UNIVERSITÀ DI PISA
FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Tesi di laurea:

PROGETTO E REALIZZAZIONE DI UN PROTOCOLLO DI TRASPORTO PER RETI AD HOC MULTI-HOP

Candidato:

Emilio Ancillotti

Relatori:

Prof. Giuseppe Anastasi

Prof. Marco Avvenuti

Ing. Enrico Gregori

Dott. Marco Conti

ANNO ACCADEMICO 2002/2003

C

Capitolo I.

Introduzione

Il diffondersi dei computer portatili (notebooks, palm tops, PDAs, smart phones) e lo sviluppo delle tecnologie wireless ha stimolato l'interesse verso le reti Ad Hoc con nodi mobili (Mobile Ad Hoc Networks o MANETs). Una MANET è una rete formata da un insieme di nodi mobili che comunicano in maniera wireless senza richiedere la presenza di alcuna infrastruttura di rete fissa. I vari nodi della rete, comunicano in maniera multi-hop. Più precisamente, la comunicazione tra due nodi che non si trovano nel raggio di copertura reciproco, si appoggia su altri nodi intermedi della rete, che inoltrano il pacchetto verso il destinatario finale (in tali reti, i nodi devono comportarsi anche da routers). La topologia delle MANET, essendo i nodi mobili, è in continua evoluzione e cambia ogni qualvolta lo spostamento di un nodo provoca l'attivazione di un nuovo link wireless (nodo che si sposta dentro il raggio di copertura di un altro nodo) o la disconnessione di un link esistente (nodo che muovendosi esce dal raggio di copertura di un altro nodo). Le MANET, possono essere utilizzate in tutte quelle situazioni in cui non è possibile fornire delle infrastrutture di comunicazione o per motivi economici e fisici o per la particolarità delle circostanze in cui è richiesto il loro utilizzo (supporto in situazioni di soccorso).

Negli anni passati, le attività di ricerca nel campo delle MANET si sono concentrate principalmente sullo studio dei protocolli di routing. Tuttavia, diversi altri studi hanno anche evidenziato come le prestazioni del protocollo TCP nelle MANET risultino scadenti. Il protocollo TCP è stato progettato per reti wired in cui i nodi sono statici e la quasi totalità dei pacchetti vengono persi a causa di fenomeni di congestione ai router intermedi. In tale scenario, l'algoritmo di gestione della congestione del TCP costituisce un efficiente meccanismo di controllo della congestione. Questo meccanismo lavora bene fino a quando la percentuale di pacchet-

ti persi per errori di trasmissione e *route failure* (invalidazione della strada seguita dai pacchetti per giungere a destinazione) è molto bassa. Le MANETs, si comportano in maniera differente dalle reti wired tradizionali (come Internet). In particolare, in tale ambiente sono frequenti gli eventi di *route failure* e *route change* (cambiamento della strada seguita dai pacchetti per giungere a destinazione) i quali possono provocare la perdita di pacchetti e/o la ricezione di pacchetti fuori sequenza. Inoltre, il fenomeno della congestione nelle MANET [9], è differente rispetto a quello che si verifica nelle reti wired. L'origine di tale fenomeno è da ricercare nel protocollo di accesso basato su contesa, mentre il numero di pacchetti persi per buffer overflow ai nodi intermedi risulta trascurabile (se i buffer sono di dimensionati in modo appropriato). Da quanto detto, risulta evidente come il TCP non lavori bene in ambiente MANET, poiché il suo comportamento è svincolato dall'effettivo stato della rete. L'unica informazione che il TCP ha sullo stato della rete, è quella relativa alla perdita di pacchetti. Tale perdita, viene presa dal TCP come indicazione di congestione di qualche router intermedio e determina l'attivazione del meccanismo di controllo della congestione. Nelle MANETs, le cause di perdita di pacchetti sono varie e raramente sono riconducibili alla congestione. Questo spiega le scarse prestazioni del TCP in tale ambiente.

Un protocollo di livello trasporto, per lavorare correttamente in ambiente MANET, deve avere dei meccanismi per determinare lo stato della rete (cioè per determinare eventuali fenomeni di congestione, *route-failures*, *route-changes*) e deve essere in grado di modificare il proprio comportamento in funzione di questo. Per raggiungere tale obiettivo si possono seguire due approcci. L'approccio tradizionalmente seguito si basa sulla modifica del TCP standard al fine di renderlo adatto ad operare correttamente anche in ambiente MANET [1-11]. Il secondo approccio consiste nella realizzazione di un nuovo protocollo di trasporto progettato appositamente per le MANET.

