ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Dipartimento di Informatica - Scienza e Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

TESI DI LAUREA

in

Protocols And Architectures For Space Networks M

Progetto di una implementazione veloce del Licklider Transmission Protocol per link ottici

CANDIDATO:

Dott. Andrea Bisacchi

RELATORE:

Prof. Carlo Caini

Anno Accademico: 2019/2020

A mia mamma e Furio, per ciò che
siete e per ciò che avete sempre fatto per me.
"E che vittoria sia, giorno dopo giorno!" cit.

Indice

Prefazione

L’innata ambizione dell’essere umano verso l’esplorazione dell’ignoto, unita al progredire delle conoscenze scientifiche, ha portato l’umanità a muoversi oltre ai confini di questo pianeta dapprima con semplici osservazioni, quindi grazie ai formidabili sviluppi tecnologici dell’ultimo secolo con satelliti artificiali e sonde per arrivare allo straordinario 20 luglio 1969, quando per la prima volta un uomo posò il piede sulla Luna. A questo grandioso evento seguì l’invio di innumerevoli rover sui pianeti del nostro sistema solare che hanno permesso di conoscerli “da vicino”, per arrivare alla nuova sfida di portare l’uomo su Marte.

Verso la fine degli anni ‘90 Vinton Cerf sviluppò l’idea di estendere la rete Internet terrestre allo spazio, creando così il concetto di “Internet Interplanetaria” (InterPlanetary Networking, IPN): ciò perché l’architettura TCP/IP, da lui stesso in parte progettata, non è adatta a far fronte ai problemi delle comunicazioni nello spazio interplanetario quali lunghi ritardi, alti tassi di perdita e connessioni intermittenti. L'osservazione che questi problemi sono comuni anche ad altre reti “challenged” ha portato alla creazione dell’architettura DTN, nata per rispondere non solo alle esigenze delle reti IPN ma, più in generale, a tutte le reti challenged presenti anche in scenari terrestri, sia in ambito civile che militare.

All’interno dell’architettura DTN, il Licklider Transmission Protocol (LTP) si colloca come protocollo di trasporto ideato specificamente per funzionare sulle tratte spaziali caratterizzate da lunghi ritardi e possibili interruzioni della connessione, dove il TCP non può essere utilizzato. Su queste tratte l’utilizzo di link ottici è molto promettente, in quanto consente di raggiungere velocità di trasmissione superiori di diversi ordini di grandezza rispetto ai normali canali a radiofrequenza: parliamo di circa un Gbit/s in confronto a poche centinaia di kbit/s raggiungibili con la tecnologia precedente.

Questa tesi ha avuto come obiettivo la creazione di una nuova implementazione dell’LTP, denominata Unibo-LTP, capace di sfruttare meglio le velocità offerte dai link ottici rispetto all’implementazione contenuta in ION, cioè la suite dei protocolli DTN realizzata dalla NASA-JPL (National Aeronautics and Space Administration, Jet Propulsion Laboratory).

Il lavoro è stato organizzato e suddiviso in quattro fasi: studio del protocollo, progetto e scrittura del codice, valutazione delle prestazioni ed, infine, realizzazione di nuove funzionalità sperimentali.

Durante la prima fase si sono studiate le specifiche del protocollo LTP contenute nell'RFC 5326 e si è analizzata in dettaglio l'implementazione presente in ION.

La seconda ha portato alla realizzazione del codice di Unibo-LTP, garantendo la piena compatibilità con la suite DTN della NASA. A differenza di quella originale, l'implementazione attuale è stata progettata per funzionare come un unico processo pesante composto da più thread, così da poter processare contemporaneamente più segmenti.

La terza è dedicata alla realizzazione di test comparativi tra le velocità raggiungibili dalle due implementazioni, realizzati tra macchine reali su sistema operativo GNU/Linux connesse con un hardware che consente velocità vicine al Gbit/s, così da simulare l'utilizzo di link ottici.

Infine nell'ultima fase sono state realizzate alcune estensioni dello standard, in particolare quella dei segmenti "orange", da proporre per l'inserimento nello standard CCSDS attualmente in fase di revisione.

1 Introduzione

1.1 Delay-/Disruption-Tolerant Networking (DTN)

L'architettura di rete DTN (Delay-/Disruption-Tolerant Networking) è stata ideata per rendere possibile la comunicazione nelle reti "challenged", ossia reti caratterizzate da:

- elevato tempo di propagazione
- disconnessioni improvvise
- canale di comunicazione intermittente e asimmetrico

Ciò era diventato necessario perché i normali protocolli TCP/IP che utilizziamo quotidianamente non sono in grado di far fronte efficientemente ai tre aspetti sopra citati: si basano infatti sia sul fatto che una comunicazione tra due macchine è possibile solo a condizione che esista sempre un percorso continuo che le connette, sia che sia fattibile e conveniente la ritrasmissione dei pacchetti persi a partire dalla sorgente (principio "end-to-end"). L'architettura DTN supera questo limite e permette la comunicazione anche quando le due macchine non sono sempre connesse con continuità, anzi è addirittura possibile mandare dati da una macchina che non esiste più o verso una macchina che ancora non esiste: questo avviene dando la possibilità ai nodi intermedi di memorizzare i pacchetti (chiamati *bundle*) e di inviarli al nodo successivo alla prima occasione utile. Analogamente, l'architettura DTN rende possibile il superamento di lunghi ritardi tramite una ridefinizione del ruolo dello strato di trasporto e l'impiego di protocolli progettati ad hoc, come il Licklider Transmission Protocol (LTP), oggetto di questa tesi.

Le situazioni che presentano uno o più dei problemi sopra citati sono molteplici e comprendono sia le comunicazioni spaziali (satellitari ed interplanetarie) sia terrestri: reti militari tattiche, reti di emergenza (si pensi cosa può accadere in caso di grandi calamità naturali, quali ad esempio terremoti o inondazioni che possono causare interruzioni delle comunicazioni anche per tempi prolungati), reti sottomarine, comunicazioni in regioni prive di infrastruttura di comunicazioni (regioni polari, desertiche, ecc.), reti di sensori wireless: tuttavia il presente lavoro si focalizza esclusivamente sull'ambito spaziale, considerando in particolare l'utilizzo del protocollo Licklider in combinazione a link ottici, spiegati in dettaglio nella sezione 1.2.

In questo capitolo viene approfondita la struttura, il processo di standardizzazione in atto e le attuali implementazioni delle DTN.

1.1.1 Struttura

È importante collocare l'architettura DTN nel più importante standard per la progettazione di reti: il modello ISO-OSI (Open Systems Interconnection), normalmente utilizzato per Internet ed altre reti terrestri.
DTN crea un nuovo livello, denominato Bundle Layer, che rappresenta il livello relativo al Bundle Protocol (BP) e che si colloca subito sopra il Trasporto, di cui deve necessariamente utilizzare i servizi offerti.

Il BP non è un protocollo *end-to-end*, tuttavia è possibile che la comunicazione attraversi diversi *hop* DTN, i quali tramite un algoritmo di *routing* consentono di inoltrare i bundle al successivo nodo DTN. Tra un hop DTN e l'altro si possono utilizzare protocolli di trasporto differenti; ciò è reso possibile dalla ridefinizione del ruolo di trasporto, che ora non è più end-to-end ma è confinato all'interno di un hop DTN: apposite interfacce, chiamate Convergence Layer Adapter (CLA), forniscono al BP un'astrazione dei sottostanti protocolli di Trasporto. La figura sottostante mostra quattro nodi DTN e le relative pile OSI, comprensive di Convergence Layer Adapter, indicati con colori diversi proprio per sottolineare quanto appena detto.

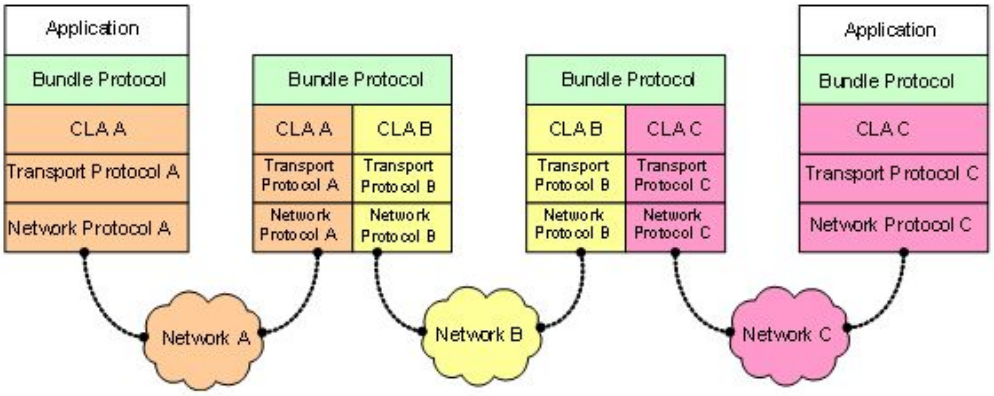

Figura 1. Esempio di collegamento di quattro nodi DTN che utilizzano tecnologia DTN, formanti 3 hop DTN. L'esempio mostra le macchine collegate tra di loro tramite tre pile di protocolli differenti. Il Network A può comprendere, o meno, nodi non DTN intermedi, non rappresentati nella figura e trasparenti al Bundle Protocol.

I protocolli di trasporto in questo contesto vengono anche detti "Convergence Layer"; tra questi troviamo i comuni protocolli di trasporto che si utilizzano quotidianamente, cioè il TCP e l'UDP solitamente utilizzati per collegare hop DTN terrestri, mentre quello più adatto per le comunicazioni interplanetarie è l'LTP.

1.1.2 Standardizzazione

Nell'ambito delle telecomunicazioni è importante avere un protocollo che sia il più possibile standard, così da facilitare l'interoperabilità di implementazioni differenti: per questo motivo il Bundle Protocol ha subìto e sta tuttora subendo importanti processi di standardizzazione.

Cronologicamente, il primo documento è l'RFC (Request For Comment) 4838 [1], dedicata alla descrizione dei principi dell'architettura DTN, avente come coautore Vinton Cerf, coautore anche del TCP/IP, pioniere sia di Internet che delle reti interplanetarie (IPN). Essa è stata seguita quasi immediatamente dall'RFC 5050 [2] che standardizza il Bundle Protocol (versione 6, o BPv6).

Si sono susseguite poi diverse RFC su argomenti vari, come ad esempio gli aspetti relativi alla sicurezza del protocollo.

Nel 2015 il BP è stato anche standardizzato dal CCSDS (Consultative Committee for Space Data Systems, organizzazione comprendente le principali agenzie spaziali) nel Blue Book (standard) [3] attualmente in fase di revisione.
Mentre le prime RFC erano state rilasciate dall'IRTF (Internet Research Task Force), con il progredire della ricerca e delle esperienze la standardizzazione è passata in ambito IETF (Internet Engineering Task Force), dove sono in fase finale di standardizzazione la versione 7 del BP (BPv7) [4], le estensioni di sicurezza (BPsec) [5] ed una nuova versione del CLA del TCP [6].

L'Università di Bologna è da anni attiva nell'ambito di ricerca delle DTN ed ha fornito diverse soluzioni, contribuendo in modo significativo alla definizione degli standard attuali; si veda al riguardo [7].

1.1.3 Implementazioni

Esistono diverse implementazioni dei protocolli DTN; quelle più importanti, tutte rilasciate come software libero, sono tre:

- ION (Interplanetary Overlay Network), sviluppata dalla NASA-JPL;
- DTN2, sviluppata inizialmente dai laboratori Intel presso l'Università di Berkeley e recentemente portata avanti da NASA-MSFC con il nome DTNME;
- IBR-DTN, sviluppata da un gruppo di ricerca dell'Università di Braunschweig.

Il lavoro presentato in questa tesi sarà oggetto di studio, approfondimento e integrazione con ION [8] nella sua ultima versione (3.7.3); inoltre va precisato che grazie alla totale modularità del codice sviluppato potrebbe in futuro essere facilmente inserito nelle altre implementazioni DTN.

1.2 Link ottici

Per collegare fra di loro dei nodi spaziali, oltre all'utilizzo dei normali canali a radiofrequenza è possibile anche utilizzare link ottici a propagazione libera, molto interessanti in ogni situazione nella quale, pur non essendo utilizzabili le fibre ottiche, si vogliono raggiungere alte velocità di trasmissione: i primi infatti hanno velocità molto variabili ma nello spazio tipicamente limitate alle decine, massimo centinaia di kbit/s; i secondi arrivano all'ordine di circa un Gbit/s.

Le comunicazioni ottiche hanno svariate caratteristiche che le rendono particolarmente interessanti in ambito spaziale, in particolare velocità di comunicazione molto alta, resistenza ad interferenze elettromagnetiche e maggiore sicurezza nel canale. Sebbene questi benefici siano importanti bisogna considerare anche gli aspetti negativi: il canale è asimmetrico; inoltre, essendo richiesta una linea di vista continua tra i due terminali è sufficiente anche solo una nuvola per bloccare il raggio ottico. Le tipologie di perdite sono quindi estremamente differenti nei due sistemi: nei canali a radiofrequenza le perdite sono di norma uniformemente distribuite nell'arco dell'intera comunicazione; nei link ottici spaziali, con un terminale a terra, si possono invece trovare perdite concentrate e prolungate dovute a fenomeni atmosferici.

2 Licklider Transmission Protocol

Il Licklider Transmission Protocol (LTP) è un protocollo di trasporto progettato per essere utilizzato come Convergence Layer nelle reti caratterizzate da ritardi estremamente elevati con interruzioni deterministiche, cioè in pratica nelle reti spaziali. Esistono tre RFC che lo descrivono: la 5325 [9] spiega le motivazioni, la 5326 [10] dà le specifiche e quella successiva, la 5327 [11], definisce le estensioni di sicurezza; oltre a questi esiste anche un Blue Book emesso dal CCSDS [12], attualmente in fase di revisione.

2.1 Differenza tra TCP e LTP

Poiché LTP è stato creato per risolvere i limiti di TCP, nella tabella seguente vengono riportate le principali differenze tra i due protocolli.

TCP	**LTP**
Bidirezionale.	Unidirezionale dal punto di vista logico (il ritorno è usato solo dalla segnalazione).
Instaurazione della connessione tramite *three way handshake*.	Assenza di procedura di apertura della connessione. I dati vengono inviati quando esiste un contatto programmato (scheduled).
La velocità di trasmissione è dinamicamente determinata dal controllo di flusso e di congestione.	La velocità di trasmissione è dettata dalla velocità nominale indicata dal contatto in uso.
I dati trasmessi sono un flusso di byte.	I dati da trasmettere sono incapsulati in un *blocco* (in modo simile a quanto avviene con i *datagram* UDP).
Il servizio offerto è un servizio affidabile.	Il servizio può essere sia affidabile (*red*), sia non affidabile (*green*).
Basato su una molteplicità di interazioni tra i due nodi.	Pensato per ridurre al minimo il numero di interazioni tra i due nodi.

Richiede un RTT molto corto, tipicamente [1, 200] millisecondi.	Può funzionare con RTT di decine di minuti.

Tabella 1. Le principali differenze tra i protocolli TCP e LTP.

2.2 Spiegazione del protocollo

LTP organizza i dati in *blocchi* con dimensioni anche molto grandi; ogni blocco è composto in generale da due parti, una *red* ed una *green*. La prima consente di avere un servizio affidabile, pertanto su quei dati sono richieste delle conferme (*acknowledgment)* ed in caso di mancato arrivo entro un lasso di tempo predefinito è prevista la ritrasmissione. La seconda offre un servizio non affidabile, quindi senza ritrasmissioni.
L'invio di un blocco avviene tramite una nuova *sessione* dedicata, che viene identificata da due numeri: il numero del nodo che ha creato la sessione ed un numero progressivo.
Un blocco viene suddiviso in *segmenti dati*, red o green a seconda della parte del blocco a cui appartengono. Oltre ai segmenti dati esistono i *segmenti di segnalazione*, generalmente molto piccoli, che permettono lo scambio di informazioni per garantire l'affidabilità della parte red; questi sono:

- *Checkpoint* (CP): questo è l'unico segmento "ibrido", contenente sia dati red sia la richiesta di conferma dei dati trasmessi fino a quel punto.
- *Report segment* (RS): conferma i dati ricevuti fino a quel momento. Esso consiste nell'elenco dei blocchi di byte ricevuti, detti *claim*. Normalmente questo segmento viene mandato in risposta ad un CP.
- *Report-acknowledgment segment* (RAS): conferma di ricezione di un RS.
- *Cancel segment from block sender* (CS): richiesta di cancellazione di una sessione da parte del mittente.
- *Cancel-acknowledgment segment to block sender* (CAS): conferma di ricezione di un CS.
- *Cancel segment from block receiver* (CR): richiesta di cancellazione di una sessione da parte del ricevente.

