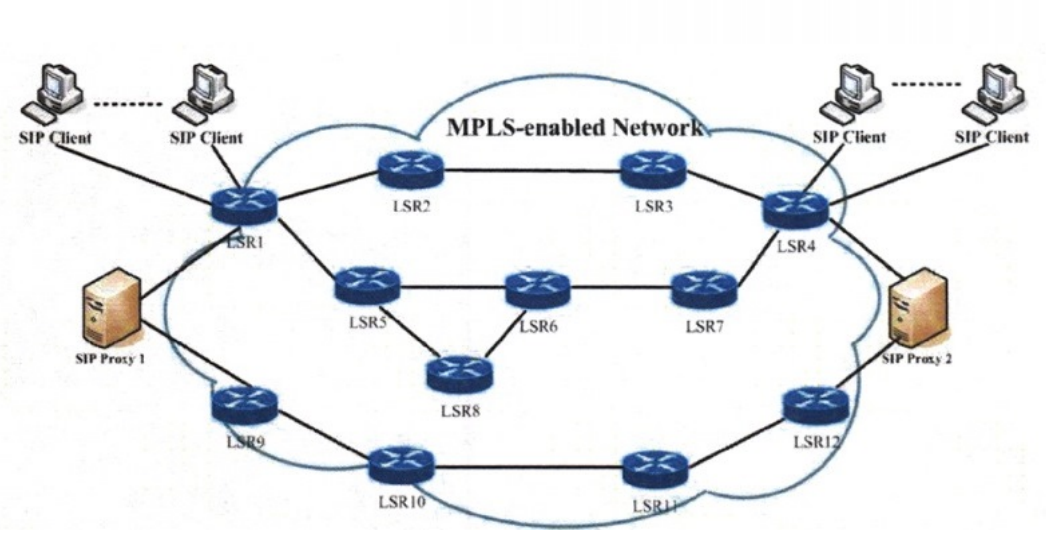


Figura 5.1, architettura SCTP-MPLS

Nell'architettura proposta (Figura 5.1), SCTP viene impiegato per trasmettere i segnali SIP. Inoltre, viene impiegato un *SIP Proxy Server multihomed* con due interfacce per la gestione della trasmissione degli stessi. Un'interfaccia é connessa al *path* primario, l'altra invece funge da *backup path*. Per massimizzare l'utilizzo delle risorse di rete, i messaggi di controllo come *SIP-INVITE* vengono trasmessi all'indirizzo primario mentre le relative risposte sono inviate con l'indirizzo di backup. Nel caso di una ritrasmissione del messaggio *SIP-INVITE*, esso verrebbe inoltrato anche all'indirizzo di backup cosí da ottenere performance migliori. In piú, per cercare di ridurre il fallimento di chiamate vocali e cercare di garantire quanta PIú *fault tolerance* possibile, i messaggi di *SIP-INVITE* vengono trasmessi all'indirizzo di backup qualora si rilevi un mal funzionamento o inattivitá del primario. Per il trasporto dati della voce, invece, si é deciso di tenerlo separato dai messaggi di controllo ed altri segnali, preferendo trasmetterlo via UDP a causa della ridotta dimensione dell'header dei suoi datagrammi. Infine, per garantire una buona qualitá trasmissione voce, é stato impiegato MPLS per il *setup* dei percorsi di trasporto dei dati voce. La proprietá di MPLS del re-instradamento (cambiamento di rotta) veloce, riduce di molto la perdita di pacchetti voce causa congestione o caduta di *links* fisici e/o nodi.

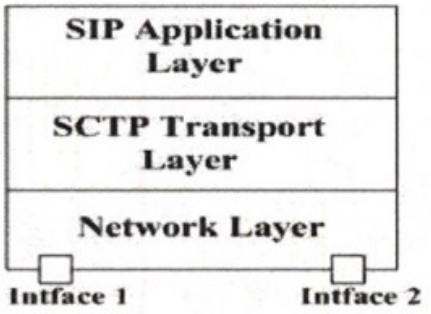


Figura 5.2, struttura del proxy multihomed

Dunque, l'architettura proposta contiene clients SIP, *multihomed* SIP *Proxy Servers* ed una rete configurata con MPLS che si occupa della comunicazione tra i clients SIP, il tutto configurato come descritto sopra.