Numerosi studi hanno evidenziato come nelle MANET il numero di pacchetti che una connessione può avere in volo senza sovraccaricare la rete, risulta essere basso (intorno a 3-4). Questo, rende inutile l'utilizzo di un meccanismo di gestione

della congestione come quello utilizzato dal TCP. Tale osservazione, unita alla necessità di studiare dei meccanismi in grado di minimizzare il numero di ritrasmissioni non necessarie di pacchetti (per risparmiare energia e minimizzare il carico di lavoro offerto alla rete) e di rilevare lo stato della rete (al fine di adattarsi ad esso), ci hanno portato a seguire il secondo approccio. In tale ottica è stato definito e realizzato il protocollo TPA (Transport Protocol for Ad Hoc), pensato appositamente per le MANET. Esso fornisce un servizio di trasporto affidabile e connection-oriented, e presenta diverse caratteristiche innovative rispetto al protocollo TCP. In particolare, il TPA è in grado di gestire le varie situazioni che si possono verificare a causa della mobilità dei nodi, quali le condizioni di *route failure* e di *route change*. Inoltre, il meccanismo di controllo della congestione del TPA viene completamente ridisegnato rispetto a quello del TCP. Tale meccanismo permette al nuovo protocollo di ridurre al minimo il carico di lavoro offerto alla rete (utilizza una finestra di controllo della congestione di domensioni ridotte), al fine di ridurre al minimo il fenomeno della congestione. Per finire, il TPA implementa una nuova politica di ritrasmissione dei pacchetti persi (o supposti tali) cercando di ridurre il numero di ritrasmissioni non necessarie. Questo permette di diminuire il carico sulla rete e di risparmiare energia.

La tesi è organizzata come segue. Il capitolo 2 descrive le limitazioni del protocollo TCP in ambiente MANET e riporta lo stato dell'arte sulle soluzioni proposte per migliorarne le prestazioni. Il capitolo 3 esamina i limiti di tali soluzioni e fornisce una descrizione dettagliata del protocollo TPA. Il capitolo 4 fornisce una descrizione dell'ambiente di sviluppo e simulazione QualNet Developer. Il capitolo 5 descrive l'implementazione del protocollo TPA in ambiente QualNet. Nel capitolo 6 sono riportati alcuni risultati preliminari sulle prestazioni del protocollo TPA. Infine, le conclusioni sono riportate nel capitolo 7.

Capitolo II.

Stato dell'arte

Nel seguente capitolo sono mostrati i principali risultati ottenuti dallo studio delle performance del protocollo TCP in ambiente MANET, facendo una carrellata dei vari problemi che ne affliggono il comportamento e delle varie soluzioni che sono state proposte in letteratura per cercare di mitigare tali problemi.

2.1 Problemi del TCP in ambiente MANET

Il protocollo TCP è stato progettato per reti wired in cui i nodi sono statici e la percentuale di pacchetti persi a causa d'errori di trasmissione e *route-failures* (l'invalidazione della strada seguita dai pacchetti per andare a destinazione) è molto bassa. In tale scenario, la strategia di *additive-increase/multiplicative-decrease*, unita ai meccanismi di *fast recovery e fast retransmit* ([18], [19]), fornisce un efficiente meccanismo di controllo della congestione. L'idea principale su cui si basa il meccanismo di congestione del TCP, è quella di testare la rete per determinare la disponibilità di risorse: i pacchetti sono spediti nella rete ad un rate crescente, fino a quando non si presenta una perdita di pacchetti. Il TCP, a questo punto, ipotizza che la perdita di pacchetti sia causata dalla congestione della rete, ed intraprende le seguenti azioni: restringe la *congestion window,* ritrasmette i pacchetti persi e riprende la trasmissione ad un rate più basso (rincrementandolo fino a quando non si presenta una nuova perdita di pacchetti). Se scattano più timeout consecutivi, ad ogni ritrasmissione viene raddoppiato il RTO (*Retransmission Timeout*), secondo lo schema di *exponential backoff*. Questo meccanismo di controllo della congestione, lavora bene fino a quando la percentuale di pacchetti persi per errori di trasmissione e *route-failures* è molto bassa. In ambiente MANET, il considerare la congestione come l'unica causa di perdita di

pacchetti, non è una buona scelta, e porta il TCP a comportarsi in maniera errata in molti frangenti.

La mobilità dei nodi porta frequentemente a problemi di *route-failures* e *route-changes* (cambiamento della strada seguita dai pacchetti per andare a destinazione), i quali possono provocare la perdita e la ricezione di pacchetti fuori ordine. Il TCP, non essendo in grado né di gestire né di rilevare tali situazioni, interpreta lo scattare dei timeout e la ricezione degli ack duplicati come indice di congestione della rete, ed invoca il meccanismo di controllo della congestione, causando un certo numero di ritrasmissioni non necessarie e un degrado del throughput (cap. 2.1.1).

Il comportamento del TCP risulta scadente anche in reti MANET statiche, in quanto presentano caratteristiche molto differenti rispetto alle reti wired classiche. In reti *wired* come Internet, la perdita di pacchetti è causata quasi esclusivamente dalla congestione, provocata da un sovraccarico dei buffer dei routers del path. Nelle MANETs, la congestione causata da un sovraccarico dei buffer intermedi è un evento raro e la causa principale di perdita di pacchetti è dovuta alla *link-layer contention*. Il TCP non è in grado di distinguere tra pacchetti persi a causa di *buffer overflow* e pacchetti persi a causa della contesa e reagisce in entrambi i casi invocando il meccanismo di controllo della congestione, portando ad un degrado del throughput (cap. 2.1.2).