- *Cancel-acknowledgment segment to block receiver* (CAR): conferma di ricezione di un CR.

Alcuni segmenti di segnalazione prevedono una ritrasmissione dopo un RTO (Retransmission TimeOut) in caso di mancata conferma, ovvero:

- I CP prevedono ritrasmissione se non vengono confermati da un RS;
- Gli RS prevedono ritrasmissione se non vengono confermati da un RAS;
- I Cancel Segment (entrambi i tipi) prevedono ritrasmissione se non vengono confermati dai relativi Cancel-acknowledgment Segment.

2.3 Esempi di funzionamento

Una sessione LTP viene creata nel momento in cui una macchina, che in questi esempi verrà chiamata Sender, intende trasferire dei dati verso un'altra macchina, chiamata Receiver; più precisamente il momento della creazione della sessione è diverso tra le due macchine: il Sender è la macchina che crea la sessione, il Receiver ne verrà a conoscenza solo all'arrivo del primo segmento dati relativo a quella sessione. Una sessione rimane attiva finché dal lato del Sender non verranno ricevute le conferme relative all'intera parte red e non sarà stata trasmessa tutta la parte green, dal lato Receiver finché non saranno stati ricevuti tutti i segmenti dati relativi a quella sessione.

Il protocollo LTP è molto articolato, a volte perfino troppo; per descriverlo faremo quindi riferimento a degli esempi considerando la presenza della sola parte red in quattro casi di complessità crescente:

1. Condizioni ideali (assenza di perdite);
2. Perdita esclusivamente di segmenti dati;
3. Perdita di segmenti dati e di segnalazione;
4. Perdita del RAS finale.

2.3.1 Condizioni ideali (assenza di perdite)

Questo caso è illustrato nella Figura 2. Il Sender genera i segmenti dati contrassegnando l'ultimo segmento come CP; in questo modo si richiede la conferma dei dati. L'ultimo segmento trova impostati due ulteriori *flag*: End Of Red Part (EORP) e End Of Block

(EOB) di cui il primo indica il termine della parte red, il secondo indica che il blocco è terminato e che quello era l'ultimo segmento. In presenza di condizioni ideali, cioè un'assenza assoluta di perdite, il Receiver riceve ogni segmento; è quindi in grado di ricostruire il blocco e consegnare i dati contenuti al livello soprastante senza richiedere ritrasmissioni: tuttavia il CP mandato dal Sender richiede una risposta, pertanto viene generato un Report Segment che indica che tutto è stato ricevuto correttamente. Alla ricezione dello stesso da parte del Sender, questo può considerare chiusa la propria sessione di trasmissione; contestualmente invia un RAS in risposta al RS ricevuto.

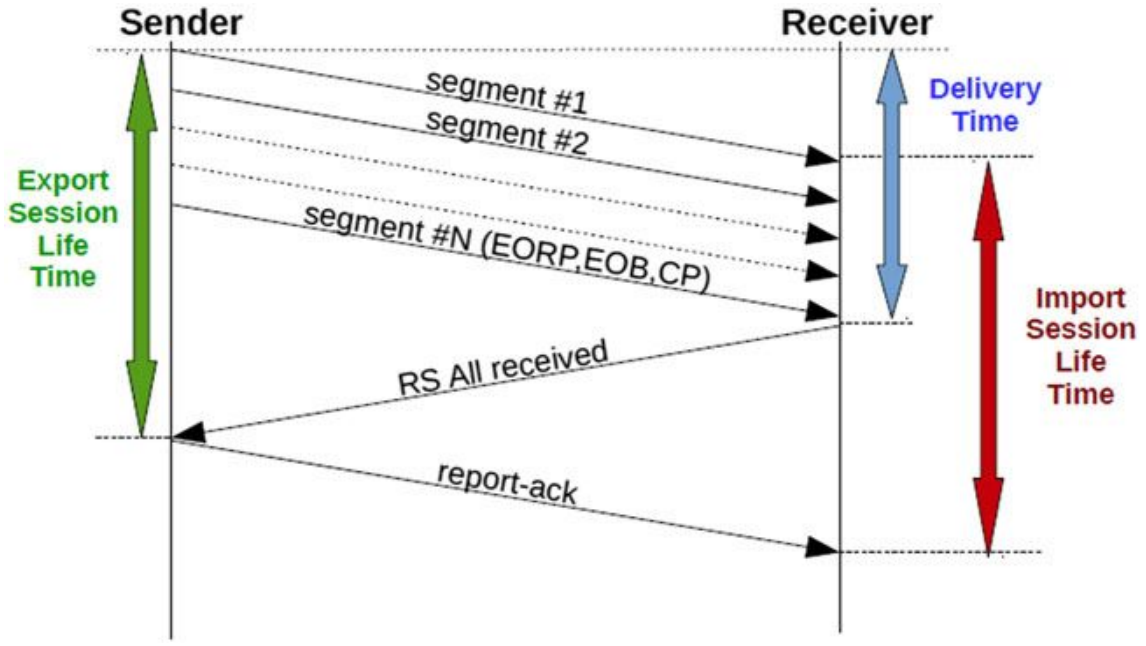

Figura 2. Esempio di una sessione LTP (red) in caso di canale ideale (assenza di perdite).

2.3.2 Perdita esclusivamente di segmenti dati

Si veda la Figura 3: come nell'esempio precedente, il Sender genera i segmenti da trasmettere contrassegnando l'ultimo come CP, EORP ed EOB. Questa volta un sottoinsieme di segmenti trasmessi viene perso, nell'esempio i segmenti numero 2, 5 e 6: il Receiver genera un Report Segment che indica che sono stati ricevuti tutti i segmenti, ad eccezione del 2, 5 e 6; alla ricezione del RS, il Sender invia il RAS, quindi provvede alla immediata ritrasmissione dei segmenti non confermati, contrassegnando l'ultimo segmento come CP: questo iter consente di proteggere la ritrasmissione dei dati. Se questa volta i

segmenti ritrasmessi vengono ricevuti correttamente il protocollo si conclude similmente a quanto visto in precedenza, ovvero con l'invio di un RS che conferma la ricezione dell'intero blocco al Sender, il quale alla sua ricezione considera chiusa la sessione di invio (export session); in caso contrario la procedura appena descritta sarebbe stata ripetuta fino ad ottenere la corretta ricezione dei dati.

Per precisione è necessario specificare che è stata fatta una semplificazione concettuale in quanto il protocollo attuale non numera i segmenti, pertanto le conferme si fanno indicando l'intervallo dei *byte* che si vogliono confermare.

È importante inoltre sottolineare che i RS danno informazioni di ciò che è stato ricevuto e non delle mancanze, ovvero sono dei cosiddetti "positive ack".

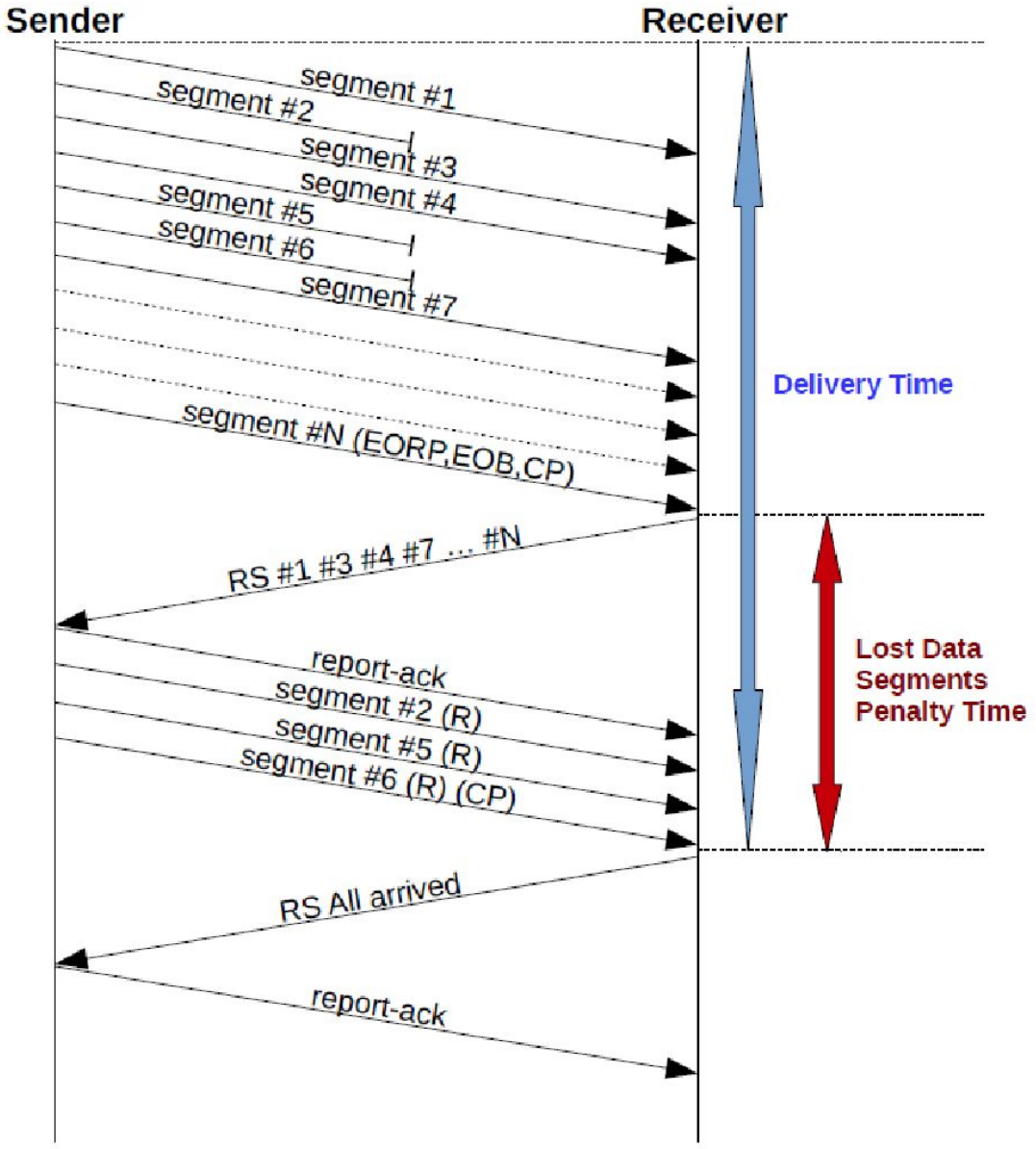

Figura 3. Esempio di sessione LTP (red) in caso di perdite di soli segmenti dati.

2.3.3 Perdita di segmenti dati e di segnalazione

Anche in questo terzo esempio (Figura 4) il Sender genera i segmenti da trasmettere indicando l'ultimo come CP, EORP e EOB; tuttavia in questo esempio viene perso proprio tale segmento, oltre a qualche altro segmento dati precedente. Mantenendo la semplificazione concettuale spiegata nell'esempio precedente si ha la perdita dei segmenti 2, 3 e del CP finale: la perdita del CP implica la mancata generazione del RS da parte del Receiver. La sessione va pertanto "in stallo" per un RTO, finché il *timer* di ritrasmissione

del Sender non scatta e non viene quindi nuovamente inviato il segmento contrassegnato come CP (ed in questo caso anche come EORP ed EOB). Una volta arrivato correttamente il CP, il Receiver crea e trasmette un RS e da questo momento in poi la sessione continua come nell'esempio precedente.

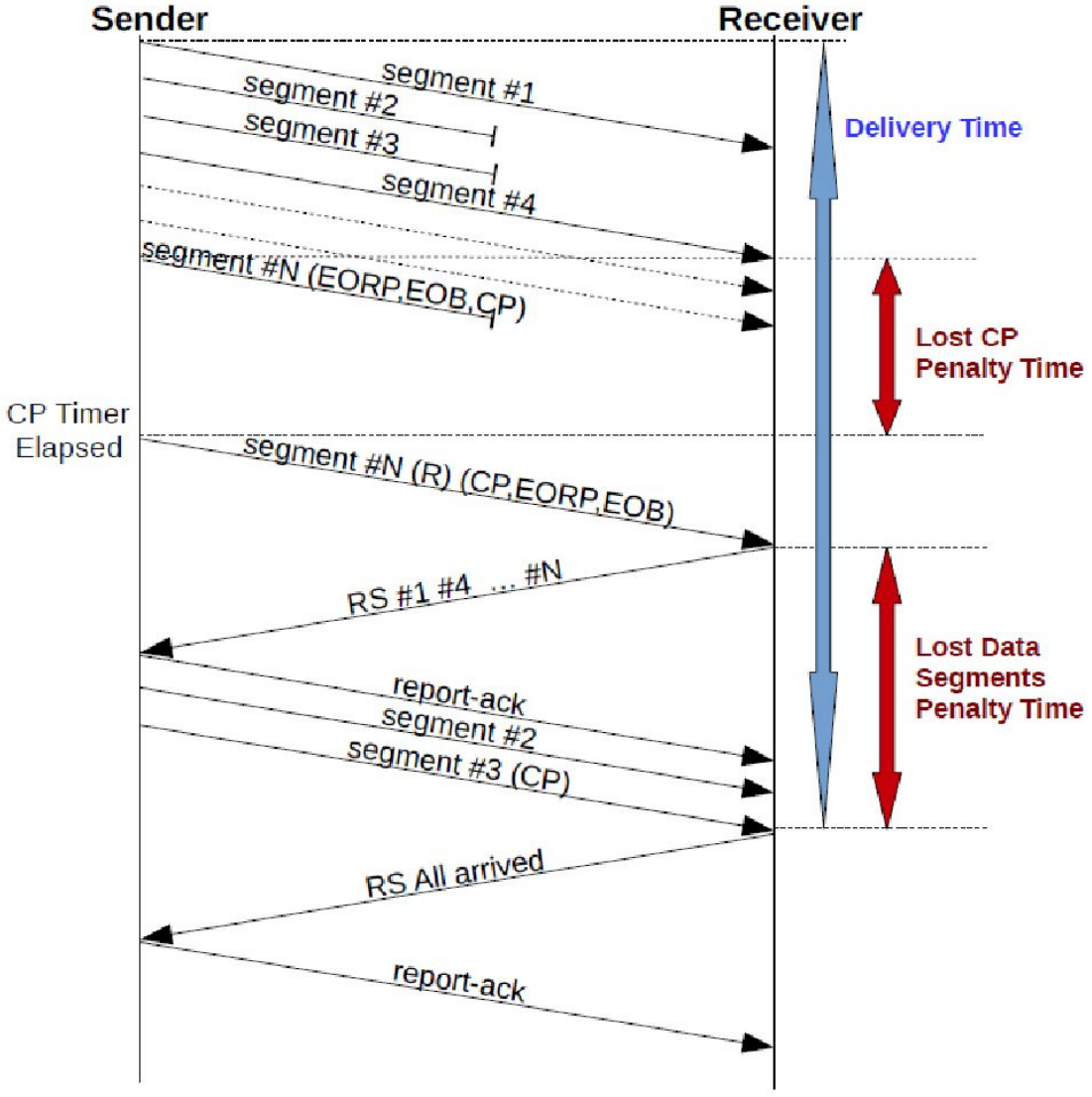

Figura 4. Esempio di sessione LTP (red) in caso di perdite di segmenti dati e del primo CP inviato.

L'esempio appena descritto può essere facilmente generalizzato alla perdita di uno qualunque dei segmenti di segnalazione; questo comporterà un ritardo nella consegna dei dati pari ad un RTO.

2.3.4 Perdita del RAS finale

Si considera ora l'eventualità in cui viene perso il RAS di conferma dell'ultimo RS, ovvero quello riportante l'avvenuta ricezione dell'intero blocco. Sebbene sia un'eventualità relativamente rara (la probabilità che accada è uguale alla probabilità di perdita di un segmento) è da tenere comunque in considerazione perché le specifiche dell'LTP contengono un baco a questo proposito (manca l'equivalente del timer "time wait" del TCP). La perdita dell'ultimo RAS comporta, dopo un RTO, la ritrasmissione del RS da parte del Receiver; tuttavia il Sender, nell'algoritmo originale, non è più in grado di riconoscere la sessione alla quale fa riferimento il RS e non risponde con un RAS, dando luogo ad una serie di ritrasmissioni successive dell'RS fino al raggiungimento del numero massimo di ritrasmissioni ed alla conseguente cancellazione da parte del Receiver. Questo problema è stato risolto dall'Università di Bologna in [13] memorizzando le sessioni terminate di recente, così da poter trasmettere un RAS nel caso in cui si riceva un RS dopo la chiusura della sessione di invio.