Dopo aver ricevuto il messaggio *SIP-INVITE* da un client SIP, il Proxy 1 stabilisce un'associazione SCTP col Proxy 2 e poi gli inoltra tale richiesta al suo percorso di rete primario. A questo punto il Proxy 2, sempre utilizzando il suo indirizzo primario, lo inoltra al client SIP destinatario della chiamata. Se quest'ultimo accetta la chiamata, risponde con un messaggio *"200 OK"* al Proxy 2 che si occupa di inoltrare tale risposta al Proxy 1 (che inoltrerá al chiamante) utilizzando peró come detto prima, l'indirizzo di rete secondario di backup. Tutto questo é descritto dalla Figura 5.3.

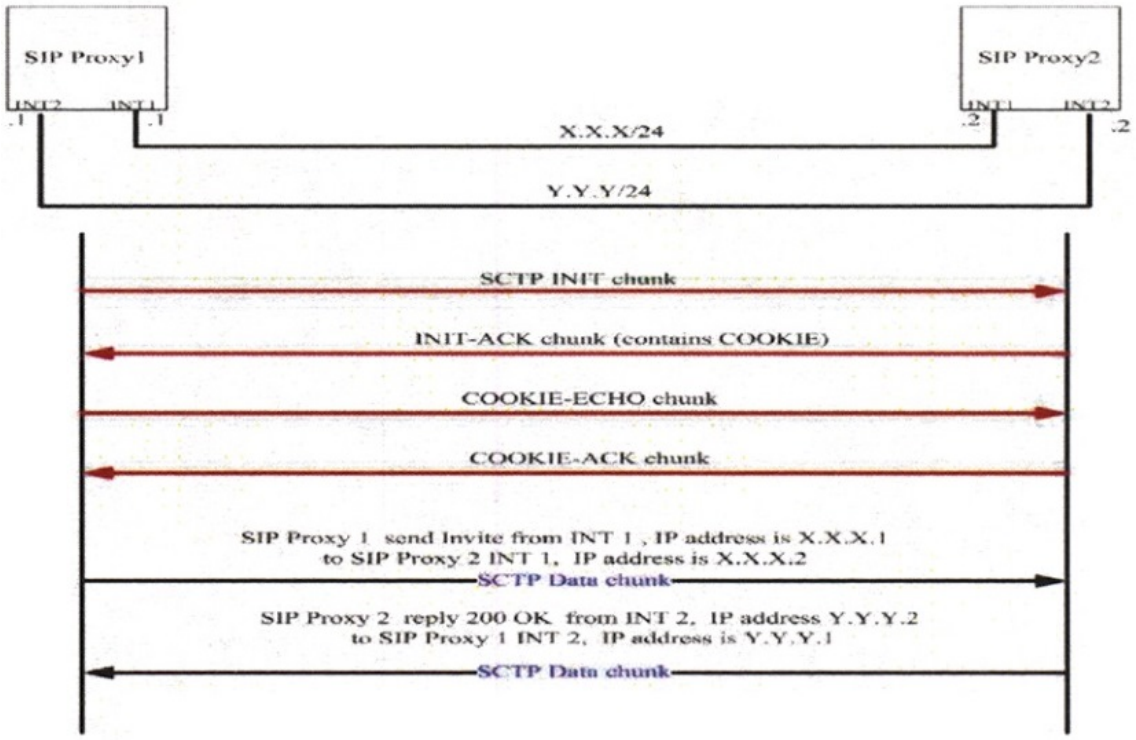


Figura 5.3, scambio di messaggi tra Proxy 1 e Proxy 2

Una volta che l'associazione tra i due Proxy é stabilita, vengono regolarmente scambiati i messaggi di *HEARTBEAT* e *HEARTBEAT ACK*. Se l'interfaccia primaria di rete dovesse risultare inattiva, il messaggio di textitSIP-INVITE, e relative risposte, sará trasmesso interamente con l'interfaccia secondaria e poiché l'associazione non deve essere ristabilita, nemmeno i segnali SIP sono interrotti. Tutti i successivi messaggi di *SIP-INVITE* verranno sempre trasmessi con l'interfaccia secondaria fino a quando la primaria non é di nuovo disponibile. Questo per evitare future inutili perdite di pacchetti.