Di seguito sono analizzate in dettaglio le situazioni descritte in precedenza.

2.1.1 Mobilità

In questo capitolo sono analizzati i problemi che la mobilità dei nodi induce sul comportamento del TCP. Cerchiamo di capire come funziona l'instradamento in una *mobile ad hoc Network*. Ogni *mobile host* (**MH**) appartenente alla rete coopera alla spedizione dei dati, permettendo a pacchetti che non sono a lui destinati di passargli attraverso per arrivare a destinazione. In altre parole, ogni *host* si comporta, se necessario, da *router*. Prendiamo la situazione descritta in figura 1:

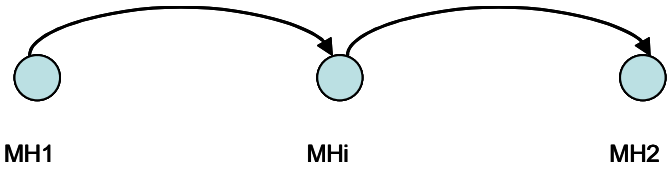

Figura 1 – Instradamento dei pacchetti in reti mobile ah Hoc.

Il MH1 deve spedire un pacchetto al MH2, ma il MH2 non si trova nel raggio di trasmissione del MH1. In questo caso, il MH1 spedisce il pacchetto ad uno dei suoi vicini (MHi) lasciando al MHi il compito di far pervenire il pacchetto a destinazione (MHi si comporta da *router*). Se il MHi non riesce a raggiungere il MH2, il procedimento si ripete. In generale, il pacchetto per arrivare a destinazione attraverserà un certo numero di nodi intermedi, seguendo la strada stabilita dal protocollo di routing. La topologia di una *mobile ad hoc Network* è dinamica, e cambia ogni qualvolta lo spostamento di un host provoca lo stabilirsi di un nuovo wireless link (host che si sposta dentro il raggio di trasmissione di un altro host) o la disconnessione di un link esistente (host che era nel raggio di trasmissione di un host e muovendosi ne esce). Il cambiamento topologico della rete (anche lo spostamento di un singolo host) può portare all'invalidazione di una o più strade trovate in precedenza dal protocollo di routing (il protocollo di routing deve essere abile nell'inseguire i cambiamenti topologici della rete), causando numerosi problemi al TCP. Mettiamoci nella situazione di figura 2, in cui tra il nodo 1 ed il nodo 7 è presente una connessione:

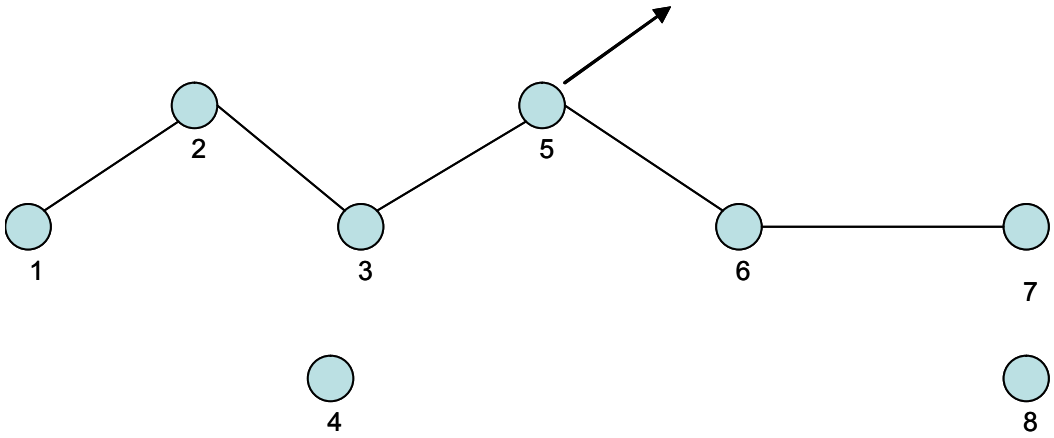

Figura 2 - Il movimento del nodo 5 interrompe la strada tra 1 e 7.

Supponiamo che il nodo 1 abbia dei pacchetti da spedire al nodo 7, e cominci la trasmissione. Ad un certo istante, il nodo 5 si sposta interrompendo la strada. Il livello MAC del nodo 3, non riuscendo a raggiungere il nodo 5, riporta un messaggio di *link-failures* (rottura del collegamento tra i due nodi) al suo livello superiore, forzando il protocollo di routing ad iniziare una fase di ricerca di una strada alternativa per andare a destinazione. Tale fase può essere risolta localmente dal nodo 3, se conosce una strada alternativa, altrimenti viene inviato un messaggio di *route-failures* alla sorgente del pacchetto, forzando la ricerca di un nuovo percorso. Tale ricerca, impiega un tempo non trascurabile per essere portata a compimento e causa lo scattare di diversi timeout presso il Sender. Inoltre, si possono perdere i pacchetti che si trovavano nel tratto di percorso tra la sorgente e il nodo che ha rilevato la *link-failures*, provocando, presso il Sender, lo scattare dei relativi timeout. Il TCP, come detto in precedenza, interpreta lo scattare dei timeout e la ricezione di un certo numero di ACK duplicati, come indice di congestione, e si comporta in modo errato.