2.4 Implementazioni

L'implementazione dell'LTP più importante trattata in questa tesi è quella presente all'interno di ION. LTP è stato implementato come Convergence Layer per il Bundle Protocol; l'implementazione realizzata dalla NASA è conforme all'RFC 5326 ed include una serie di ottimizzazioni, la più importante delle quali è la possibilità di aggregare più bundle all'interno di un unico blocco LTP. LTP utilizza un Link Service Adapter (LSA) per interfacciarsi al protocollo inferiore, che può essere UDP o un altro protocollo conforme al CCSDS; l'implementazione di ION contiene anche come opzioni tutte le estensioni proposte in [13].

Il mio lavoro, illustrato nei capitoli seguenti, è finalizzato alla creazione di una nuova implementazione di LTP ideata per funzionare come CL del Bundle Protocol, autonoma ma integrabile in ION e che possa superare la velocità massima di trasmissione dell'implementazione di LTP presente in ION con l'obiettivo di essere capace di sfruttare meglio la banda fornita dai link ottici.

3 Progetto di Unibo-LTP

Vediamo quindi i passi che hanno portato alla nuova implementazione dell'LTP, denominata Unibo-LTP, in grado di superare le velocità raggiungibili da quella attualmente presente in ION.

3.1 Analisi dell'implementazione di riferimento

Il primo è stato l'analisi dell'implementazione LTP di riferimento, cioè quella originale in ION sviluppata da Scott Burleigh, coautore anche delle specifiche dell'LTP. In ION l'LTP è implementato come libreria e viene utilizzato come tale dai Link Service Adapter (LSA), ovvero dai protocolli sottostanti; analizzando ciò che avviene con l'LSA di UDP troviamo due applicazioni, una di ricezione (*udplsi*) ed una di invio (*udplso*).

Le operazioni svolte dalla prima sono:

1. ricezione di un *datagram* UDP
2. passaggio ad LTP del segmento LTP contenuto nel payload del pacchetto UDP ricevuto
3. e così via ripartendo dal punto 1.

La seconda, invece, esegue queste operazioni:

1. recupero da LTP del segmento LTP da trasmettere
2. incapsulamento ed invio del segment LTP in un datagram UDP
3. e così via dal punto 1.

L'intera logica è concentrata all'interno della libreria LTP.

Il problema evidenziato nei test preliminari ad "alta velocità" riguardava perdite UDP interne al buffer del ricevitore: in pratica, i segmenti UDP arrivavano ad una velocità superiore alla velocità massima con la quale erano drenati dal buffer UDP, causando perdite interne. Per capire meglio l'origine di ciò la prima operazione svolta è stata l'introduzione di stampe del tempo richiesto dall'LTP per effettuare il *parsing* e la gestione dei segmenti ricevuti, al fine di analizzare possibili rallentamenti e trovarne le funzioni responsabili; per svolgere queste analisi, in cui si è utilizzato l'applicativo Valgrind [14] in

modalità *profiling*, è emerso che la causa principale delle perdite era dovuta al fatto che in ION una volta letto un segmento LTP, il segmento successivo non viene estratto dal buffer UDP prima di aver terminato il processamento del precedente. Il tempo di processamento, a sua volta, non è costante per ogni segmento, ma dipende dal tipo di segmento ricevuto, la qual cosa causa delle fluttuazioni nella velocità di estrazione dei segmenti dal buffer, con perdite quindi abbastanza casuali. Il processamento di un segmento, inoltre, è rallentato dall'utilizzo estensivo di una particolare memoria condivisa utilizzata per la sincronizzazione di tutti i processi di ION, la memoria SDR (Spacecraft Data Recorder): essa funziona come una specie di database molto robusto, ma è anche lenta e difficilissima da debuggare.

Si è quindi optato per procedere alla realizzazione di un'implementazione dell'LTP svincolata dalle caratteristiche peculiari di ION, e quindi anche da SDR. Avere un'implementazione "propria" inoltre fa sì che la si possa usare come base di partenza per svolgere future analisi e dà la possibilità di testare nuove funzionalità, eventualmente da proporre per l'inserimento nella versione rivista dello standard ufficiale CCSDS [12].

3.2 Requisiti

Per raggiungere gli obiettivi prefissati è stato necessario soddisfare una serie di requisiti, primo tra tutti il raggiungimento di velocità superiori rispetto all'implementazione di ION.

Altro importante requisito è la compatibilità con ION, che rende possibile utilizzare Unibo-LTP in un BP oltre a rendere più accurati i test effettuati sulle prestazioni.

È stato inoltre utilizzato come linguaggio di programmazione il C, in quanto usato sia in ION sia nelle diverse applicazioni DTN realizzate all'interno dell'Università di Bologna.

Infine, riguardo la concorrenza si è scelto di utilizzare un singolo processo e suddividerlo in più thread, piuttosto che realizzare tanti processi separati (come viene fatto in ION): ciò ha consentito una gestione parallela dei segmenti ricevuti ed ha semplificato notevolmente la comunicazione tra i diversi thread, fattori questi che hanno contribuito a raggiungere velocità superiori.

3.3 Astrazione del protocollo superiore ed inferiore

Unibo-LTP è fornito di due moduli, uno per il livello superiore ed uno per quello inferiore, per poter disaccoppiare il "cuore" dai protocolli specifici utilizzati.

Il primo consente di avere un'astrazione del BP superiore (attualmente implementato per funzionare con ION) tramite l'implementazione di un adapter. Da questo, oltre all'invio ed alla ricezione di bundle, è possibile recuperare il contact plan, ovvero l'insieme dei contatti e dei range; questi dati servono anche ad Unibo-LTP, pertanto si è deciso di recuperarli direttamente dal protocollo superiore, in modo da utilizzare gli stessi dati che il BP già usa.

Il modulo per il protocollo inferiore è invece più complesso, in quanto si è lasciata libertà di scegliere in ogni nodo "vicino" quale protocollo inferiore utilizzare: ogni protocollo inferiore prevede la creazione di un adapter specifico. L'implementazione attuale ne fornisce due, uno per l'UDP ed uno per ECLSA [15] (sviluppato all'interno dell'Università di Bologna e presente nella cartella *contrib* rilasciato all'interno della versione ufficiale di ION e sul quale ho lavorato durante la tesi conclusiva del corso di laurea triennale); questo permette di scegliere protocolli specifici adatti ad ogni evenienza, ad esempio si può usare l'UDP su link dove le perdite sono rare ed ECLSA (che implementa un codice a correzione d'errore) su link soggetti a perdite.

3.4 Timer

Come precedentemente descritto, l'LTP prevede di ritrasmettere i segmenti di segnalazione in caso di mancata ricezione degli appositi segmenti di conferma entro un tempo limite; pertanto è stato necessario implementare dei timer che possono essere avviati, interrotti, messi in pausa e rimessi in esecuzione. La creazione dei timer consente di facilitare la gestione delle ritrasmissioni e consente di eseguire funzioni dopo un certo lasso di tempo (ad esempio l'aggiornamento del contact plan prendendolo dal Bundle Protocol, oppure l'eliminazione di una sessione dall'elenco delle sessioni terminate di recente).

3.5 Segmenti, sessioni e span

I concetti chiave dell'LTP sono tre: segmenti, sessioni e "span".

I *segmenti* sono gli elementi alla base dell'LTP; come abbiamo visto nel Capitolo 2, mittente e destinatario si scambiano segmenti (dati o di segnalazione) che riguardano una *sessione*, identificata dal numero della macchina ("engine") che l'ha originata e da un numero progressivo. Una medesima sessione (trasferimento di un blocco LTP) viene vista come "sessione di invio" nel nodo mittente, e come "sessione di ricezione" nel nodo ricevente, come descritto nel capitolo precedente.

La comunicazione fra due nodi LTP non viene negoziata ma è fissata a priori nel file di configurazione; in particolare, in ION si ha una istruzione "span" per ogni possibile destinatario. In questa istruzione vengono indicati il numero (engine) del destinatario, il numero massimo delle sessioni di invio per quel destinatario, quello delle sessioni di ricezione (nelle ultime versioni il numero dei buffer di ricezione), due parametri di aggregazione (non considerata in questa tesi), la dimensione massima di un segmento LTP, il protocollo sottostante (ad esempio udplso con i suoi parametri), il tempo di processamento nominale nel nodo ricevente (per determinare il valore dei timer di ritrasmissione). In Unibo-LTP uno span richiede meno parametri, ma affinché la comunicazione fra il nodo X ed il nodo Y sia possibile occorre, così come in ION, che nel nodo X sia definito uno span con destinatario Y e viceversa. In pratica un nodo deve avere tanti span quanti possibili corrispondenti.

3.6 I diversi thread

Vengono ora descritti in dettaglio i thread e il loro funzionamento.

3.6.1 Il "Receiver thread"

Il thread principale è quello di ricezione, unico per tutti i mittenti. È anche il più critico in quanto è richiesto che la sua velocità sia la più elevata possibile, pertanto si è cercato di rendere semplice la sua logica e si è deciso di fargli svolgere solo poche operazioni:

1. Ricezione dei segmenti LTP dal protocollo inferiore, in generale appartenenti a più sessioni, non necessariamente originate dallo stesso mittente (engine).
2. Parsing dei segmenti LTP ricevuti.
3. Ricerca dello span relativo a quella sessione; se lo span non viene trovato il segmento ricevuto viene scartato.

4. (Se il passo 3 ha trovato lo span) Gestione dei segmenti, differenziando l'operazione svolta a seconda che sia stato ricevuto un segmento dati o un segmento di segnalazione. Nel primo caso lo si gestisce subito, inserendolo nell'elenco dei segmenti ricevuti di quella sessione, ovvero nel buffer di ricezione per quella sessione; nel secondo si delega la sua gestione allo span thread corrispondente (vedi sezione successiva).

   Si noti che in caso di ricezione di un CP si effettuano entrambe le operazioni, in quanto i CP sono segmenti "misti" ovvero segmenti dati che contengono anche segnalazione.

La seguente figura riassume e schematizza il funzionamento del Receiver thread:

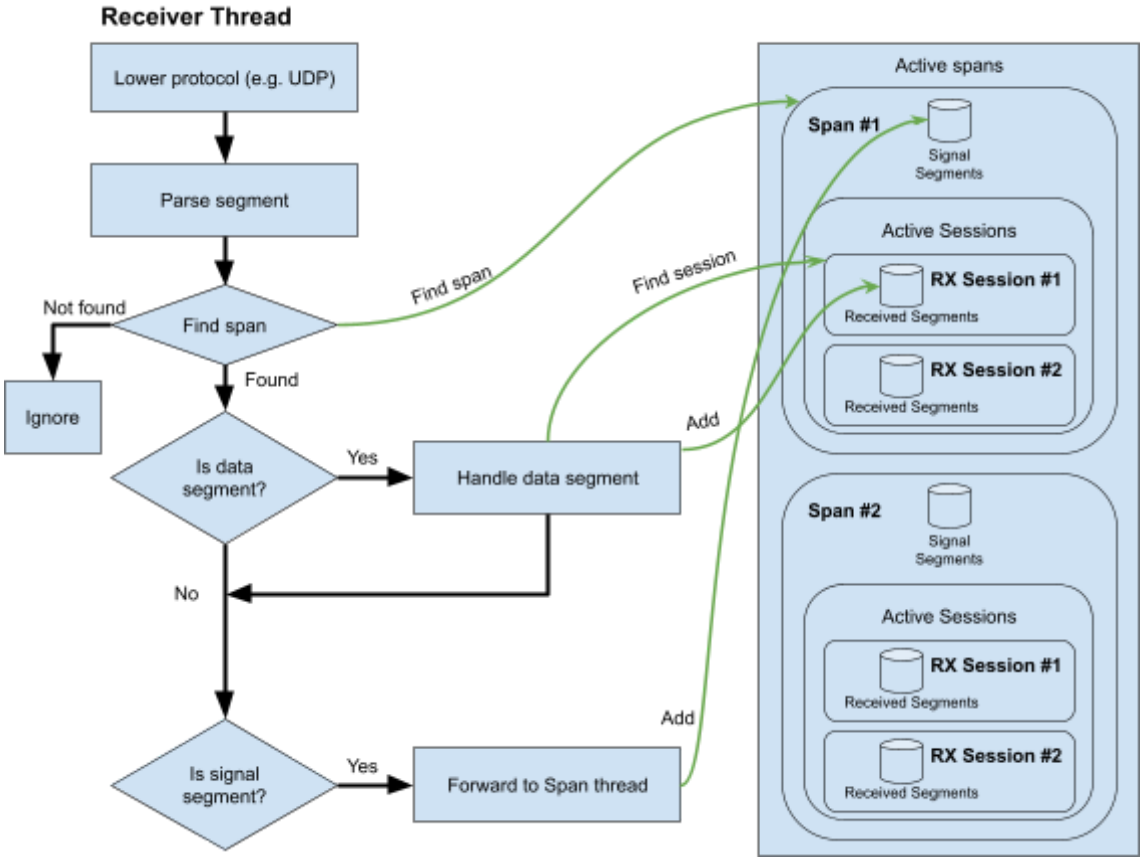

Figura 5. Operazioni svolte dal Receiver thread. Una volta ricevuto il segmento LTP ne effettua il parsing e procede alla gestione: se è un segmento dati lo inserisce nel buffer di ricezione della sessione corrispondente, se è un segmento di segnalazione non lo gestisce ma lo passa direttamente allo Span thread relativo a quel mittente. Se il segmento è un CP, vengono effettuate entrambe le cose.

3.6.2 I thread per la gestione di un corrispondente

I thread che consentono la gestione della comunicazione con un medesimo corrispondente sono tre e vengono descritti in dettaglio nelle sottosezioni che seguono; saranno attivi (non sospesi) finché il contact plan indica che esiste un contatto con il nodo gestito da essi.

3.6.2.1 Lo “Span thread”

È il più importante e svolge due operazioni:

1. gestione dei segmenti di segnalazione
2. creazione delle sessioni di trasmissione (se ci sono blocchi da trasmettere)

La prima serve a gestire i segmenti di segnalazione in arrivo da quello corrispondente, passati dal Receiver thread. È opportuno distinguere i segmenti da processare a seconda se si riferiscono, rispetto al nodo corrente, ad una sessione di ricezione o di trasmissione.

Sessione di ricezione:

- RAS: conferma la ricezione di un RS, pertanto viene immediatamente bloccata la ritrasmissione del RS medesimo.
- CP: se il CP fa riferimento ad un RS, esso lo conferma; pertanto come prima cosa il thread span blocca la ritrasmissione di quell'RS (se non già fatto a seguito della ricezione di un RAS). Procede poi al controllo dei dati ricevuti; se è stato ricevuto l'intero blocco il thread span lo consegna al BP liberando il buffer di ricezione della sessione.
- CS: il thread genera il CAS come conferma di ricezione e si procede alla cancellazione della sessione di ricezione corrispondente.
- CAR: questo segmento viene ignorato in quanto si è deciso di non attivare la ritrasmissione dei CR, similmente a quanto viene fatto in ION.

Sessione di trasmissione:

- RS: si genera immediatamente il RAS di risposta. Il RS conferma un CP, pertanto si procede a bloccare la ritrasmissione di quel CP. Si procede quindi all'invio degli eventuali segmenti corrispondenti ai dati non confermati; se tutti i dati inviati sono stati confermati (RS finale) si procede alla segnalazione al BP del successo della trasmissione dei bundle contenuti nel blocco appena confermato, infine si cancella la sessione di trasmissione.
- CR: si genera il CAR come conferma di ricezione e si procede alla cancellazione della sessione, segnalando al BP che la trasmissione dei bundle contenuti nel blocco ha avuto esito negativo.
- CAS: si blocca la ritrasmissione del CS generato.

L'altra operazione svolta dallo span thread è la creazione di nuove sessioni di invio: per far ciò recupera i bundle da trasmettere prelevandoli dal BP, li inserisce in un blocco (ogni

blocco contiene un bundle), frammenta il blocco in segmenti ed invia i segmenti, contrassegnando l'ultimo come CP.