Se osserviamo la Figura 5.1 noteremo che il miglior percorso tra i due Proxy é LSRl-LSR2-LSR3-LSR4 ed é proprio attraverso tale percorso che verranno solitamente inviati i pacchetti dati voce, una volta che é stato settato da MPLS. Nel caso venga rilevato l'inattivitá di un *link* fisico o di un nodo tra questi, allora verrá effettuato uno *switch routing* ad un percorso alternativo, tramite il meccanismo delle etichette tra *routers* adiacenti. Quindi il mantenere separato il traffico voce dai messaggi di controllo, consente un miglior e maggiore uso delle risorse di rete.

Una volta definita tale architettura, si sono eseguiti i testi comparando tre tipi di caso diversi:

- UDP-SIP: messaggi di controllo e dati voce trasmessi tutti da UDP;
- mSCTP-SIP: rete *multihomed* in cui SCTP é usato unicamente per messaggi di controllo, mentre il traffico voce 1'e affidato a UDP.
- mSCTP-SIP-MPLS: l'archiettura sopra descritta, ossia una rete con SCTP e *Multihoming* ma supportata e configurata da MPLS.

Di seguito, verranno comparate le relative performance di queste tre architetture in termini di *setup* della chiamata vocale e di quantitá di perdita/recupero pacchetti sotto certe condizioni.

Nella simulazione, si é generata una chiamata SIP per secondo, abilitando i Proxy a gestire al massimo 100 chiamate in contemporanea e generando in automatico traffico voce per una durata totale di circa 3 minuti. Nel caso della rete UDP-SIP, poiché non vi á alcun controllo di congestione e perdita pacchetti a livello di trasporto, tale meccanismo é stato abilitato a livello applicativo SIP a differenza degli altri due tipi di architetture dove ovviamente tale compito é lasciato a SCTP. Si é visto che il tasso di caduta delle chiamate é di circa lo 0.2% nell'architettura UDP-SIP risultando invece nulla nelle altre due. Simulando anche il guasto o l'inattivitá temporanea di un link e/o un nodo router, si é visto che il tasso di fallimento é dello 0.21% nell'architettura UDP-SIP e resta ancora nulla nelle altre due, grazie all'uso vantaggioso di MPLS e del *Multihoming* SCTP.
Infine, si é studiato anche quale sia, in media, il tempo richiesto per il *setup* di una chiamata. In generale, il tempo richiesto nella migliore delle ipotesi é di 1 secondo mentre vengono considerati inacettabili i tempi maggiori di 5 secondi. Secondo le sperimentazioni, sia in termini di congestione rete sia in condizione di guasto di *link* fisici, l'architettura che risponde meglio é ovviamente la mSCTP-SIP-MPLS subito seguita dalla mSCTP-SIP mentre nella semplice architettura UDP-SIP il tempo di *setup* richiesto é sempre almeno di circa 5 secondi. Questo poiché sia i segnali SIP che il traffico dati voce vengono trasmessi sullo stesso *path* ed é l'applicazione SIP stessa che si preoccupa della trasmissione. Grazie al solo uso di SCTP si puó ottenere una alta *fault tolerance* e tempi di *setup* minimi, soprattutto nella rete mSCTP-SIP-MPLS la quale separa i messaggi di segnalazione SIP dal traffico voce. Le Figure 5.4 e 5.5 mostrano questi risultati. Dunque tramite questi meccanismi e grazie alle associazioni SCTP é possibile fornire allocazioni di risorse di reti anche per VoIP (ricordiamo che VoIP su UDP, come standard, non fornisce allocazioni di risorse di rete in quanto UDP non é orientato alla connessione) le quali danno una forte tolleranza ai guasti e problemi di altro genere. Tuttavia, per ottenere una stretta QoS come richiesto, sarebbe una buona idea applicare il protocollo MPLS assieme al VoIP e SCTP.