Esaminiamo il caso di figura 3 in cui la sorgente e la destinazione si vengono a trovare in partizioni distinte della rete e non sono raggiungibili tra loro (*partizione della rete)*:

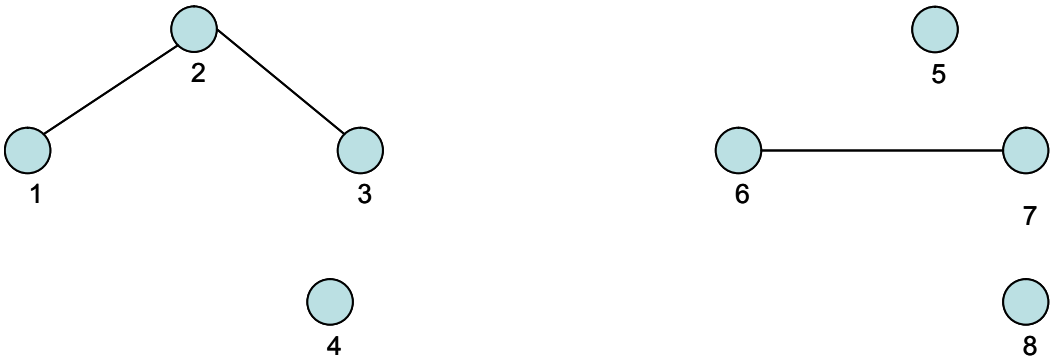

Figura 3 - I nodi 3 e 4 non sono in grado di comunicare con il nodo 6 e pertanto la rete risulta partizionata.

Dopo che il nodo 5 si è spostato, la rete si viene a trovare divisa in due e, né il nodo 3 né il nodo 4 sono in grado di raggiungere il nodo 6. Siamo nella condizione di partizione della rete. La partizione dura fino a quando il nodo 8, che è in movimento, non si viene a trovare tra il nodo 4 ed il nodo 6 (figura 4).

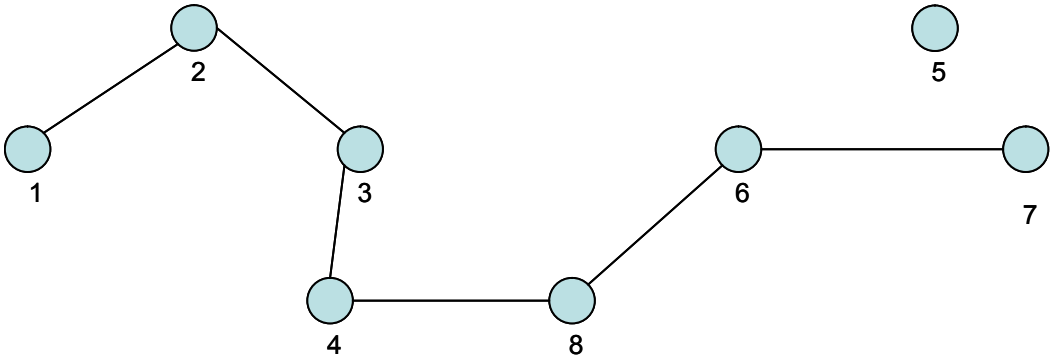

Figura 4 - Il nodo 8 muovendosi ristabilisce un contatto tra le due partizioni della rete.

A questo punto, il protocollo di routing è in grado di trovare una strada tra 1 e 7. Il TCP, utilizzando solo la perdita di pacchetti come traccia dello stato della rete, non si accorge della mancanza della strada per andare a destinazione e continua la sua attività in modo normale (continua la spedizione di pacchetti fino al raggiungimento della dimensione massima della finestra). Presso il nodo 1 scatteranno diversi timeout, in quanto, né i pacchetti né gli ack spediti dal TCP possono essere

inoltrati alla relativa destinazione. Il livello TCP del nodo 1, allo scattare dei timeout, esegue le seguenti azioni:

- Invoca il meccanismo di controllo della congestione (calcolo del RTO con l'algoritmo di backoff e riduzione della finestra di controllo ad un pacchetto).
- Entra nella fase di *slow start*.
- Ritrasmette i pacchetti che sono andati in timeout.