Si noti che la gestione dei segmenti di segnalazione è stata resa prioritaria rispetto alla creazione di nuove sessioni di invio: questo consente di evitare di incappare in problematiche causate da una eccessiva creazione di sessioni di esportazione, col rischio di non riuscire a concludere quelle già in corso.

3.6.2.2 Il Congestion control thread

Il congestion control thread aggiunge dei token ad un "token bucket" utilizzato per limitare la velocità di trasmissione dei segmenti LTP alla velocità corrispondente alla velocità nominale indicata dal contatto in corso verso il destinatario. Si noti che la velocità indicata nel contatto è relativa al livello BP, mentre quella implementata dal token bucket è relativa all'LTP, per cui in linea di principio devono essere diverse (quella dell'LTP deve essere maggiore per tenere conto dell'overhead LTP). La dimensione del bucket, ovvero il numero massimo di token che il token bucket è capace di memorizzare, è configurabile per ogni span: questo permette di configurare la dimensione massima del burst di segmenti inviato dopo un periodo di pausa dello span.

3.6.2.3 Il thread per la segnalazione di nuovi dati da inviare

Questo thread controlla se sono disponibili nuovi bundle da trasmettere; in caso affermativo lo segnala allo Span thread. Questo disaccoppiamento consente di rendere bloccante questo thread e permette allo Span thread di rimanere sbloccato, così che non risulti compromessa la gestione dei segmenti di segnalazione.

3.7 Esempio di interazione tra il Receiver e lo Span thread

Di seguito si mostra l'interazione tra i diversi thread appena discussi, ponendoci in due possibili scenari:

1. ricezione di un blocco
2. trasmissione di un blocco

Il primo viene mostrato nella successiva Figura 6. Il Receiver thread all'arrivo del primo segmento crea la sessione di ricezione ed inserisce quel segmento, insieme ad ogni altro

futuro segmento dati della medesima sessione, nel buffer di ricezione della sessione. Quando riceve un segmento dati contrassegnato come CP, questo viene inserito anche nella coda dei segmenti di segnalazione ricevuti da quel nodo e lo Span thread corrispondente lo processa: dato che nell'esempio vengono ricevuti tutti i segmenti consegna i dati al protocollo superiore, genera un RS in risposta che indica che tutto è stato ricevuto ed infine elimina la sessione dall'elenco delle sessioni attive e la inserisce nell'elenco delle sessioni terminate di recente.

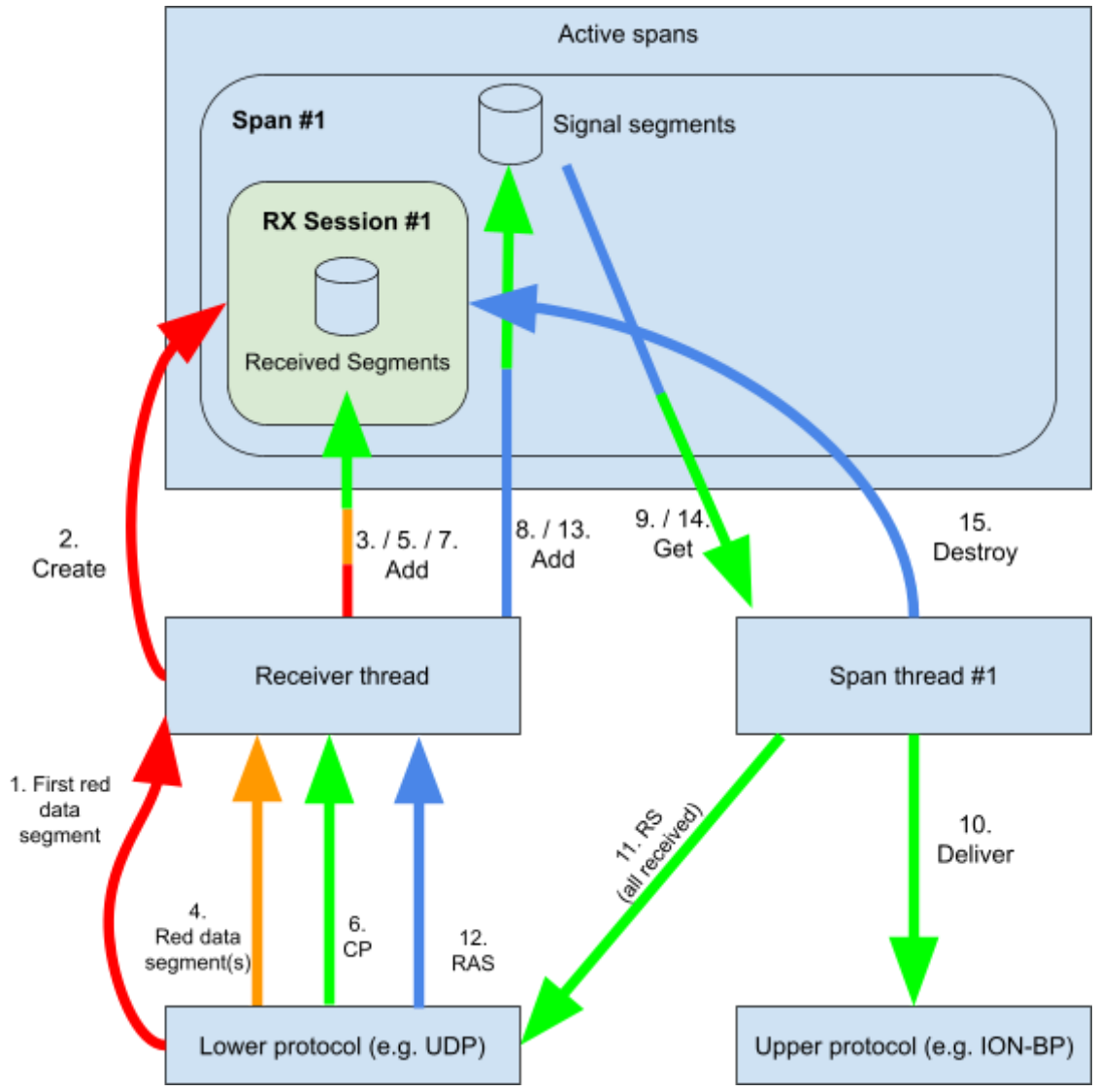

Figura 6. Interazione tra Receiver thread e Span thread in una sessione “red” in assenza di perdite. La sequenza temporale è indicata dai numeri in ordine crescente; i colori indicano in quali operazioni è coinvolto ogni segmento.

Il secondo scenario riguarda la trasmissione di un blocco, mostrata in Figura 7. Quando il BP segnala che nuovi dati sono disponibili, il thread adibito a ricevere queste segnalazioni (si veda la sezione 3.5.2.3) si attiva e segnala allo Span thread che un nuovo dato da trasmettere è disponibile: lo Span thread recupera i dati da trasmettere, crea la sessione di invio, divide il blocco in segmenti e li invia, contrassegnando l’ultimo come CP, EORP e EOB. Il Receiver thread quindi riceve il RS, lo inserisce nella coda dei segmenti di segnalazione ricevuti letta dallo Span thread (il quale genera il RAS), segnala il successo

della trasmissione dei bundle contenuti nel blocco al BP ed infine rimuove la sessione dalle sessioni attive e la inserisce nelle sessioni terminate di recente.

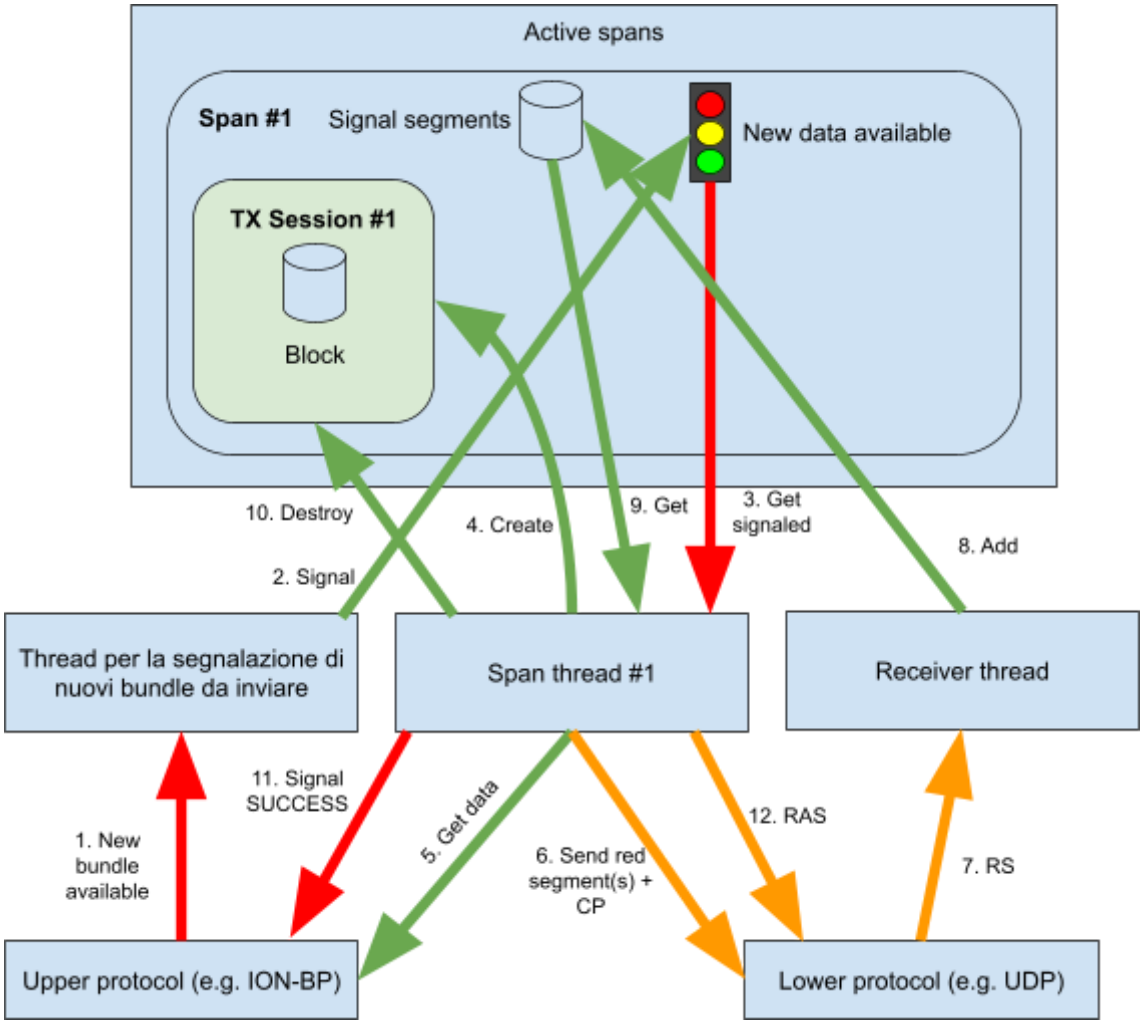

Figura 7. Interazione tra Receiver thread ed i thread che gestiscono uno span nel caso di trasmissione di nuovi dati provenienti dal protocollo superiore (BP) tramite sessione “red” in caso di assenza di perdite. La sequenza temporale è indicata dai numeri in ordine crescente; i colori indicano la tipologia di operazione svolta: in rosso gli eventi, in verde le funzioni ed in arancione l’invio o la ricezione di segmenti.

4 Dettagli implementativi

In questo capitolo viene proposta una spiegazione più tecnica delle scelte di progetto fatte durante la realizzazione di Unibo-LTP.

4.1 Libreria generica per le Linked List

Nella realizzazione di LTP era importante poter disporre di una libreria per la gestione delle liste puntate: ho quindi deciso di riutilizzare, modificandola quando necessario come successivamente illustrato, la libreria da me realizzata per il lavoro di tirocinio svolto durante la laurea triennale presso il centro di ricerca Arces dell'Università di Bologna, rilasciata come software libero e disponibile su Gitlab [16].

Tale libreria consente la manipolazione di liste puntate generiche, cioè liste contenenti qualunque tipo di dato, grazie all'utilizzo dei puntatori generici in C (*void**); un nodo della lista è così implementato:

```
typedef struct Node
{
      void* data;
      size_t data_size;

      struct Node* next;
} Node;
```

Questa caratteristica è risultata particolarmente vantaggiosa in questo progetto in quanto ha permesso l'utilizzo di un unico strumento per la gestione di liste di tipologie diverse.

Alla versione originale ho aggiunto diverse caratteristiche legate alla programmazione funzionale, implementate tramite il passaggio di puntatori a funzione fra gli argomenti delle funzioni così da "iniettare" comportamenti; peraltro questi concetti erano già in parte utilizzati, ad esempio era presente il metodo

```
int list_find (
          List list,
          void* data_to_search,
          size_t data_to_search_size,
          int (*compare)(void*, size_t, void*, size_t)    );
```

che permetteva di ricercare all'interno della lista il primo elemento uguale a quello passato come argomento (*data_to_search*), dove l'uguaglianza era dettata dalla funzione *compare* passata come argomento (in accordo con la normale convenzione dove "0" indica uguaglianza).

Le funzioni introdotte nella nuova versione sono tre.

La prima riguarda la possibilità di eseguire una funzione su ogni elemento della lista; a questo scopo si è realizzata la funzione

```
void list_for_each(
          List list,
          void (*function)(void*, size_t)            );
```

che è stata utilizzata in vari contesti, ad esempio per l'eliminazione delle sessioni attive oppure per "congelare" tutti i timer relativi ad una sessione alla scadenza di un contatto (nell'ipotesi in cui non ce ne sia uno subito continuo).

La seconda funzione introdotta è

```
bool list_remove_if(
          List* list,
          bool (*isToRemove)(void*, size_t)          );
```

che consente di rimuovere dalla lista tutti gli elementi in cui la funzione *isToRemove* restituisce vero (*true*); è stata utilizzata per la rimozione di contatti e range scaduti.

L'ultima funzione implementata è

```
void* list_push_ordered(
          List *list,
          void* data,
          size_t data_size,
          int (*compare)(void*, size_t, void*, size_t)  );
```

che dà la possibilità di inserire elementi in maniera ordinata; l'ordine è stabilito dalla funzione *compare* passata come argomento, che confronta l'elemento da inserire con gli elementi della lista e trova quindi il suo corretto posizionamento. Si noti che l'ordine

crescente o decrescente della lista è deciso dalla funzione compare, pertanto sta all'utilizzatore della libreria decidere in che ordine vuole inserire gli elementi. Questa funzione è stata utile per la memorizzazione di contatti e "range" ma, soprattutto, ha consentito di velocizzare il thread ricevitore: la lista dei segmenti ricevuti viene mantenuta in ordine decrescente di *offset*, ciò ha permesso di velocizzare notevolmente l'inserimento dei nuovi segmenti (solitamente vengono aggiunti in testa) ed ha velocizzato anche il controllo della completa ricezione di un blocco.

4.2 Ereditarietà e tipi generici in C

I linguaggi di programmazione ad oggetti si basano sul principio di ereditarietà ed offrono la possibilità di utilizzare tipi generici, il che consente di creare un modello realistico che porta ad una semplificazione e ad un codice più comprensibile. È noto che il C non prevede costrutti nativi basati su queste assunzioni, tuttavia si può far uso di strutture dati (*struct*) ed unioni (*union*) per simulare questo comportamento. Nel progetto in esame ho utilizzato questa tecnica sia per la rappresentazione dei segmenti, sia per le sessioni (di invio e di ricezione); esaminiamo come esempio il seguente caso, in cui i segmenti sono caratterizzati da un campo *header* comune ed uno o più campi specifici diversi a seconda della tipologia di segmento, così rappresentati:

```
typedef struct {
     LTPSegmentHeader    header;

     union {
          LTPDataSegment           dataSegment;
          LTPReportSegment         reportSegment;
          LTPReportAckSegment      reportAckSegment;
          LTPCancelSegment         cancelSegment;
      };
} LTPSegment;
```

Così facendo si fornisce un concetto generico di segmento; l'utilizzo della union consente di risparmiare spazio di memoria in quanto impone un vincolo di mutua esclusione tra i suoi campi.