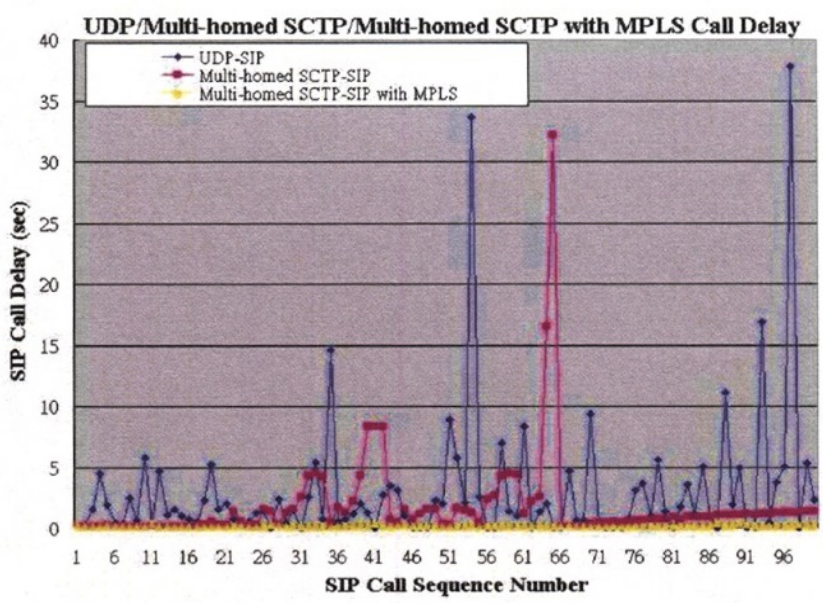


Figura 5.4, Voice call setup time as network congestion

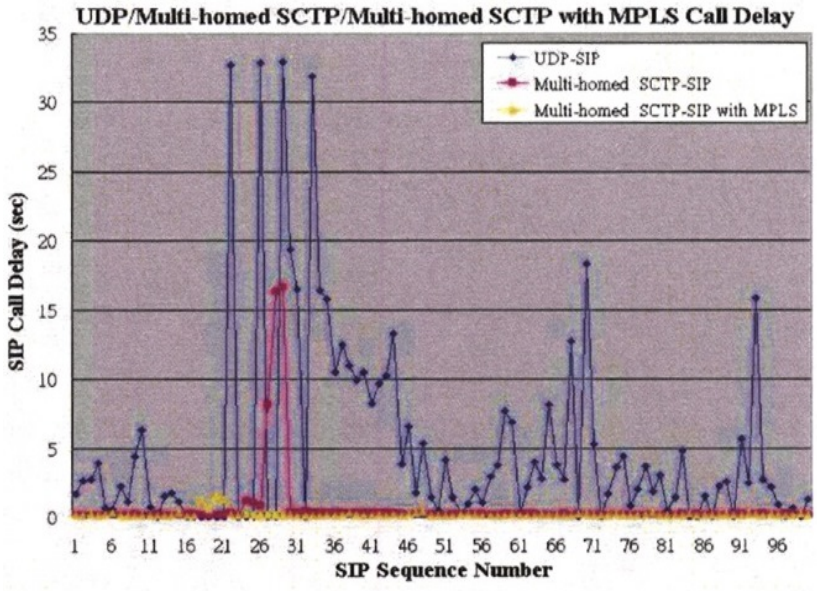


Figura 5.5, Voice call setup time as network link failure

Capitolo 6

Conclusioni

Al giorno d'oggi i requisiti di comunicazione sono cambiati moltissimo rispetto a tempo fa. In modo particolare sono entrati in campo il concetto di ***Big Data*** e le comunicazioni ***Inter Data-Center*** nell'ambito del ***Cloud Computing***. Nella definizione generalmente accettata i *Big Data* sono gli insiemi di dati talmente grandi che é molto difficoltoso se non impossibile utilizzarli, ma questa definizione é doppiamente inadeguata. In primo luogo perché, con il progressivo miglioramento di hardware e software, il concetto di "troppo grande" deve essere continuamente rivisto al rialzo; e poi anche perché quando si parla di Big Data non lo si fa per lamentarsi ma per tentare di trovare un modo di utilizzare le tecnologie hardware e software esistenti per manipolarli. In effetti il concetto *Big Data* Ã¨ semplicemente l'altra faccia della medaglia di *data analytics*, perché l'analisi dei dati usa varie modalitá di suddivisione o selezione tra grandi quantitá di dati per individuare i frammenti di informazione pertinenti o rilevanti per una particolare richiesta. Ogni giorno la miniera dei Big data si arricchisce di nostre informazioni prodotte da social media, blog, portali di e-commerce, siti di informazione, motori di ricerca. Enormi aggregazioni di dati che raccolgono gli umori della rete. Tanto grandi e complessi che richiedono, per essere trattati e analizzati, strumenti tecnologicamente avanzati. Ma i *Big Data* stanno anche diventano un problema importante, da risolvere al piú presto. Man mano che vengono raccolte piú informazioni e che aumentano i collegamenti alla rete Internet, tutto diventa un "problema da *Big Data*", anche il controllo della rete stessa.