Questo comportamento è inaccettabile poiché:

- Finché non è disponibile una strada, è inutile ritrasmettere i pacchetti allo scattare del RTO, perché questi non possono raggiungere la destinazione.
- Appena viene ritrovata una strada, il throughput della connessione si viene a trovare ad un livello molto basso (*slow start recovery*) anche se la rete non è congestionata.
- La politica con cui viene calcolato il RTO in presenza di congestione (*exponential backoff*), peggiora ulteriormente le cose, perché se la strada rimane per molto tempo non disponibile, il RTO scatta varie volte (si hanno tante ritrasmissioni) e viene così ad assumere un valore molto elevato che degrada ancora di più il throughput (il Sender è come se testasse la disponibilità di una strada ad intervalli pari al RTO e quindi non è in grado di rilevare una nuova strada appena diventa disponibile).

I problemi esaminati in precedenza restano anche se non si viene a creare una partizione della rete. In generale, i protocolli di routing impiegano un tempo non trascurabile per recuperare da una *route-failures*, causando lo scattare di diversi timeout presso la sorgente anche in assenza di partizione. Questo, porta all'invocazione del meccanismo di controllo della congestione e al presentarsi dei problemi visti in precedenza. Un caso particolare si ha quando la strada viene ritrovata localmente dal nodo che ha rilevato la *link-failures*, utilizzando i propri vicini. In questo caso, il recupero del path è veloce e non provoca né la perdita di pacchetti né l'assenza prolungata di un percorso per andare a destinazione. In tale situazione, non si presentano i problemi visti in precedenza ma possiamo andare

incontro alla perdita dell'ordinamento dei pacchetti che arrivano a destinazione (fenomeno degli *out-of-order*). Mettiamoci nel caso di figura 5 e supponiamo che la sorgente spedisca nell'ordine i pacchetti P1 e P2.

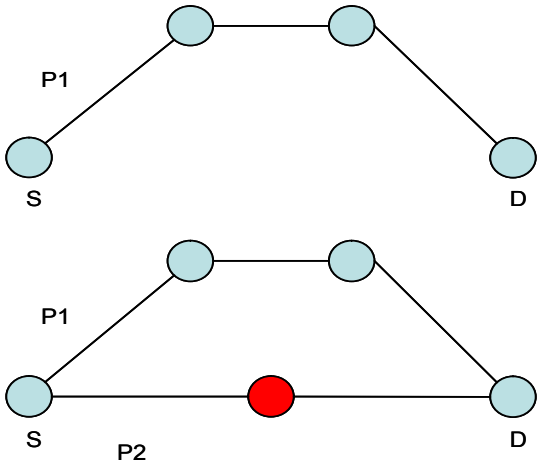

Figura 5 – fenomeno degli out-of-order.

Dopo la spedizione del pacchetto P1, tra S e D si viene ad inserire un nodo A, e la strada per andare da S a D cambia (figura 5). La strada seguita dal pacchetto P2 è più corta rispetto a quella seguita dal pacchetto P1 e quindi P2 può arrivare a destinazione prima di P1. Il fenomeno degli *out-of-order*, può portare il Sender a ricevere tre ack duplicati (si viene a creare un buco nella finestra di controllo del Receiver) e ad invocare il meccanismo di controllo della congestione, con conseguente degrado del throughput.

Un altro problema a cui si può andare incontro, è causato dalle *route-changes*. Il cambiamento di percorso tra sorgente e destinazione (e viceversa), può portare a fenomeni di sovra e sottostima del *retransmission timeout*. Una **sottostima del RTO** si ha quando il RTT del pacchetto è più elevato del valore del RTO associatogli. Una situazione di questo genere si presenta quando il nuovo path è più lungo di quello precedente. Una sottostima può portare allo scattare dei timeout prima del ricevimento degli ACK, facendo passare per persi pacchetti che in realtà non lo sono, e causando l'invocazione del meccanismo di controllo della conge-

stione (portando ad un degrado del throughput). Una **sovrastima del RTO,** si ha quando il RTT del pacchetto è molto più basso del valore del RTO associatogli (il nuovo path è più corto di quello precedente). Questo comporta che il Sender si accorga della perdita di un pacchetto con eccessivo ritardo.

Da quanto detto, risulta evidente che, nelle *mobile ad hoc Networks,* lo scattare dei timeout e la ricezione di ACK duplicati non possono più essere presi come indice di congestione della rete. Di conseguenza, il TCP per funzionare correttamente dovrebbe essere messo nella condizione di poter distinguere tra le varie cause di perdita di pacchetti, in modo da invocare il meccanismo di controllo della congestione solo quando necessario. In letteratura esistono due approcci diversi per cercare di migliorare il comportamento del TCP. Il primo approccio si basa sull'utilizzo di una notifica esplicita da parte del livello network. Tale notifica, informa il TCP dell'evento di *route-failures* mettendolo nella condizione di poter distinguere tra la perdita di pacchetti causata dalla congestione e la perdita di pacchetti causata da una *route-failures*. Tra le varie soluzioni di questo tipo si trovano: TCP-F [1], ENIC [2], TCP-Bus [3], TCP-ELFN [4], ATCP [11]. Il secondo approccio è di tipo *end-to-end* e non richiede nessuna notifica da parte del livello network. Tramite la misurazione di alcuni parametri da parte dei due capi della connessione, il TCP viene modificato in modo da poter rilevare le *route-changes*. Tra queste soluzioni si trovano: Fixed RTO and out-of-order detection [5] [6], ADTCP [7].