4.3 Polimorfismo in C

Il concetto di polimorfismo è un concetto chiave della programmazione ad oggetti, tuttavia il C non prevede l'utilizzo di interfacce, classi ed ereditarietà, pertanto non prevede neanche il polimorfismo. La necessità di poter utilizzare contemporaneamente più adapter per i protocolli inferiori porterebbe all'utilizzo di questi costrutti, pertanto ho scelto di simulare il loro funzionamento tramite strutture dati (*struct*) e puntatori a funzione creando la seguente struttura

```
typedef struct {
   const char*  protocolName;
   bool         (*init) (char* initString);
   bool         (*initOut) (unsigned int ipAddress,
                            unsigned short portNumber,
                            char* initString);
   void         (*dispose) ();
   bool         (*receive) (char* buffer, int *bufferLength);
   bool         (*send) (  char* buffer,
                            int bufferLength,
                            unsigned int ipAddress,
                            unsigned short portNumber);

   int          (*setFDForSelect) (fd_set* fdstruct);
   bool         (*receivedFromFDForSelect) (fd_set* fdstruct);
} LowerProtocolAdapter;
```

che simula il concetto di interfaccia, facendo sì che un'implementazione di un adapter corrisponda all'implementazione delle funzioni e dalla loro associazione all'interno della struttura dati. La stringa *protocolName* permette di distinguere "l'istanza" tramite il nome del protocollo mentre i puntatori a funzione fanno sì che si possa avere un comportamento simile al polimorfismo. Nel caso specifico dei protocolli inferiori, la funzione di inizializzazione dell'intero modulo prepara le strutture dati dei singoli adapter associando i nomi con i puntatori a funzione ed inserisce il tutto nella lista degli adapter inizializzati e pronti all'uso.

4.4 Implementazione dei timer

I timer sono uno strumento fondamentale per Unibo-LTP; il loro principale utilizzo riguarda la gestione delle ritrasmissioni ma vengono usati anche per aggiornare il contact plan, per l'eliminazione di contatti e range scaduti e per l'eliminazione delle sessioni terminate di recente. Tutte queste operazioni non richiedono una precisione estrema, pertanto si è deciso di implementarle accettando piccole imprecisioni nel tempo di attesa; quelle che si possono fare su un timer sono cinque:

1. creazione
2. pausa
3. ripresa
4. interruzione
5. riavvio

Sottolineo che le operazioni di pausa e ripresa sono necessarie in quanto, in caso di assenza di contatto o di range, tutti i timer di ritrasmissione devono essere messi in pausa e devono riprendere quando saranno attivi due range (uno per ogni lato del canale) ed un contatto.

La prima operazione è la creazione di un timer ed avviene mediante la funzione

```
TimerID startTimer (
     unsigned int   millisToSleep,
     TimerHandler   handler,
     void*          data              );
```

Questa crea un thread che rimane in attesa (*sleep*) per il tempo richiesto (*millisToSleep*), al termine del quale viene chiamata la funzione *handler* passando come argomento il puntatore *data*. La funzione restituisce un identificatore del timer, che è stato chiamato TimerID; grazie a quest'ultimo è possibile eseguire le altre quattro operazioni tramite le seguenti funzioni:

```
void       pauseTimer      (TimerID timerID);
void       resumeTimer     (TimerID timerID);
bool       stopTimer       (TimerID timerID);
bool       restartTimer    (TimerID timerID);
```

Delle cinque funzioni quelle di pausa e di ripresa del timer sono le più critiche tanto che hanno richiesto di suddividere l'attesa in più periodi, pertanto si fanno più attese anziché una unica; al termine di ognuna, solamente nel caso lo stato risulti essere attivo (non in pausa) il valore complessivo da attendere, inizialmente posto uguale a millisToSleep, viene decrementato di un valore pari al tempo atteso; questa operazione viene eseguita finché il valore non arriva a 0, oppure finchè il timer non viene interrotto tramite funzione `stopTimer(TimerID timerID)`. Il codice implementato è il seguente:

```
do {
      const unsigned int toSleep = min(TIMER_CHECK_STATE,
timer->millisToSleep);
      usleep(1000 * toSleep);

      if ( stop ) {    // Stopped timer -> Exit thread
            pthread_mutex_lock(&lockActiveTimers);
            list_remove_data(&activeTimers, &(timer->timerID),
sizeof(TimerID), _findTimerID);
            pthread_mutex_unlock(&lockActiveTimers);
            _free(timer);
            return NULL;
      }

      if ( !pause ) {
            timer->millisToSleep -= toSleep;
      }

      if ( restart ) {
            timer->millisToSleep = timer->startMillisToSleep;
      }
} while ( timer->millisToSleep > 0 );
```

Come si evince dal codice, l'interruzione (stop) del timer causa la terminazione del thread alla prima occasione utile.

4.5 Span thread in attesa bloccante su due semafori

Come descritto nella sezione 3.6.2.1, lo span thread deve attendere sia l'arrivo di nuovi segmenti di segnalazione da processare, passati dal thread di ricezione, sia la presenza di nuovi bundle da trasmettere. L'attesa di un unico evento si sarebbe potuta gestire

facilmente con dei semafori di Dijkstra, ma nel caso in esame il problema riguardava l'attesa bloccante su due diversi tipi di evento: la soluzione adottata sta nell'utilizzo della *select*, funzione pensata proprio per questo scopo ma che richiede dei *file descriptor* e non dei semafori. Dopo diversi tentativi preliminari con *pipe* e *socket* ho trovato un costrutto migliore, l'*eventfd*, che consente di creare un file descriptor da utilizzare per inviare e ricevere notifiche, aumentando il livello di dettaglio. Questo nasconde un numero intero senza segno a 64 bit gestito direttamente dal kernel in maniera molto simile ai semafori; le operazioni di scrittura (*write*) permettono di incrementare il contatore di un valore pari al valore scritto sull'eventfd, mentre le letture permettono di estrarre e resettare il valore del contatore: nel caso in cui il contatore sia inizialmente 0 questo non viene restituito ma si crea un'attesa bloccante. La particolarità dell'eventfd è che il file descriptor creato supporta le funzioni "poll" e "select", pertanto risulta adatto per rimanere contemporaneamente in attesa bloccante su più di un evento. Riporto di seguito l'utilizzo che si è fatto all'interno dello Span thread, ricordando che si è ritenuta prioritaria la gestione dei segmenti di segnalazione rispetto alla creazione di nuove sessioni di invio.

```
fd_set fdset = fdstruct;
select(max+1, &fdset, NULL, NULL, &timeout);
if ( FD_ISSET(span->signalSegmentsEventFD, &fdset) ) {
      read(span->signalSegmentsEventFD,
            &numberOfRead,
            sizeof(numberOfRead));
      for (int i = 1; i <= numberOfRead; i++) {
            _handleSignalSegment(span);
      }
}
else if(FD_ISSET(span->dataFromUpperProtocolEventFD, &fdset)) {
// dataFromUpperProtocol available. NOTE: this will be executed
only when no signal segments are available!
      read(span->dataFromUpperProtocolEventFD,
            &numberOfRead,
            sizeof(numberOfRead));

      for (int i = 1; i <= numberOfRead; i++) {
            _handleSendNewBlock(span);
      }
}
```

4.6 Configurazione

Unibo-LTP ottiene dall'interfaccia verso ION il bundle da trasmettere e le informazioni del contact plan; richiede inoltre la definizione di alcuni parametri di configurazione, in parte impostabili nel file "config.h" ed in parte in un file di configurazione di formato JSON.

4.6.1 Config.h

I parametri impostabili tramite istruzioni *define* all'interno del file "config.h" sono:

- MAX_RETX_NUMBER: indica il numero massimo di ritrasmissioni effettuabili prima che la sessione (di invio o ricezione) venga cancellata tramite l'opportuno cancel segment.
- SIGNAL_SEGMENT_COPIES: indica il numero di copie del medesimo segmento di segnalazione da trasmettere, per aumentare la probabilità che almeno una giunga a destinazione, come spiegato in [13] ed anche implementato in ION.
- DISTRIBUTED_COPIES: indica se le copie devono essere distribuite temporalmente all'interno di un RTO. Se vero si ha una trasmissione delle copie di tipo "distributed" anziché "burst" [13].
- RECENTLY_CLOSED_LIST_UPDATE_TIME: indica ogni quanti secondi avviene il controllo per rimuovere una sessione dall'elenco delle sessioni terminate di recente.
- TOKEN_UPDATE_TIME: indica ogni quanti millisecondi vengono inseriti nuovi token all'interno del token bucket che regola la velocità di trasmissione dei segmenti LTP.
- TIMER_CHECK_STATE: indica quanti millisecondi un timer deve attendere prima di controllare se è stato interrotto, messo in pausa o riattivato.
- CONTACT_PLAN_UPDATE_TIME: indica ogni quanti secondi viene aggiornato il contact plan interno a LTP (sincronizzandolo con quello di ION).
- MAX_NUMBER_OF_CLAIMS_IN_REPORT_SEGMENT: indica il numero massimo di claim che si possono inserire in un Report Segment (inviato da Unibo-LTP); se il numero di claim da inviare risulta superiore verranno generati più RS.
- CONFIG_FILE_NAME: indica il nome del file di configurazione (formato JSON).

- LISTEN_UDP_DEFAULT_PORT: indica la porta UDP in cui LTP si mette in ascolto se questa non viene indicata nel file di configurazione (si veda la sezione 4.6.2).
- SESSION_INTERSEGMENT_MAX_TIME: indica il tempo massimo che può intercorrere tra la ricezione di un segmento "green" o "orange" ed il successivo; oltre questo tempo la sessione di ricezione viene annullata.

La configurazione che viene fornita di default è la seguente:

```
#define MAX_RETX_NUMBER (10)

#define SIGNAL_SEGMENT_COPIES (1)

#define DISTRIBUTED_COPIES (0)

#define RECENTLY_CLOSED_LIST_UPDATE_TIME (1)

#define TOKEN_UPDATE_TIME (50)

#define TIMER_CHECK_STATE (500)

#define CONTACT_PLAN_UPDATE_TIME (60)

#define MAX_NUMBER_OF_CLAIMS_IN_REPORT_SEGMENT (20)

#define CONFIG_FILE_NAME "spans.json"

#define LISTEN_UDP_DEFAULT_PORT (1113)

#define SESSION_INTERSEGMENT_MAX_TIME (3)
```

Com'è ovvio, il cambio di una define richiede la ricompilazione del programma, quindi questo sistema non è adatto per impostare configurazioni che devono essere cambiate dall'utente o che si presume debbano comunque essere modificate spesso: per queste si è fatto ricorso ad un file esterno.

4.6.2 Il file di configurazione (ad esempio “spans.json”)

L’ultima configurazione riguarda il file JSON e consente di inserire gli span; il *parsing* del file viene effettuato mediante libreria json-c [17]. Il file fornito di default prevede due span ed è il seguente:

```
{
  "spans" :
    [
        {
            "peerEngineNumber" : 103,
            "maxExportSessions" : 50,
            "maxImportSessions" : 50,
            "maxSegmentPayloadSize" : 1024,
            "destinationIP" : "localhost",
            "destinationPort" : 1113,
            "remoteProcessingDelay" : 1,
            "TB_burstSize" : 10485760,
            "color" : "red",
            "outLowerProtocol" : "ECLSA",
            "outLowerProtocolString" : "127.0.0.1:11113 576 512
500 1 0 10000000"
        },

        {
            "peerEngineNumber" : 200,
            "maxExportSessions" : 20,
            "maxImportSessions" : 20,
            "maxSegmentPayloadSize" : 1024,
            "destinationIP" : "galaxeon",
            "destinationPort" : 1113,
            "remoteProcessingDelay" : 1,
            "TB_burstSize" : 10485760,
            "color" : "red",
            "outLowerProtocol" : "UDP",
            "outLowerProtocolString" : ""
        }
    ],

  "inLowerProtocols" :
    [
```

```
      {
            "inLowerProtocol" : "UDP",
            "inLowerProtocolString" : "0.0.0.0:1113"
      },
       {
            "inLowerProtocol" : "ECLSA",
            "inLowerProtocolString" : "0.0.0.0:11113 100 1 1026
2048"
      }
    ],
  "ownqtime" : 1
}
```

Come si evince dall'esempio, per ogni span bisogna indicare:

- peerEngineNumber: il numero del nodo dell'engine verso cui è diretto lo span;
- maxExportSessions: il numero massimo di sessioni di invio;
- maxImportSessions: il numero massimo di sessioni di ricezione;
- maxSegmentPayloadSize: la dimensione massima del payload nei segmenti dati;
- destinationIP: l'hostname (o l'IP) dell'engine LTP remoto a cui vengono inviati i segmenti LTP relativi allo span;
- destinationPort: il numero della porta dell'engine LTP remoto a cui vengono inviati i segmenti LTP relativi allo span;
- remoteProcessingDelay: la stima del tempo tempo di processamento dell'engine remoto, usato per il calcolo dell'RTO;
- TB_burstSize: la dimensione massima (in byte) del token bucket utilizzato per controllare l'invio (traffic shaping) dei segmenti LTP; coincide con la lunghezza massima di un burst, dopo un periodo di pausa.
- color: indica il colore da utilizzare, la scelta deve essere tra: "green", "orange" e "red". Ogni sessione di trasmissione verso lo span verrà generata utilizzando il colore indicato.
- outLowerProtocol: indica quale protocollo deve essere utilizzato nel protocollo sottostante.
- outLowerProtocolString: permette di personalizzare il funzionamento del protocollo inferiore scelto nel punto precedente.

Bisogna inoltre inserire quali protocolli si vogliono utilizzare per la ricezione di segmenti; ogni protocollo prevede una propria stringa specifica di inizializzazione che consente di personalizzare il funzionamento (ad esempio indicando indirizzo IP e porta di ascolto).

L'ultimo parametro prende spunto da ION: è stato inserito il campo *ownqtime* che permette di configurare il tempo nominale di elaborazione processamento dei segmenti nel nodo locale; è opportuno ricordare che l'RTO è dato dalla somma dei due range (unidirezionali), dall'ownqtime e dal remoteProcessingDelay.

4.7 Compilazione e avvio

Unibo-LTP viene compilato come applicazione a sé stante tramite il comando *make*, a cui deve essere passato il parametro obbligatorio "ION_DIR" indicante il percorso alla cartella dei sorgenti di ION. Il comando "make help" (o semplicemente "make" senza parametro obbligatorio) produce alcune righe di aiuto:

```
Usage: make ION_DIR=<ion_dir>
To compile with debug symbols add DEBUG=1
To change the BP version use BP=bpv7, BP=bpv6 is default. For
old ION versions use BP=bp
```

Preciso che LTP è compatibile sia con la versione 6 che con la versione 7 del BP; per compilare il programma è richiesta inoltre che sia installata la libreria "json-c", utilizzata nel parsing del file di configurazione. L'operazione di "make" produce un file eseguibile "UniboLTP" nella cartella corrente, installabile come di norma nelle cartelle di sistema tramite il comando "make install" in qualità di utente *root*.

Unibo-LTP è un processo autonomo, ma completamente integrato in ION tramite la sua interfaccia. Per comodità dell'utente ho fatto in modo che sia possibile lanciarlo direttamente dal file di configurazione di ION: a questo scopo è necessario modificare la configurazione delle istruzioni *induct* e *outduct* del file di configurazione principale di ION, o ".rc". Sotto è riportato come esempio l'estratto di un file di configurazione di un nodo DTN, che prevede l'utilizzo dell'LTP come Convergence Layer per le comunicazioni con sé stesso (nodo 100) e con un vicino (nodo 200):

```
a induct  ltp 100 ltpcli
```

```
a outduct ltp 100 ltpclo
a outduct ltp 200 ltpclo
```

La prima riga consente di avviare il ricevitore dell'LTP (*induct*) sul nodo locale, le successive configurano gli *outduct*, ovvero i link in uscita ed è necessario un outduct per ogni nodo destinatario, incluso il nodo medesimo. Questa configurazione deve essere modificata per l'utilizzo di Unibo-LTP nel modo seguente:

```
a induct  ltp 100 'UniboLTP'
a outduct ltp 100 ''
a outduct ltp 200 ''
```

All'interno delle virgolette dell'induct (dopo UniboLTP) è possibile inserire il parametro "--log", che consente di attivare la stampa delle informazioni dei log.

4.8 Log

Quando attivati, i log stampano a video ogni operazione eseguita, ad eccezione della ricezione e dell'invio di segmenti dati (genererebbero troppe righe uguali). Osservando in modo sinottico il terminale del mittente e quello del ricevitore è possibile studiare in tempo reale, o al termine dell'esperimento, il funzionamento di LTP. Nella realizzazione delle scritte si è tenuta in massima considerazione la leggibilità delle medesime, in modo che i log possano essere utilizzati non solo a fine di debug ma anche a scopo didattico, per mostrare in tempo reale il funzionamento del protocollo LTP; vediamo ora due esempi.

4.8.1 Primo esempio: singola sessione di invio

Invio di un blocco dati da 10 MB dal nodo 100 in presenza di perdite; l'unica sessione è la 1.