La domanda che quindi sorge spontanea é:
Puó il protocollo SCTP venire incontro a tali esigenze moderne? Le due figure sottostanti ci mostrano l'utilizzo di SCTP con *Multistreaming* comparato con TCP nell'ambito dello scambio di files di piccole e grosse dimensioni.

La Figura 6.1 A, che mostra lo scambio di files da 10 KB fino a 1 MB, ci suggerisce che SCTP potrebbe essere un ottimo protocollo per la navigazione

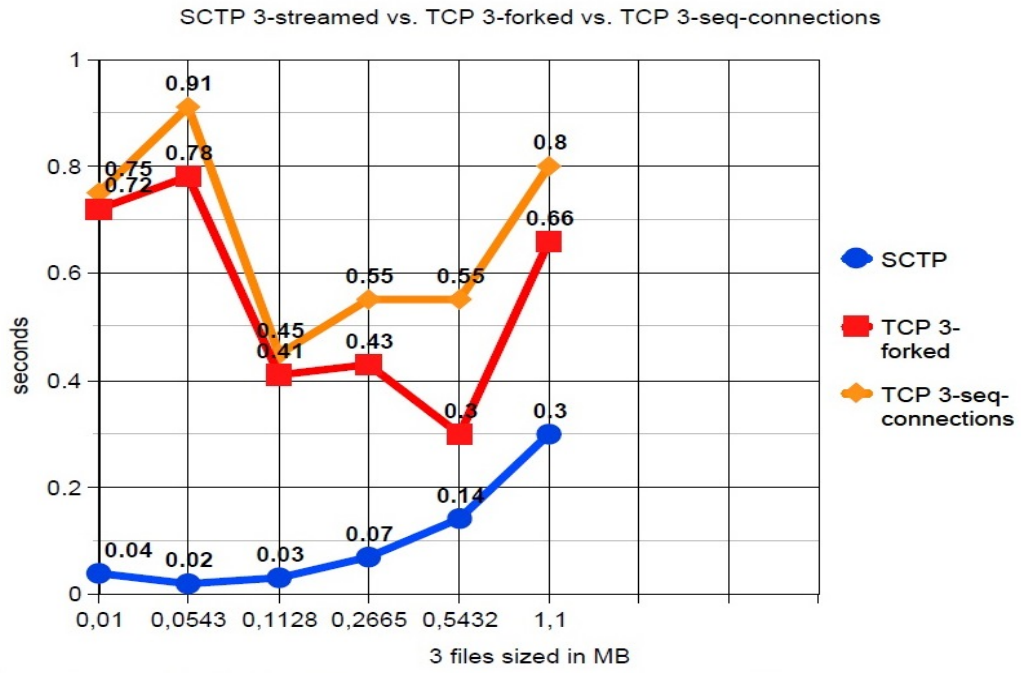


Figura 6.1 A, SCTP e TCP in trasferimento di 3 piccoli files

Web. Infatti HTML e *script* hanno dimensione di pochi KB ed utilizzando correttamente il *Multistreaming* si otterrebbe l'esecuzione di *script* e caricamento di pagine Web in pochi centesimi di secondo rispetto invece a TCP. Come supporto ai *Big Data*, osservando la Figura 6.1 B, possiamo stabilire che SCTP non sia ancora abbastanza ottimizzato e performante essendo dunque in uno stadio ancora un pó "acerbo" per supporto a tale situazione odierna. In effetti, pur essendo piú performante di n connessioni sequenziali TCP, SCTP con n *streams* rimane comunque piú lento rispetto a n connessioni parallele TCP quando i files entrano nell'ordine di dimensione di centinaia di MB. Pertanto, oggi come oggi pare non esserci alcun beneficio significativo per SCTP nell'ambito dei *Big Data*. Potremmo eventualmente giustificare l'uso di SCTP in questo ambito qualora si volesse dare sia piú sicurezza alla rete nel trasporto di queste massicce quantitá di dati (con *Multihoming*) sia avere una migliore reazione alla congestione della rete (poiché solo gli *streams* congestionati subiscono ritardi, e non tutti gli *streams* dell'associazione in quanto sono indipendenti l'uno dall'altro).