2.1.2 Interazione tra MAC e TCP

Numerosi studi ([12], [8], [9]) hanno evidenziato che il TCP ha un comportamento non accettabile anche in ambienti statici, vale a dire in reti Ad-Hoc in cui i nodi non si muovono (non si hanno perdite di pacchetti dovute alla mobilità). Il TCP, su reti Ad-Hoc di questo tipo, mostra una serie di comportamenti inaspettati che si possono così riassumere:

- *Instability problem* - Il throughput delle connessioni, TCP può essere molto instabile (si avvicina molto a zero e poi risale mostrando un andamento altalenante).
- *Incompatibility problem* - Può capitare che due connessioni non possano spedire dati contemporaneamente. Quando trasmette una connessione l'altra non riesce ad accedere al canale e viceversa.
- *Unfairness problem* - La capacità del canale non viene condivisa equamente tra le varie connessioni.

Le cause di questi problemi hanno origine al livello fisico e al livello MAC, e le varie problematiche vengono risaltate dalla interazione tra il livello MAC ed meccanismo di controllo di congestione del TCP. Diamo un rapido sguardo al protocollo di livello MAC descritto nello standard IEEE 802.11, per capire meglio il comportamento scadente del TCP in reti Ad-Hoc.

Lo standard IEEE 802.11 [13] specifica sia il livello fisico sia il livello MAC delle WLAN. Il livello MAC da tale standard mette a disposizione due differenti metodi d'accesso: il *Distributed Coordination Function* (***DCF***) e il *Point Coordination Function* (***PCF***).

Il *DCF* è il metodo di accesso base dello sandard IEEE 802.11 e si basa sul protocollo *carrier sense multiple access with collision avoidance* (***CSMA/CA***). Le reti Ad-Hoc utilizzano il metodo d'accesso *DCF*.

Distributed coordination function. Un protocollo di tipo *CSMA/CA* lavora nel modo seguente: quando una stazione vuole spedire un pacchetto, testa il canale per sentire se è in corso una trasmissione. Se il canale è trovato libero per un intervallo di tempo maggiore del *Distributed Inter Frame Space* (***DISF***) la stazione è abilitata a trasmettere. Per garantire un accesso *fair* al canale, una stazione che ha appena trasmesso un pacchetto e ne ha un altro pronto per la trasmissione, deve eseguire l'algoritmo di *backoff* prima di iniziare la seconda trasmissione. Se il canale è trovato occupato (una stazione vicina sta trasmettendo) la stazione deferisce la trasmissione ad un momento successivo. Per fare questo, seleziona un intervallo random (*backoff time*) e lo usa per inizializzare un timer (*backoff timer)*. Il *backoff*

timer viene decrementato per tutto il tempo in cui il canale è sentito libero, stoppato quando viene rilevata una trasmissione sul canale e riattivato quando il canale è sentito libero per un tempo maggiore di *DISF* (figura 6)**.**

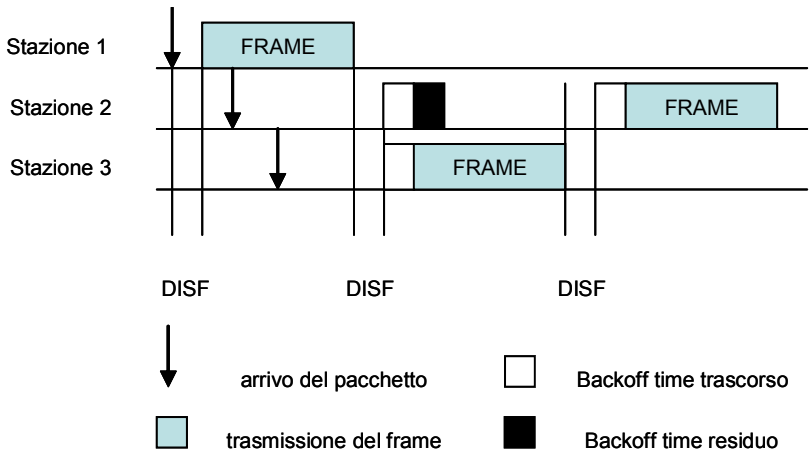

Figura 6 - Meccanismo base di accesso al canale.

La stazione è abilitata a trasmettere quando il *backoff timer* raggiunge lo zero. Il *DCF* adotta una *slotted binary exponential backoff technique*. In particolare, il tempo immediatamente seguente un *DISF* libero è slottizzato, e una stazione è abilitata a trasmettere solo all'inizio di ciascuno *Slot Time* (uguale al tempo necessario ad una stazione per accorgersi di una trasmissione da parte di un altro pacchetto). Il *backoff time* viene scelto in un intervallo uniforme pari a (0, *CW*-1), detto *Backoff Window* o *Contention Window*. Al primo tentativo di trasmissione, è posto *CW* = *CWmin*, e *CW* viene raddoppiato ad ogni tentativo di ritrasmissione fino al valore *CWmax* (è un algoritmo *BEB* cioè esponenziale). Una stazione che riceve correttamente un *data frame*, spedisce immediatamente alla sorgente un pacchetto di acknowledgement (**ACK**) dopo un intervallo pari a *Short InterFrame Space* (***SIFS***), il quale è minore del *DISF*, per dare maggiore priorità alla stazione che ha ricevuto il dato rispetto a quelle che devono spedire (figura 7).