Sender (engine 100)
Session 100-1: Created (red TX) Session 100-1: Data segment(s) sent Session 100-1: CP 1 sent, waiting for RS Session 100-1: RS received 29 in response to CP 1 claimCount = 9 bounds 0-10000049

```
Session 100-1:    Sent RAS to confirm RS 29
Session 100-1:            Gap in RS between 730113 and 731136
Session 100-1:            Gap in RS between 974849 and 975872
Session 100-1:            Gap in RS between 1416193 and 1417216
Session 100-1:            Gap in RS between 1454081 and 1455104
Session 100-1:            Gap in RS between 2835457 and 2836480
Session 100-1:            Gap in RS between 4643841 and 4644864
Session 100-1:            Gap in RS between 5768193 and 5769216
Session 100-1:            Gap in RS between 7418881 and 7419904
Session 100-1:    Data segment(s) sent
Session 100-1:    CP 2 sent in response to RS 29. Waiting for RS
Session 100-1:    RS received 30 in response to CP 2
claimCount = 1          bounds 0-10000049
Session 100-1:    Sent RAS to confirm RS 30
Session 100-1:    Block acked, signaling SUCCESS to upper
protocol
Session 100-1:    TX buffer freed
Session 100-1:    Inserted in terminated sessions
Session 100-1:    Closed
Session 100-1:    Removing the session from the terminated
session list
```

Receiver

```
Session 100-1:    Created (red RX)
Session 100-1:    CP 1 received
Session 100-1:    Building RS 29 in response to CP 1      Lower
Bound = 0
Session 100-1:            Claim # 1.      Offset = 0      Length
= 730112 Absolute ranges: 1 - 730112
Session 100-1:            Claim # 2.      Offset = 731136 Length
= 243712 Absolute ranges: 731137 - 974848
Session 100-1:            Claim # 3.      Offset = 975872 Length
= 440320 Absolute ranges: 975873 - 1416192
Session 100-1:            Claim # 4.      Offset = 1417216
Length = 36864  Absolute ranges: 1417217 - 1454080
Session 100-1:            Claim # 5.      Offset = 1455104
Length = 1380352        Absolute ranges: 1455105 - 2835456
Session 100-1:            Claim # 6.      Offset = 2836480
Length = 1807360        Absolute ranges: 2836481 - 4643840
Session 100-1:            Claim # 7.      Offset = 4644864
```

```
Length = 1123328        Absolute ranges: 4644865 - 5768192
Session 100-1:            Claim # 8.      Offset = 5769216
Length = 1649664        Absolute ranges: 5769217 - 7418880
Session 100-1:            Claim # 9.      Offset = 7419904
Length = 2580145        Absolute ranges: 7419905 - 10000049
Session 100-1:    RS 29 sent, waiting for RAS
Session 100-1:    RAS received confirming RS 29
Session 100-1:    CP 2 received in response to RS 29
Session 100-1:    Block completed; to be delivered to upper
protocol
Session 100-1:  Building RS 30 in response to CP 2      Lower
Bound = 0
Session 100-1:            Claim # 1.      Offset = 0      Length
= 10000049       Absolute ranges: 1 - 10000049
Session 100-1:    RS 30 sent, waiting for RAS
Session 100-1:    RAS received confirming RS 30
Session 100-1:    Completed
Session 100-1:    RX buffer freed
Session 100-1:    Inserted in terminated sessions
Session 100-1:    Closed
Session 100-1:    Removing the session from the terminated
session list
```

4.8.2 Secondo esempio: due sessioni di invio contemporanee

Trasmissione contemporanea da uno stesso nodo (100) di due blocchi in presenza di perdite su entrambi; nelle sessioni di invio (numero 2 e 3) vengono trasmessi blocchi dati da 10 MB.

Sender (engine 100)

```
Session 100-2:    Created (red TX)
Session 100-2:    Data segment(s) sent
Session 100-2:    CP 3 sent, waiting for RS
Session 100-3:    Created (red TX)
Session 100-3:    Data segment(s) sent
Session 100-3:    CP 4 sent, waiting for RS
Session 100-2:    RS received 31 in response to CP 3
```

```
claimCount = 16          bounds 0-10000049
Session 100-2:    Sent RAS to confirm RS 31
Session 100-2:             Gap in RS between 815105 and 816128
Session 100-2:             Gap in RS between 1429505 and 1430528
Session 100-2:             Gap in RS between 2027521 and 2028544
Session 100-2:             Gap in RS between 2789377 and 2790400
Session 100-2:             Gap in RS between 3399681 and 3400704
Session 100-2:             Gap in RS between 4062209 and 4063232
Session 100-2:             Gap in RS between 4358145 and 4359168
Session 100-2:             Gap in RS between 5057537 and 5058560
Session 100-2:             Gap in RS between 5594113 and 5671936
Session 100-2:             Gap in RS between 6384641 and 6396928
Session 100-2:             Gap in RS between 6397953 and 6398976
Session 100-2:             Gap in RS between 7495681 and 7496704
Session 100-2:             Gap in RS between 8584193 and 8585216
Session 100-2:             Gap in RS between 9207809 and 9208832
Session 100-2:             Gap in RS between 9842689 and 9843712
Session 100-2:    Data segment(s) sent
Session 100-2:    CP 5 sent in response to RS 31. Waiting for RS
Session 100-3:    RS received 32 in response to CP 4
claimCount = 7           bounds 0-10000049
Session 100-3:    Sent RAS to confirm RS 32
Session 100-3:             Gap in RS between 615425 and 616448
Session 100-3:             Gap in RS between 1073153 and 1074176
Session 100-3:             Gap in RS between 3253249 and 3254272
Session 100-3:             Gap in RS between 3856385 and 3857408
Session 100-3:             Gap in RS between 4260865 and 4261888
Session 100-3:             Gap in RS between 5773313 and 5774336
Session 100-3:    Data segment(s) sent
Session 100-3:    CP 6 sent in response to RS 32. Waiting for RS
Session 100-2:    RS received 33 in response to CP 5
claimCount = 1           bounds 0-10000049
Session 100-2:    Sent RAS to confirm RS 33
Session 100-2:    Block acked, signaling SUCCESS to upper
protocol
Session 100-2:    TX buffer freed
Session 100-2:    Inserted in terminated sessions
Session 100-2:    Closed
Session 100-3:    RS received 34 in response to CP 6
claimCount = 1           bounds 0-10000049
```

```
Session 100-3:    Sent RAS to confirm RS 34
Session 100-3:    Block acked, signaling SUCCESS to upper
protocol
Session 100-3:    TX buffer freed
Session 100-3:    Inserted in terminated sessions
Session 100-3:    Closed
Session 100-2:    Removing the session from the terminated
session list
Session 100-3:    Removing the session from the terminated
session list
```

Receiver

```
Session 100-2:    Created (red RX)
Session 100-2:    CP 3 received
Session 100-2:    Building RS 31 in response to CP 3      Lower
Bound = 0
Session 100-2:            Claim # 1.      Offset = 0      Length
= 815104 Absolute ranges: 1 - 815104
Session 100-2:            Claim # 2.      Offset = 816128 Length
= 613376 Absolute ranges: 816129 - 1429504
Session 100-2:            Claim # 3.      Offset = 1430528
Length = 596992 Absolute ranges: 1430529 - 2027520
Session 100-2:            Claim # 4.      Offset = 2028544
Length = 760832 Absolute ranges: 2028545 - 2789376
Session 100-2:            Claim # 5.      Offset = 2790400
Length = 609280 Absolute ranges: 2790401 - 3399680
Session 100-2:            Claim # 6.      Offset = 3400704
Length = 661504 Absolute ranges: 3400705 - 4062208
Session 100-2:            Claim # 7.      Offset = 4063232
Length = 294912 Absolute ranges: 4063233 - 4358144
Session 100-2:            Claim # 8.      Offset = 4359168
Length = 698368 Absolute ranges: 4359169 - 5057536
Session 100-2:            Claim # 9.      Offset = 5058560
Length = 535552 Absolute ranges: 5058561 - 5594112
Session 100-2:            Claim # 10.     Offset = 5671936
Length = 712704 Absolute ranges: 5671937 - 6384640
Session 100-2:            Claim # 11.     Offset = 6396928
Length = 1024   Absolute ranges: 6396929 - 6397952
Session 100-2:            Claim # 12.     Offset = 6398976
Length = 1096704        Absolute ranges: 6398977 - 7495680
```

```
Session 100-2:            Claim # 13.     Offset = 7496704
Length = 1087488        Absolute ranges: 7496705 - 8584192
Session 100-2:            Claim # 14.     Offset = 8585216
Length = 622592 Absolute ranges: 8585217 - 9207808
Session 100-2:            Claim # 15.     Offset = 9208832
Length = 633856 Absolute ranges: 9208833 - 9842688
Session 100-2:            Claim # 16.     Offset = 9843712
Length = 156337 Absolute ranges: 9843713 - 10000049
Session 100-2:    RS 31 sent, waiting for RAS
Session 100-3:    Created (red RX)
Session 100-3:    CP 4 received
Session 100-3:    Building RS 32 in response to CP 4      Lower
Bound = 0
Session 100-3:            Claim # 1.      Offset = 0      Length
= 615424 Absolute ranges: 1 - 615424
Session 100-3:            Claim # 2.      Offset = 616448 Length
= 456704 Absolute ranges: 616449 - 1073152
Session 100-3:            Claim # 3.      Offset = 1074176
Length = 2179072        Absolute ranges: 1074177 - 3253248
Session 100-3:            Claim # 4.      Offset = 3254272
Length = 602112 Absolute ranges: 3254273 - 3856384
Session 100-3:            Claim # 5.      Offset = 3857408
Length = 403456 Absolute ranges: 3857409 - 4260864
Session 100-3:            Claim # 6.      Offset = 4261888
Length = 1511424        Absolute ranges: 4261889 - 5773312
Session 100-3:            Claim # 7.      Offset = 5774336
Length = 4225713        Absolute ranges: 5774337 - 10000049
Session 100-3:    RS 32 sent, waiting for RAS
Session 100-2:    RAS received confirming RS 31
Session 100-2:    CP 5 received in response to RS 31
Session 100-2:    Block completed; to be delivered to upper
protocol
Session 100-2:  Building RS 33 in response to CP 5      Lower
Bound = 0
Session 100-2:            Claim # 1.      Offset = 0      Length
= 10000049       Absolute ranges: 1 - 10000049
Session 100-2:    RS 33 sent, waiting for RAS
Session 100-3:    RAS received confirming RS 32
Session 100-3:    CP 6 received in response to RS 32
Session 100-3:    Block completed; to be delivered to upper
```

```
protocol
Session 100-3:  Building RS 34 in response to CP 6      Lower
Bound = 0
Session 100-3:            Claim # 1.      Offset = 0      Length
= 10000049       Absolute ranges: 1 - 10000049
Session 100-3:    RS 34 sent, waiting for RAS
Session 100-2:    RAS received confirming RS 33
Session 100-2:    Completed
Session 100-2:    RX buffer freed
Session 100-2:    Inserted in terminated sessions
Session 100-2:    Closed
Session 100-3:    RAS received confirming RS 34
Session 100-3:    Completed
Session 100-3:    RX buffer freed
Session 100-3:    Inserted in terminated sessions
Session 100-3:    Closed
Session 100-3:    Removing the session from the terminated
session list
Session 100-2:    Removing the session from the terminated
session list
```

4.9 Note sul rilascio

Unibo-LTP è stato rilasciato come software libero sotto licenza GNU General Public License v2.0 ed è stato creato un repository Git sulla piattaforma GitLab, consultabile in [18].

Il codice è stato documentato facendo utilizzo del tool Doxygen [19]; si è fatto inoltre uso dei servizi di CI/CD (Continuous Integration / Continuous Delivery) disponibili in GitLab [20] per la creazione e pubblicazione automatica della documentazione prodotta, in modo che essa risulti sempre perfettamente allineata al codice.

5 Valutazione delle prestazioni

In questo capitolo vengono mostrati i risultati ottenuti con l'utilizzo di Unibo-LTP, facendo un raffronto con la versione dell'LTP di ION.

Per l'esecuzione dei test è stato usato DTNperf [21]: è il programma DTN di maggior successo sviluppato presso l'Università di Bologna, ispirato all'analogo programma IPerf e da quasi dieci anni inserito in ION (la suite DTN sviluppata e mantenuta da NASA-JPL), particolarmente utile per la valutazione delle prestazioni in reti DTN. In modalità client funziona come "generatore" di bundle destinati ad un'istanza di "DTNperf server"; il client può funzionare in due modalità:

- Rate based: genera bundle alla velocità indicata dall'utente, espressa in kbit/s oppure Mbit/s, oppure ancora in bundle/s. Questa modalità non richiede che i bundle vengano confermati dal server, pertanto il risultato ottenuto, in termini di byte trasmessi in media per unità di tempo, è un "throughput".
- Window based: questa modalità prevede che ogni bundle ricevuto venga confermato dal server con un apposito bundle; in questa modalità il client ne genera con continuità in modo da mantenere un certo quantitativo di bundle "in volo": ad esempio una window posta a 5 prevede l'invio di un burst di 5 bundle, seguito dall'invio di un nuovo bundle alla ricezione di ogni bundle di conferma. Poiché si ha una conferma dei dati ricevuti, il risultato in byte dati confermati (anziché trasmessi) in media per unità di tempo è un "goodput".

Mi permetto di aggiungere una nota personale: dato che negli ultimi due anni, dopo la tesi conclusiva del corso di laurea triennale, ho contribuito a sviluppare e mantenere le ultime versioni di DTNperf, per me è stato di particolare soddisfazione poterlo impiegare in veste di utente nel corso di questa tesi.

5.1 Mantenimento della velocità di trasmissione

Come già specificato, è stato utilizzato un token bucket come strumento per regolare la velocità di trasmissione; per dimostrarne la corretta implementazione ho effettuato un test, mostrato in Figura 8. Facendo uso di DTNperf in modalità rate è stato generato un singolo bundle da 30 MB, quindi volutamente "gigante", per mettere sotto stress Unibo-LTP. Il

TOKEN_UPDATE_TIME è stato posto a 50 ms, il TB_burstSize a 13 MiB (ovvero a 13.631.488 byte, poco più di un terzo del bundle inviato) con la dimensione massima del payload dei segmenti LTP impostata a 1024 byte. La velocità della scheda di rete era di 1 Gbit/s.

L'esperimento è stato eseguito due volte, prima con una velocità di trasmissione indicata nel contatto pari a 800 Mbit/s (in blu nella figura sotto), quindi ridotta di un ordine di grandezza a 80 Mbit/s (arancione nella figura). I tempi di ricezione dei segmenti sono stati raccolti tramite il programma *tcpdump*, filtrati con Wireshark ed infine elaborati e graficati con Excel. Nella figura ogni marker rappresenta l'invio di un segmento LTP, graficato in funzione del tempo di invio: i marker blu formano una retta continua, evidenziando il fatto che il token bucket non ha causato rallentamenti apprezzabili nell'invio nel caso di contatto a 800 Mbit/s; quelli di colore arancione, invece, si sovrappongono ai precedenti finchè non si consumano i token relativi ai 13 MiB della dimensione massima del burst, quindi si ottengono dei "mini burst" ogni 50 millisecondi (corrispondenti all'intervallo di aggiornamento dei token, impostabile dall'utente). In altre parole a 800 Mbit/s, velocità molto vicina al limite fisico della scheda di rete, il token bucket non limita in modo apprezzabile la velocità di invio (evidentemente limitata dall'hardware), mentre a 80 Mbit/s il traffico è sagomato, in modo da disperdere temporalmente i segmenti LTP del bundle, come voluto. Questo è naturalmente di fondamentale importanza per non mandare in overflow i buffer UDP dei ricevitori più lenti.