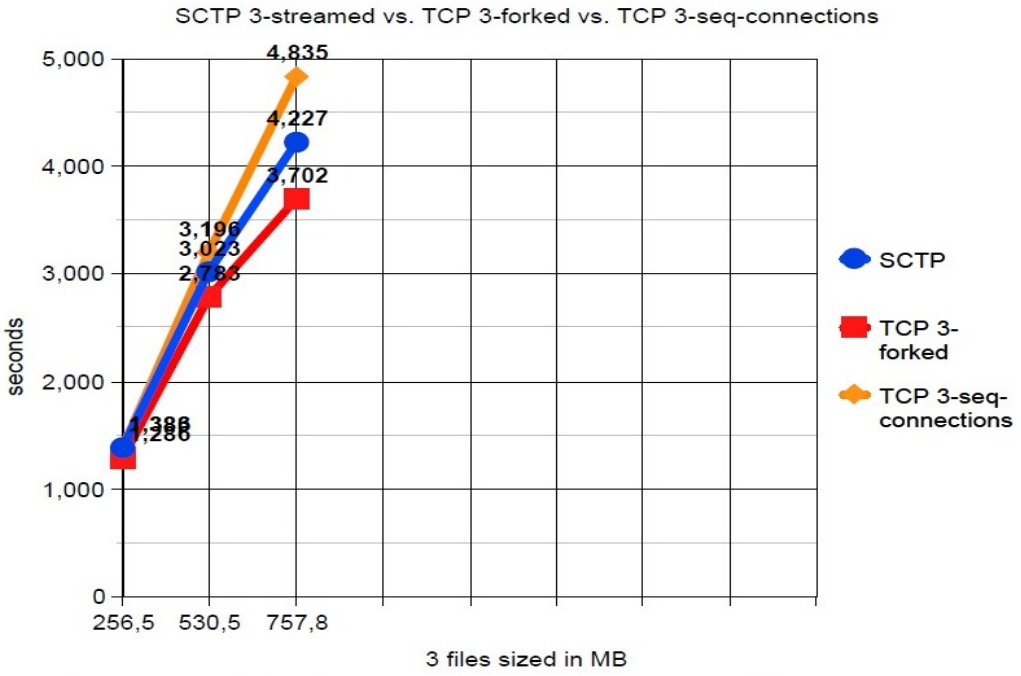


Figura 6.1 B, SCTP e TCP in trasferimento di 3 grossi files

Non dobbiamo poi dimenticare che, al giorno d'oggi, é diventata di fondamentale l'importanza la tecnologia del *Could Computing*, ossia la possibilitá di sfruttare risorse software ed hardware installati su server remoti e quindi distribuiti e virtualizzati nella rete, per elaborare ed archiviare i propri dati utente, invece che sul proprio computer locale. In questo contesto, possiamo dire che il *Multihoming* di SCTP si presta molto bene al supporto per tale tecnologia. Infatti, fornendo a tali risorse distribuite multiple interfacce di rete diverse, nel caso della caduta di una di queste, l'elaborazione e la comunicazione con l'utente potrá continuare indisturbata senza perdite di dati o errori che ne comprometterebbero la consistenza. Il *Multihoming* di SCTP puó quindi offrire una solida robustezza e sicurezza della rete per questo tipo di attivitá.

Infine, é importante ricordare un'ultimo grande vantaggio che SCTP puó offrire alla tecnologia moderna: **il supporto alla mobilitá**. Nell'era degli *smartphones* é importante poter garantire il loro corretto funzionamento con la rete, senza la quale uno *smartphone* non sarebbe considerato tale poiché la maggior parte delle sue applicazioni richiedono accesso alla rete. Trattandosi di tecnologia mobile, uno *smartphone* puó passare da un'*access point* all'altro, a causa del "movimento" del suo utente. In questo caso, qualora vi fosse in corso un'attivitá, come per esempio una comunicazione VoIP oppure