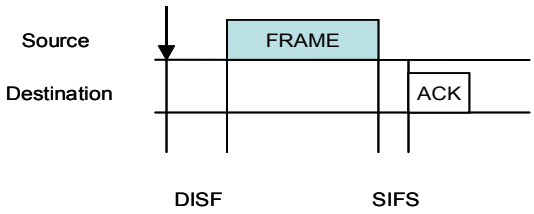

Figura 7 – Interazione tra sorgente e destinazione. Il SISF è più corto del DISF.

E' bene notare che con il CSMA/CD, una stazione non ha la capacità di rilevare una collisione mentre sta spedendo un pacchetto. Se la sorgente del pacchetto non riceve un acknowledgement, il *data frame* viene considerato perso e viene schedulata una ritrasmissione. Al fine di scoprire gli errori di trasmissione è utilizzato l'algoritmo di *Cyclic Redundancy Check* (***CRC***). L'ACK non viene trasmesso se il pacchetto ricevuto risulta essere corrotto. Dopo che la sorgente scopre che il pacchetto è andato perso (a causa di errori di trasmissione o di collisione), deve rimanere in attesa per un intervallo pari a *Extended InterFrame Space* (***EIFS***) prima di riattivare l'algoritmo di backoff.

In reti wireless (specialmente se hanno topologie Ad-Hoc) che adottano lo standard 802.11, ci sono due problemi che possono venire fuori con un protocollo di tipo CSMA/CD: *exposed node problem* e *hidden node problem*.

Il problema dell'*exposed node* è mostrato in figura 8.

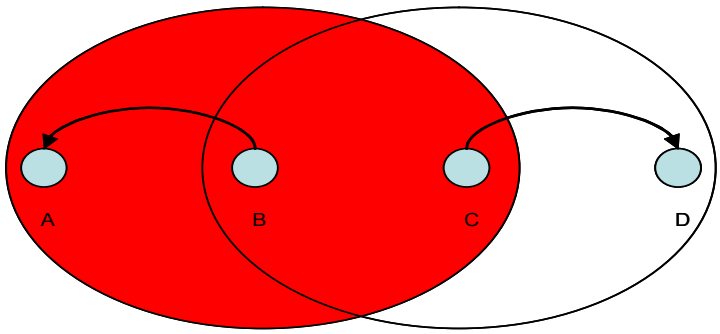

Figura 8 – Problema dell'exposed node.

Supponiamo che la stazione B stia trasmettendo dei dati alla stazione A e la stazione C sia intenzionata a trasmettere dei dati alla stazione D. C, prima di trasmettere, testa il canale, lo trova occupato (si trova nel raggio d'azione di B) e quindi deferisce la sua trasmissione ad un momento successivo. Quindi, la trasmissione tra C e D non viene effettuata, anche se non interferisce con la trasmissione in corso tra B e A (A non si trova nel raggio d'interferenza di C). Il problema dell'*exposed node* porta quindi ad una diminuzione del throughput.
Il problema dell'*hidden node* è mostrato in figura 9.

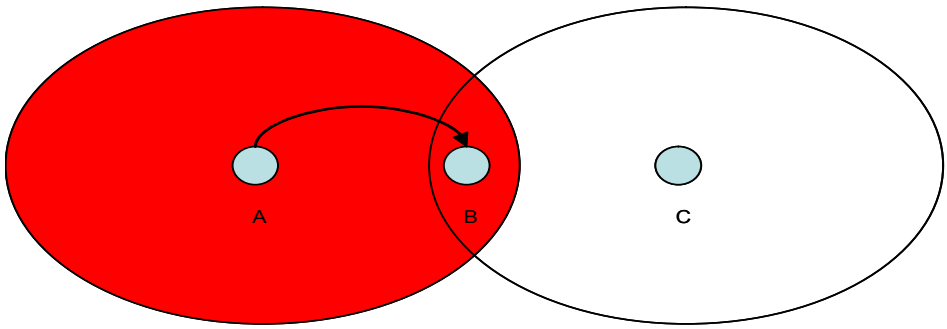

Figura 9 – Problema dell'hidden node.