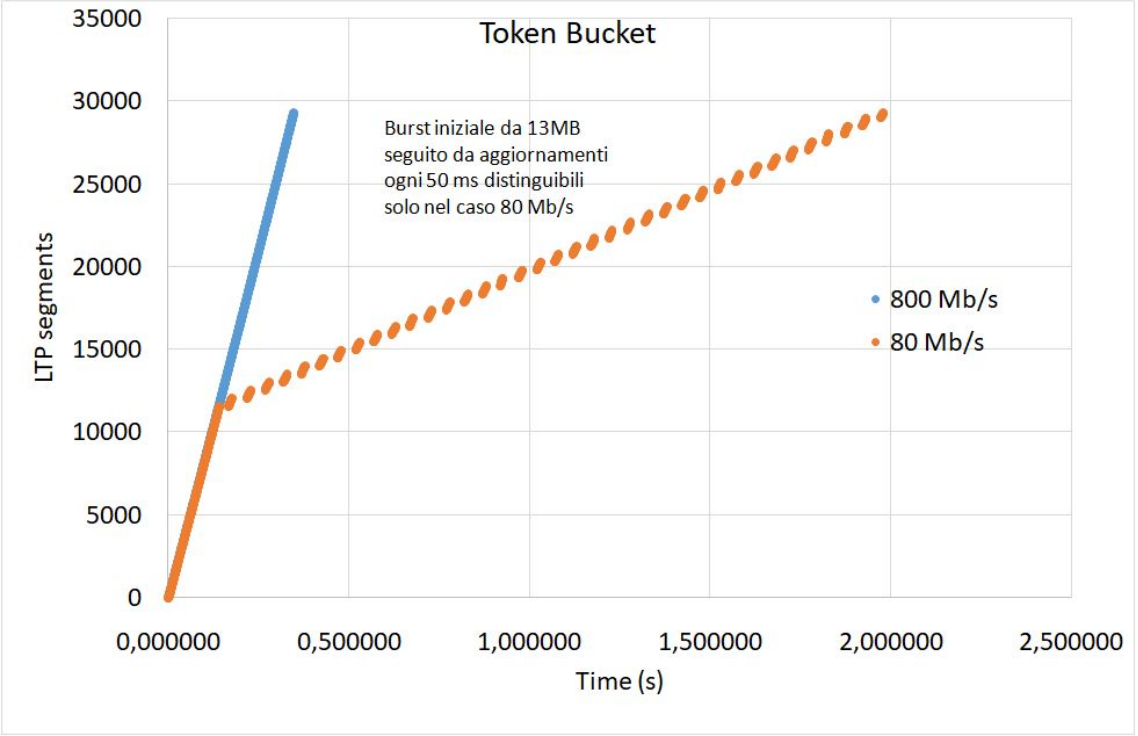

Figura 8. Test del corretto funzionamento del token bucket. I marker blu formano una retta continua, evidenziando la mancanza di interruzioni dovute all'assenza di token, prodotti in quantità corrispondente a 800 Mbit/s; i marker arancioni, sono sovrapposti ai precedenti per i primi 10 MiB (il burst impostato), quindi si presentano come una serie di piccoli burst ogni 50 ms (l'intervallo fra consegne successive di nuovi token), a causa della produzione di token limitata a soli 80 Mbit/s.

5.2 Confronto velocità Unibo-LTP e ION-LTP

Il test descritto in questa sezione si prefigge invece lo scopo di valutare quantitativamente l'incremento di velocità raggiungibile con Unibo-LTP rispetto alla versione standard presente in ION. Viene eseguito anch'esso mediante utilizzo di DTNperf, in modo che il client generi bundle da 10 MB di payload per 20 secondi con una dimensione della finestra pari a 20. Nell'impostazione degli span di entrambi i nodi sono stati inseriti 20 buffer di invio e 20 di ricezione (pari al numero massimo di bundle in volo, ovvero di blocchi LTP non completati) per non causare rallentamenti dovuti all'assenza di buffer di invio o ricezione.

Il test è stato effettuato tra due macchine reali connesse tramite NIC Ethernet Intel a 1 Gbit/s, cavo Ethernet, switch di rete professionale. Nella configurazione dei contatti di ION la velocità è impostata a 800 Mbit/s, e non 1 Gbit/s, per evitare che siano influenzati dai limiti fisici delle porte Ethernet e dallo switch di rete con cui sono stati fatti gli esperimenti; per valutare tali limiti si è fatto uso di IPerf con UDP, ottenendo velocità raggiungibili attorno ai 900 Mbit/s, vicini quindi al limite teorico, grazie all'ottima qualità delle schede di rete Intel.

La figura seguente mostra il grafico dei goodput prodotti da DTNperf client in 20 esecuzioni: 10 effettuate con ION-LTP (in rosso) e 10 effettuate con Unibo-LTP (in blu). Si può notare che le velocità indicate sono quelle rilevate da un'applicazione DTN, comprensive quindi del livello Bundle Protocol: esse ovviamente dipendono sia dalle schede di rete utilizzate sia dalla velocità dell'hardware, entrambi fattori che definiscono i limiti fisici, in particolare in ricezione, cioè il collo di bottiglia dell'implementazione di ION. Il vantaggio ottenuto dimostra chiaramente che a parità di potenza di calcolo della CPU è ora possibile avvicinarsi al limite fisico delle schede di rete a 1 Gbit/s.

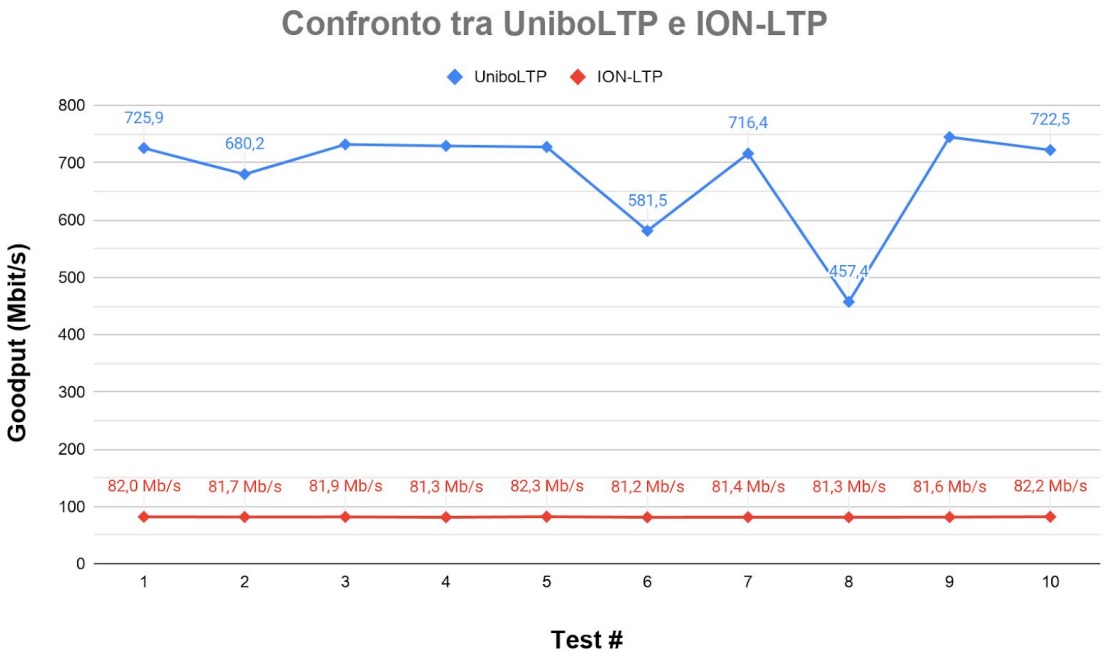

Figura 9. Grafico dei “goodput” ottenuti in 20 esecuzioni di DTNperf: 10 utilizzando Unibo-LTP (in blu) e 10 utilizzando ION-LTP. Si precisa che nel test ci sono state perdite sporadiche utilizzando Unibo-LTP (dati 581 e 457).

In conclusione, l’obiettivo più importante di questo progetto, ovvero il conseguimento di velocità maggiori rispetto all'implementazione presente in ION, è stato pienamente conseguito: sono state raggiunte velocità superiori di circa 10 volte rispetto a quelle dell’implementazione di riferimento, sebbene vada precisato che l’implementazione di ION risulta essere più robusta rispetto ad Unibo-LTP grazie all’utilizzo della memoria SDR.

Il medesimo test è stato anche effettuato incrociando le implementazioni, ovvero configurando la macchina mittente per utilizzare Unibo-LTP ed il ricevente per utilizzare l’LTP presente in ION. In termini di velocità il miglior risultato ottenuto è stato di 220 Mbit/s; ciò dimostra sia la piena interoperabilità tra le due implementazioni (si ricorda che DTNperf client in modalità window based prevede l’invio di un bundle di conferma in direzione opposta), sia che l’LTP di ION risulta lento in trasmissione ed in ricezione.

5.3 Modifiche al buffer di ricezione UDP

Per analizzare come il tempo di elaborazione di un segmento LTP sia influenzato dallo scheduler del sistema operativo, è possibile riutilizzare i dati raccolti con tcpdump nell'esperimento a 800 Mbit/s analizzato nella sezione 5.1: dai tempi di spedizione dei segmenti LTP tramite Excel è immediato calcolare e rappresentare il tempo trascorso tra l'invio di un segmento ed il successivo, come mostrato nella seguente Figura 10. È importante ricordare che a queste velocità l'effetto del token bucket si era dimostrato del tutto trascurabile.

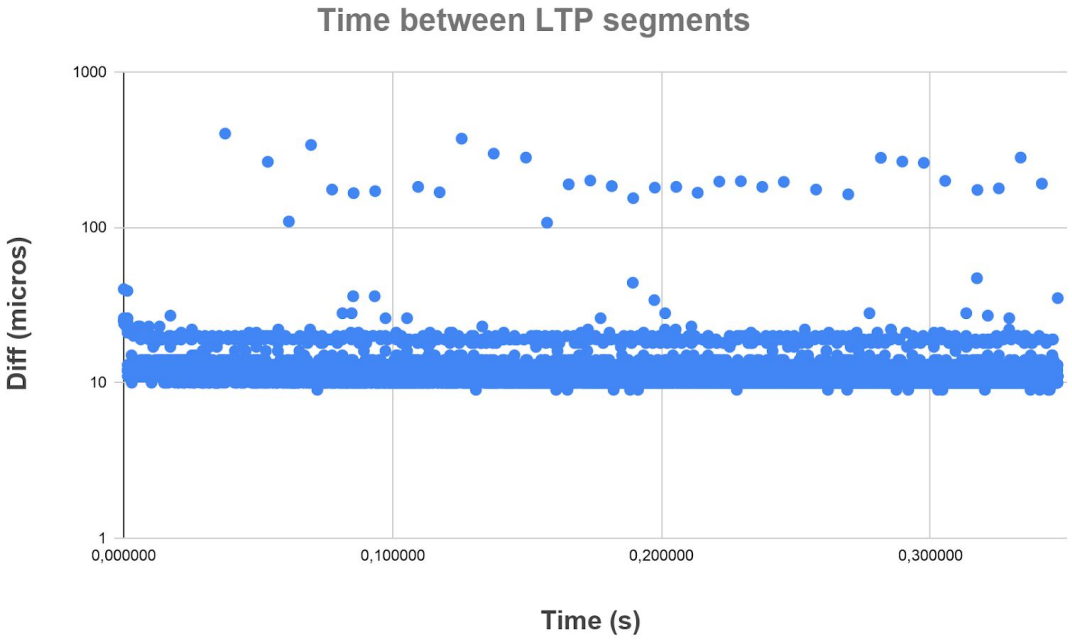

Figura 10. Tempo intercorso tra l'invio di un segmento LTP e quello successivo, a 800 Mbit/s, in Unibo-LTP. Le variazioni di tempo sono presumibilmente dovute allo scheduler del sistema operativo.

Elaborando i dati presentati nella figura sopra, risulta che l'intervallo medio fra segmenti consecutivi è di 11,86 µs; tuttavia, come evidente dalla figura, talvolta la differenza di tempo intercorsa è molto maggiore: queste alterazioni sono presumibilmente causate dallo scheduler del sistema operativo. Dal lato della trasmissione questi ritardi causano un jitter della velocità di trasmissione, facilmente contrastabile con adeguati buffer di trasmissione

e ricezione UDP. Il problema maggiore, tuttavia, si verifica quando il sistema operativo non elabora a velocità costante i segmenti ricevuti: infatti un blocco temporaneo nell'esecuzione dei thread di ricezione, ed in particolare del Receiver thread, può causare un temporaneo aumento dell'occupazione del buffer di ricezione UDP che, se riempito al massimo, causa delle perdite interne per overflow (va tenuto presente che a 800 Mbit/s un ipotetico secondo di interruzione causerebbe un aumento del riempimento del buffer di 100 MB, o in altra scala 10 ms di stasi un riempimento di 1 MB). Entità e frequenze dei ritardi dovuti allo scheduler sembrano casuali, ma in realtà dipendono dalla quantità di lavoro concorrente che deve svolgere la CPU: maggiore quindi il numero dei processi, maggiore la probabilità di incorrere in perdite interne a parità di altre condizioni. A questo proposito, occorre specificare che i dati riportati nelle sezioni precedenti si riferiscono a due macchine scariche, in cui l'unica applicazione significativa, a parte quelle di gestione del sistema operativo, era l'invio e la ricezione dei blocchi LTP. Per far fronte quindi all'imprevedibilità dello scheduler del sistema operativo è possibile adottare due soluzioni: la prima consiste nell'utilizzo di un sistema operativo realtime, ma Unibo-LTP non è stato adattato all'utilizzo di sistemi operativi di questo tipo; la seconda consiste nell'allargamento delle dimensioni del buffer UDP di ricezione. Studiando l'implementazione dell'LTP presente in ION ho notato che le perdite interne dovute al riempimento del buffer UDP di ricezione, drenato troppo lentamente, sono di fatto impedite dall'inserimento dal lato trasmissione di ritardi inter segmento, voluti o impliciti, che di fatto limitano la velocità di trasmissione: rimuovendo o allentando questi limiti si hanno immediatamente delle perdite lato ricezione, dovute all'insufficiente velocità di elaborazione dei segmenti ricevuti. Sottolineo a questo proposito che in ION i segmenti LTP vengono letti e quindi elaborati in sequenza dal medesimo thread, il che comporta una velocità di lettura rallentata quando il segmento letto richiede molto tempo per la sua elaborazione come nel caso ad esempio di un segmento RS indicante dei buchi da riempire. Per alleviare questo problema, nella progettazione di Unibo-LTP ho previsto la separazione dei compiti di lettura a carico del Receiver thread dai compiti di elaborazione ed invio, assegnati al thread "span". In ogni caso nulla può impedire che il Receiver thread sia momentaneamente bloccato dallo scheduler del sistema operativo per dare spazio ai thread di altre applicazioni, per cui resta necessario poter disporre di adeguati buffer di ricezione UDP: tuttavia va precisato che l'allargamento dei buffer non è sufficiente a ridurre le

perdite, poiché consente solo di posticipare l'insorgenza del problema in caso di velocità media di elaborazione dei segmenti LTP minore della velocità di trasmissione. L'allargamento del buffer UDP risulta invece efficace contro possibili ritardi causati dallo scheduler quando la velocità media di elaborazione dei segmenti è superiore alla velocità di trasmissione; per questo motivo i test mostrati in precedenza sono stati eseguiti dopo aver portato il buffer di ricezione UDP a 20 MB.

I numerosi test effettuati in merito hanno permesso di dimostrare che un allargamento dei buffer UDP permette di raggiungere velocità di circa 750 Mbit/s con una velocità del contatto massima posta a 800 Mbit/s.

6 Estensioni dello standard

6.1 Colore “Orange”

Si è deciso di estendere le funzionalità di Unibo-LTP introducendo un nuovo colore intermedio tra green e red, l’*orange*. Unibo-LTP si differenzia dallo standard RFC 5326 in quanto supporta solo blocchi monocromatici, anticipando quanto probabilmente verrà previsto dalla versione rivista dello standard CCSDS. Dato che tutti i segmenti di un blocco sono ora di un medesimo colore possiamo associare questo colore al blocco: avremo quindi blocchi red, green e orange, caratterizzati da livelli di affidabilità diversi. Come già detto il red è affidabile, poiché si basa su una ritrasmissione selettiva dei segmenti persi; il green è invece inaffidabile, in quanto non sono previste né ritrasmissioni né conferme. Il nuovo colore orange invece implementa un servizio non affidabile (con semantica *best effort*, simile al green), senza ritrasmissioni, in cui però il mittente riceve una notifica binaria riguardo alla ricezione o meno di tutti i segmenti del blocco. La notifica del successo/insuccesso è quindi passata dall’LTP mittente al BP.