una navigazione in internet "delicata" (come un acquisto online su eBay o Amazon), essa verrebbe inevitabilmente compromessa ed annullata nel momento in cui, per esempio, viene meno il segnale WiFi e lo *smartphone* passa su rete 3G o ad un altro *access point* Wifi. Perció, Il concetto di mobilitá si riferisce alla capacitá di un utente di spostarsi da una rete all'altra con il suo dispositivo, mantenendo attive tutte le comunicazioni ed attivitá in corso. Esiste un termine tecnico per descrivere questo spostamento, in base a come é "percepito" dal nostro dispositivo: ***handover***. Quest'ultimo puó essere orizzontale quando ci spostiamo fra diverse reti di tipo omogeneo (ad esempio: spostarsi da un *access point* Wifi all'altro camminando per Bologna) o verticale quando si cambia effettivamente l'interfaccia di rete utilizzata, passando ad un'altra tecnologia (da Wifi a 4G e da 4G a 3G). In entrambi i casi ció che ci interessa é la possibilitá di mantenere le nostre connessioni di rete attive durante un *handover*, minimizzando il ritardo introdotto e le modifiche necessarie ad applicazioni o infrastrutture (protocolli pre-esistenti e routers). Vogliamo quindi poter mantenere una chiamata Skype, un download HTTP in corso o la sessione di gioco online attive durante perdite di connessione, dovute ad un *handover*, di durata fortemente dipendente dai requisiti dell'applicativo che stiamo utilizzando (e-mail vs. VoIP). Tutto questo ci porta all'ovvia conclusione che la problematica principale é il cambio di indirizzo IP del nostro dispositivo, nel momento in cui esso cambia interfaccia di rete. Poiché il nostro indirizzo IP serve a localizzarci e ad identificarci all'interno della sottorete nella quale ci troviamo, é ovvio che se ne assumiamo uno nuovo allora l'host con cui comunicavamo non riesce ad associarci alla precedente comunicazione e finirá per chiuderla, costringendoci a reinizializzarla usando il nuovo indirizzo. Dunque, SCTP ed il *Multihoming* possono aiutarci a migliorare tale situazione poiché SCTP appartiene a quella classe di architetture di supporto alla mobilitá denominata ***pure end-to-end***, in quanto non introducono agenti esterni tra i nodi che comunicano (come un Proxy tra il nostro dispositivo mobile e l'host a cui siamo connessi) e non modificano in alcun modo le infrastrutture sottostanti.

Tuttavia, occorre ancora molto lavoro, come sviluppo futuro, per rendere utilizzabile SCTP in questo contesto. Prima di tutto, entra in ballo la modifica dell'applicazione (del suo codice sorgente) in quanto, usando il protocollo SCTP, é diverso sia il tipo di *socket* come sono diverse le primitive API da usare per usufruire del *Multihoming*, a differenza della attuali TCP. Bisognerebbe dunque, in qualche modo, lasciare inalterate le applicazioni rispetto al livelo di trasporto, eventualmente cercando di includere i loro pacchetti TCP dentro pacchetti SCTP. Infine, un'altra problematica riguarda *firewalls simmetrici* nei *gateway* di accesso a sottoreti, che rendono impossibile l'uso del protocollo poiché nel momento che si verifica un cambio di indirizzo (e

quindi la coppia *(IP-address, port)*) di un host, il *firewall* dell'altro host potrebbe scartare tale messaggio che avvisa del cambiamento in quanto visto come pacchetto spontaneo. Effettuare tali modifiche su larga scala nella rete e nei suoi punti di accesso (come i *providers* di servizi internet) richiede sforzi numerosi e soprattutto molto costosi. La domanda che rimane aperta é:
quanti cambiamenti bisogna apportare nel mondo circostante per usare con successo questo protocollo?. In effetti, SCTP é ancora un protocollo abbastanza "acerbo" per essere usato a tutti gli effetti come protocollo di trasporto al pari di TCP e UDP, se non al posto di essi. Tuttavia, esso rappresenta comunque un'innovazione importante verso la nuova era di Internet, basata sempre di piú su dispositivi mobili e sulla telefonia attraverso Internet.

Capitolo 7

Bibliografia

- **RFC 3286**
  An Introduction to the Stream Control Transmission Protocol (SCTP)
  Reperibile presso:
  http://www.ietf.org/rfc/rfc3286.txt

- **RFC 2960 e RFC 4960: Stream Control Transmission Protocol**
  Reperibili presso:
  http://www.ietf.org/rfc/rfc2960.txt http://tools.ietf.org/html/rfc4960
  e *errata corrige* presso:
  http://tools.ietf.org/html/rfc2960#section-6.9

- **RFC 5061**
  Stream Control Transmission Protocol (SCTP)
  Dynamic Address Reconfiguration (DAR)
  Reperibile presso:
  http://www.ietf.org/rfc/rfc5061.txt

- **Sourabh Ladha, Paul D, Amer**
  "Improving File Transfers Using SCTP Multistreaming"
  Protocol Engineering Lab, CIS Department, University of Delaware
  Anno 2003 pagine 1-5

- **RFC 3554: On the Use of Stream Control Transmission Protocol (SCTP) with IPsec**
  Reperibile presso:
  http://www.ietf.org/rfc/rfc3554.txt

- **RFC 5062: Security Attacks Found Against the Stream Control Transmission Protocol (SCTP) and Current Countermeasures**
  Reperibile presso:
  http://www.ietf.org/rfc/rfc5062.txt

- **ESBOLD UNURKHAAN, ERWIN P. RATHGEB and ANDREAS JUNGMAIER**
  "Secure SCTP - A Versatile Secure Transport Protocol" Computer Networking Technology Group, IEM, University of Duisburg-Essen.
  Anno 2004 Pagine 1-24

- **Secure SCTP**
  draft-hohendorf-secure-sctp-12.txt
  Reperibile presso:
  http://tools.ietf.org/html/draft-hohendorf-secure-sctp-12

- **Media Multihoming in SIP Sessions**
  draft-rverma-media-multihoming-over-sctp-00.txt
  Reperibile presso:
  http://tools.ietf.org/html/draft-rverma-media-multihoming-over-sctp-00

- **RFC 4168: The Stream Control Transmission Protocol (SCTP) as a Transport for the Session Initiation Protocol (SIP)**
  Reperibile presso:
  http://tools.ietf.org/html/rfc4168

- **Carsten Hohendorf, Erwing P. Rathgeb, Esbold Unurkhaan, Michael Tuxten**
  "Secure End-to-End Transport Over SCTP"
  University of Duisburg-Essen: Institute for Experimental Mathematics, Computer Networking Technology Group and
  Mongolian Science and Technological University: Computer Science and Management School and
  Munster University of Applied Sciences.
  Journal of Computer Science vol. 2, No. 4, Giugno 2007 Pagine 31-38

- **LUKASZ BUDZISZ, Technische Universitat Berlin, JOHAN GARCIA and ANNA BRUNSTROM, RAMON FERRÚS**
  "A Taxonomy and Survey of SCTP Research"
  Karlstad University, Universitat Politénica de Catalunya e Technische Universistat Berlin. ACM Comput. Surv. 44, 4, Articolo 18 di Agosto 2012

- **RFC 793: TRANSMISSION CONTROL PROTOCOL**
  Reperibile presso:
  http://www.ietf.org/rfc/rfc793.txt
- **"RFC 2581: TCP Congestion Control"**
  Reperibile presso:
  http://tools.ietf.org/html/rfc2581
- **Shaojian Fu and Mohammed Atiquzzaman**
  "SCTP: State of the Art in Research,
  Products, and Technical Challenges" Shaojian Fu and Mohammed Atiquzzaman, University of Oklahoma.
  IEEE Communication Magazine, Aprile 2004 Pagine 64-75
- **Lin-huang Chang, Po-Hsun Huang,**
  Hung-Chi Chu, Huai-Hsinh Tsai
  "Mobility Management of VoIP services using SCTP Handoff Mechanism"
  Department of Computer and Information Science, National Taichung University, Taichung, Taiwan . Department of Information and Communication Eng., Chaoyang University of Technology, Taichung, Taiwan. Conferenza in data : 7-9 July 2009 pagine 330-335
- **Fu-Min Chang, I-Ping Hsieh, Shang-Juh Kao**
  "Adopting SCTP and MPLS-TE Mechanism in VoIP
  Architecture for Fault Recovery and Resource
  Allocation"
  Department of Finance, Chaoyang University of Technology, Taiwan.
  Department of Computer Science and Engineering, National Chung Hsing University, Taiwan.
  pagine 1-5
- **"RFC 1323: TCP Extensions for High Performance"**
  Reperibile presso:
  http://www.ietf.org/rfc/rfc1323.txt
- **"RFC 1122: Requirements for Internet Hosts – Communication Layers"**
  Reperibile presso:
  http://tools.ietf.org/html/rfc1122
- **RFC 2409: The Internet Key Exchange (IKE)**
  Reperibile presso:
  http://tools.ietf.org/html/rfc2409

- **Larry L. Peterson, Bruce S. Davie**
  "Reti di Calcolatori"
  APOGEO, anno 2008

- **Andrew S. Tanenbaum, David j. Wetherall**
  "Reti di Calcolatori"
  PEARSON, quinta edizione