6.1.1 Motivazioni

Il vantaggio del colore orange rispetto al green è che esso, pur non implementando ritrasmissioni, permette di informare il BP riguardo al successo o all’insuccesso della ricezione del blocco. L’eventuale ritrasmissione dei bundle contenuti nel blocco non consegnato è demandata al BP, procedendo con una ritrasmissione immediata o rimandata. Il comportamento introdotto sembrerebbe a prima vista assurdo rispetto a ritrasmettere, come nel red, i soli segmenti mancanti; tuttavia questa percezione è legata all’esperienza delle reti terrestri, dove l’RTT è brevissimo; quando l’RTT è dell’ordine di minuti o decine di minuti le cose appaiono diverse. La versione standard dell’LTP non ha cognizione di un tempo massimo di consegna, per cui potrebbe accadere che il bundle contenuto in un blocco scada durante i tentativi di ritrasmissione, che potrebbero protrarsi per ore in presenza di canali intermittenti. Se ciò accadesse, l’LTP red rimarrebbe ostinatamente al lavoro per consegnare un blocco al cui interno vi è in realtà un bundle già scaduto. Demandando la ritrasmissione al BP, si ha il vantaggio che questo può chiamare nuovamente l’algoritmo di routing (CGR) e verificare che il bundle possa effettivamente

essere consegnato prima della scadenza, eventualmente scegliendo un nuovo percorso; se non se ne trova uno che può soddisfare il tempo di scadenza del bundle questo può essere scartato immediatamente. Inoltre, dato che i bundle hanno una priorità, la ritrasmissione di un bundle potrebbe essere rimandata in favore della trasmissione di un bundle prioritario; da ultimo, ma fondamentale nei link ad alta velocità e lunga distanza come quelli ottici, l'orange al pari del green non tiene impegnato nessun buffer di ricezione. In linea di principio, nel caso peggiore la tecnica di ARQ selettivo richiede la capacità di memorizzare una quantità di dati pari al prodotto banda ritardo; questo si traduce, ipotizzando un collegamento ottico con Marte, in una quantità enorme di dati, con tutti i problemi del caso, per cui appare preferibile una soluzione diversa, in particolare se accoppiata all'utilizzo di FEC allo strato inferiore che potrebbero recuperare eventuali perdite tramite l'aggiunta di segmenti di ridondanza. Si veda a questo proposito [15] [22].
Un confronto riassuntivo dei vari colori è riportato nella tabella sotto:

Color	Service	Retransmissions	Rx buffers
Red	Reliable	LTP segments	Many
Orange	Notified	Left to BP (new LTP blocks)	One/two
Green	Unreliable	None	One/two

Tabella 2. Comparazione dei colori LTP comprensivi del nuovo colore (orange) introdotto.

6.1.2 Implementazione

Il colore orange è stato implementato senza violare in alcun modo le specifiche dell'RFC 5326, limitando le modifiche al codice a pochi strategici punti.
Lo standard prevede che l'*header* di un qualunque segmento contenga 4 bit che indicano la sua tipologia; tuttavia 4 delle 16 possibilità non sono definite (codici 5, 6, 10 e 11), pertanto si è deciso di sfruttare due di quelle libere per introdurre due nuovi tipi di segmenti, in maniera simile a quanto già presente con i segmenti green. Abbiamo quindi un segmento dati "orange" (codice 5) ed un "orange EOB" (End Of Block; codice 6) che permette di indicare la fine del blocco. Alla ricezione del segmento "orange EOB", il

ricevitore controlla se tutti i segmenti della sessione sono stati ricevuti, come nel caso del red: in caso affermativo avvisa il mittente tramite un RS di conferma in modo identico al red. Tuttavia, in caso contrario, anziché generare uno o più RS parziali, la sessione di ricezione viene cancellata tramite CR, che funge da notificata al mittente. Sia l'RS che il CR prevedono un meccanismo di protezione dalle perdite tramite ritrasmissione se non confermati dopo un RTO. Dato che gli RS ed i CR non hanno colore, il comportamento è stato mantenuto identico a quello già implementato per i red; essi verranno ritrasmessi finché non saranno confermati dall'arrivo di un RAS o un CAR, oppure, nel peggiore dei casi, dall'esaurirsi del numero massimo di ritrasmissioni ammesso, come previsto dallo standard RFC 5326.

Nella Figura 11 è riportato un esempio di sessione orange con ricezione di tutti i segmenti.

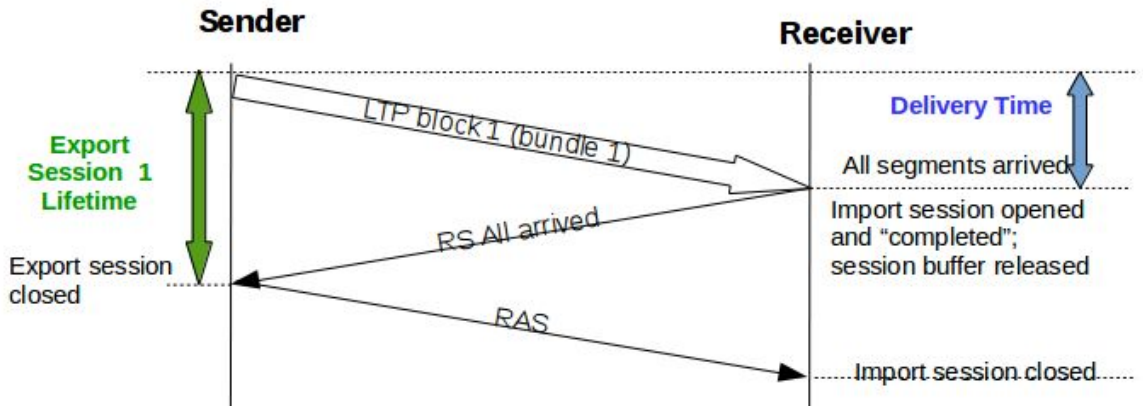

Figura 11. Esempio di sessione Orange con ricezione di tutti i segmenti e notifica al mittente mediante RS.

Segue ora la Figura 12 che mostra un esempio di ricezione parziale del blocco LTP, contenente il bundle 2, durante la sessione 2, seguita da una nuova sessione di trasmissione (la 3), causata dalla ritrasmissione del bundle 2 a livello BP.

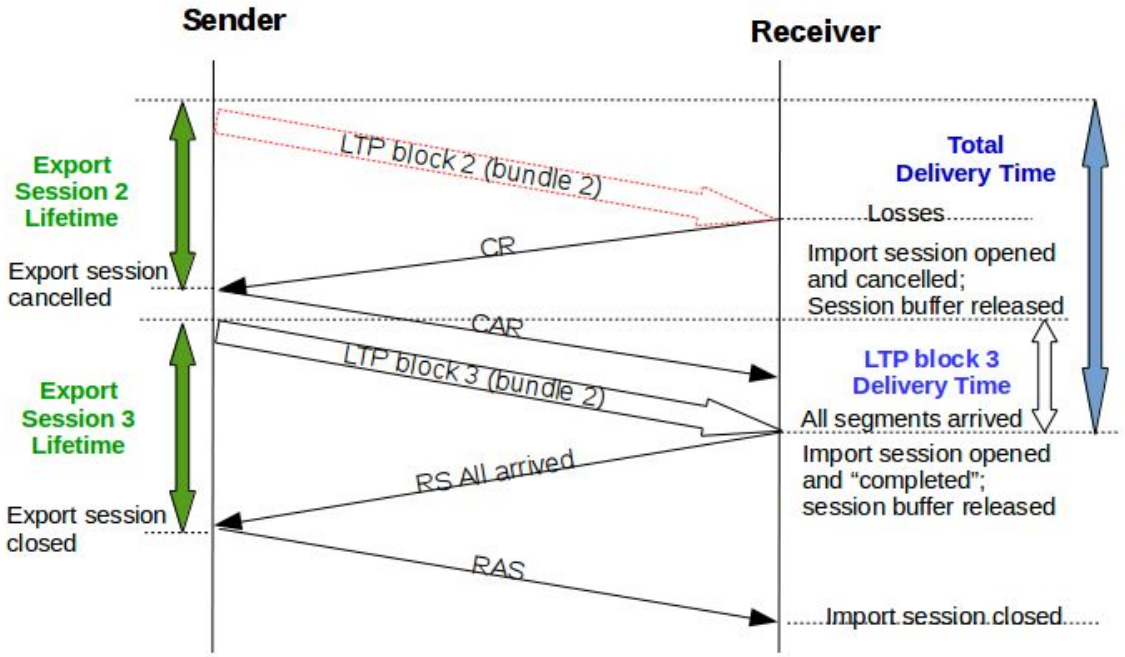

Figura 12. Esempio di sessione Orange con perdite e conseguente cancellazione della sessione con notifica al mittente mediante CR (sessione 2). La sessione 3, immediatamente successiva è dovuta alla ritrasmissione a livello BP del bundle 2.

Conclusioni

La prima conclusione che possiamo trarre è che l'obiettivo principale di questa tesi è stato raggiunto: come dimostrato dai test effettuati, Unibo-LTP permette di ottenere velocità di trasmissione superiori di 6-7 volte rispetto a quelle raggiungibili con l'implementazione di riferimento, quella di ION, mantenuta dalla NASA-JPL.

Per ottenere queste prestazioni, Unibo-LTP è stato strutturato su più thread: un Receiver thread unico, con il compito di rimuovere i segmenti LTP il più velocemente possibile dal buffer di ricezione UDP per evitare perdite dovute ad overflow, e tre thread specifici per ogni "span", ovvero per ogni nodo vicino, con il duplice compito di elaborare i segmenti di segnalazione passati dal Receiver e di creare nuove sessioni di invio nel caso fossero passati dati da inviare in direzione opposta. Il codice è stato progettato in modo modulare per facilitare future modifiche; in particolare contiene due moduli, per il protocollo superiore (di norma il BP) e quelli inferiori (di norma l'UDP), che consentono di cambiare l'implementazione del Bundle Protocol utilizzata e/o i protocolli sottostanti senza toccare il nocciolo del codice. È stata inoltre posta particolare attenzione ai log, rendendoli autoesplicativi e molto dettagliati, in modo che essi possano essere utili non solo a fini di debug, ma anche a scopi didattici.

L'ultima parte della tesi è stata dedicata alla realizzazione di estensioni del protocollo. In particolare è stato studiato ed implementato un nuovo "colore", l'orange, intermedio tra i colori red e green previsti dallo standard. Esso fornisce un servizio "notificato", senza ritrasmissioni di segmenti (quindi best effort) ma con segnalazione dell'arrivo o meno di tutti i segmenti di un blocco. Inoltre, è stata realizzata un'interfaccia per il protocollo inferiore in modo che i segmenti LTP siano passati al protocollo ECLSA, anziché direttamente a UDP, per proteggerli con codici a correzione di errore (FEC) operanti a livello di pacchetto, cosa molto utile per limitare le perdite e quindi la necessità di ritrasmissioni dei segmenti LTP. In questo contesto, la proposta di introdurre un nuovo colore, l'orange, si dimostra particolarmente interessante, in quanto demanderebbe al protocollo BP l'onere di eventuali ritrasmissioni, rese però rarissime dai FEC, evitando in questo modo la necessità di disporre di grandi buffer di ricezione a livello LTP.

In conclusione, l'auspicio è che questa implementazione di LTP possa essere effettivamente testata su link ottici sperimentali, in particolare del DLR (l'ente aerospaziale tedesco) con il quale l'Università di Bologna ha in corso da anni una proficua attività di collaborazione. Inoltre potrebbe essere di notevole aiuto nel sostenere la proposta dell'inserimento del nuovo colore orange all'interno della versione rivista dello standard CCSDS: come dice Linus Torvalds "Talk is cheap, show me the code"…

Bibliografia

[1] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, H. Weiss, “Delay-Tolerant Networking Architecture”, Internet RFC 4838, April. 2007, https://tools.ietf.org/html/rfc4838 Last Access: 25/02/2021.

[2] K. Scott, S. Burleigh, “Bundle Protocol Specification”, Internet RFC 5050, Nov. 2007, https://tools.ietf.org/html/rfc5050 Last Access: 25/02/2021.

[3] CCSDS 734.2-B-1 “CCSDS Bundle Protocol Specification”, CCSDS Blue Book, Issue 1, Sept. 2015. https://public.ccsds.org/Pubs/734x2b1.pdf Last Access: 25/02/2021.

[4] S. Burleigh, K. Fall, E. Birrane, “Bundle Protocol Version 7”, IETF Draft, January 2021, https://tools.ietf.org/html/draft-ietf-dtn-bpbis Last Access: 25/02/2021.

[5] E. Birrane, K. McKeever, IETF Draft, January 2021, https://tools.ietf.org/html/draft-ietf-dtn-bpsec Last Access: 25/02/2021.

[6] B. Sipos, M. Demmer, J. Ott, S. Perreault, February 2021, https://tools.ietf.org/html/draft-ietf-dtn-tcpclv4 Last Access: 25/02/2021.

[7] N. Alessi, A. Bisacchi, C. Caini, G. M. De Cola, L. Persampieri, "DTN research for Space Communications", IEEE SSC (Satellite and Space Communications) Newsletter, Vol. 30, No. 1, June 2020, https://ssc.committees.comsoc.org/files/2020/06/sscnlv30n1.pdf Last Access: 25/02/2021.

[8] S. Burleigh; “Interplanetary Overlay Network (ION) - Design and Operation” https://sourceforge.net/projects/ion-dtn/, Jet Propulsion Laboratory, 2012, Last Access: 25/02/2021.

[9] S. Burleigh, M. Ramadas, S. Farrell, “Licklider Transmission Protocol – Motivation”, Internet RFC 5325, Sep. 2008, https://tools.ietf.org/html/rfc5325 Last Access: 25/02/2021.

[10] M. Ramadas, S. Burleigh, S. Farrell, “Licklider Transmission Protocol – Specification”, Internet RFC 5326, Sep. 2008, https://tools.ietf.org/html/rfc5326 Last Access: 25/02/2021.

[11] M. Ramadas, S. Burleigh, S. Farrell, “Licklider Transmission Protocol – Security Extensions”, Internet RFC 5327, Sep. 2008, https://tools.ietf.org/html/rfc5327 Last Access: 25/02/2021.

[12] CCSDS 734.1-B-1 “LICKLIDER TRANSMISSION PROTOCOL (LTP) FOR CCSDS”, CCSDS Blue Book, Issue 1, May 2015, https://public.ccsds.org/Pubs/734x1b1.pdf Last Access: 25/02/2021.

[13] N. Alessi, S. Burleigh, C. Caini, T. de Cola; “Design and performance evaluation of LTP enhancements for lossy space channels“, March 2017.

[14] Valgrind, https://valgrind.org Last Access: 25/02/2021.

[15] N. Alessi, C. Caini, T. de Cola and M. Raminella, "Packet Layer Erasure Coding in Interplanetary Links: The LTP Erasure Coding Link Service Adapter," in IEEE Transactions on Aerospace and Electronic Systems, vol. 56, no. 1, pp. 403-414, Feb. 2020, DOI: 10.1109/TAES.2019.2916271.

[16] Generic-List, https://gitlab.com/andreabisacchi/Generic-List Last Access: 25/02/2021.

[17] json-c, https://github.com/json-c/json-c Last Access: 25/02/2021.

[18] Unibo-LTP, https://gitlab.com/andreabisacchi/ltp Last Access: 25/02/2021.

[19] Doxygen, https://www.doxygen.nl/index.html Last Access: 25/02/2021.

[20] Documentazione Unibo-LTP, https://andreabisacchi.gitlab.io/ltp/ Last Access: 25/02/2021.

[21] C. Caini, A. d'Amico and M. Rodolfi, "DTNperf_3: A further enhanced tool for Delay-/Disruption- Tolerant Networking Performance evaluation" in Proc. GLOBECOM 2013, Atlanta, GA, US, 2013, pp. 3009-3015.

[22] N. Alessi, C. Caini, A. Ciliberti and T. de Cola, "HSLTP—An LTP Variant for High-Speed Links and Memory Constrained Nodes," in IEEE Transactions on Aerospace and Electronic Systems, vol. 56, no. 4, pp. 2922-2933, Aug. 2020, DOI: 10.1109/TAES.2019.2958190.

Ringraziamenti

Desidero esprimere un ringraziamento particolare alle persone che sono state importanti durante il quinquennio di studi che si sta concludendo.

Ringrazio infinitamente la mia mamma e Furio per avermi sempre messo al primo posto, per il sostegno costante e l'affetto che mi hanno dato ogni giorno supportando le mie scelte e sacrificandosi sempre per me. A voi dedico questo grande traguardo.

Un ringraziamento va anche ai miei amici, senza i quali l'università non sarebbe stata la stessa e studiare sarebbe stato molto più difficile e pesante: in particolare Fabrizio, Giordan, Danilo e Marco, coi quali ho preparato praticamente tutti gli esami.