Il nodo A testa il canale, lo trova libero e comincia a trasmettere dei dati al nodo B. A questo punto, il nodo C, che ha dei dati da spedire a B, testa il canale, lo trova libero (è fuori dal raggio di azione di A) e comincia a spedire il suo dato ad B. Su B arrivano due pacchetti da nodi differenti e quindi si ha collisione.
Da questi due esempi si vede che il CSMA/CD non funziona correttamente in reti *wireless Ad-Hoc*. Per alleviare il problema dell'*hidden node* il meccanismo di CSMA/CD è stato esteso ed è diventato il *virtual carrier sensing mechanism*. Il suo funzionamento si basa sull'uso di due *frames* di controllo: *Request To Send* (***RTS***) e *Clear To Send* (***CTS***). Il Sender, prima di trasmettere un dato, spedisce il *frame RTS* alla stazione di destinazione per informarla dell'inizio della trasmissione dati. Il Receiver, una volta ricevuto il *frame RTS*, replica spedendo al Sender il *CTS frame*, indicandogli che è disponibile a ricevere il dato. Sia il *RTS* che il

CTS frame contengono l'informazione sulla durata totale della trasmissione (il tempo necessario per spedire il pacchetto più quello per ricevere l'ACK). Quest'informazione, viene usata da tutte le stazioni che si trovano nel raggio di trasmissione del Sender e del Receiver per settare un timer chiamato *Network allocation vector* (***NAV***). Con il meccanismo di RTS/CTS, un nodo è abilitato a trasmettere un pacchetto solo quando il valore del suo NAV raggiunge lo zero. La figura 10 schematizza tale funzionamento:

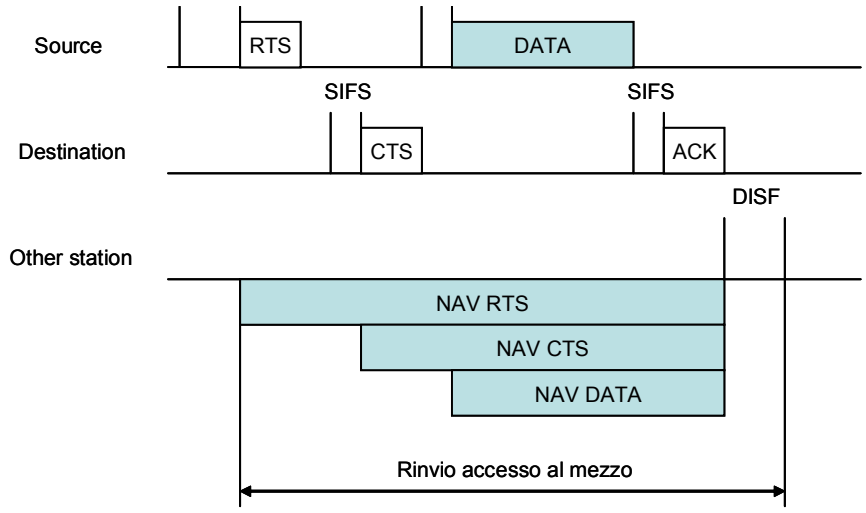

Figura 10 - Virtual Carrier Sensing Mechanism.

Con questo meccanismo si riesce ad alleviare l'*hidden node problem*. Riesaminiamo l'esempio di figura 9: la stazione C, anche se non si trova nel raggio di trasmissione di A, e quindi non sente la sua trasmissione, riesce ad ascoltare il *CTS* proveniente da B e destinato ad A. Su ricezione del *CTS*, C setta il proprio *NAV* e non accede al canale fino al raggiungimento a zero del timer (anche se il canale è sentito libero). La collisione viene così evitata.

Con il meccanismo di RTS/CTS, tutte le stazioni che si trovano ad un hop dal Sender e dal Receiver riescono ad avere informazioni sufficienti ad evitare possibili collisioni.

Nonostante l'introduzione del meccanismo RTS/CTS, il protocollo MAC esaminato continua a soffrire il problema dell'*exposed node* e non risolve completamente neanche quello dell'*hidden node*. Mettiamoci in una situazione di figura 11:

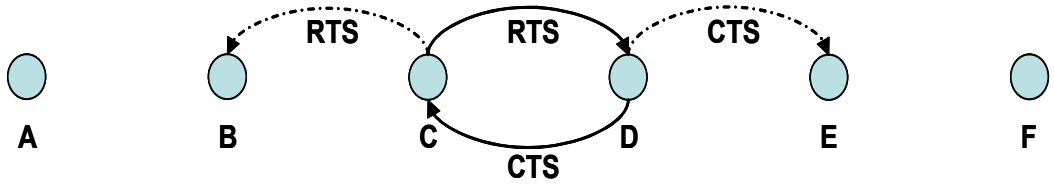

Figura 11 – Hidden and exposed node problem in presenza del meccanismo di RTS/CTS.

Il nodo C testa il canale, lo trova libero ed invia un *RTS* a D (viene ricevuto anche da B dato che si trova nel suo raggio di azione). Il nodo D, a sua volta, testa il canale, lo trova libero, e spedisce a C il *CTS* (viene ricevuto anche da E). In tale situazione si ha il comportamento seguente:

- Il nodo B riceve l'*RTS* di C e quindi setta il *NAV*. Il nodo A non essendo nel raggio di trasmissione di C (non sente la trasmissione e non riceve l'*RTS*) e di D (non riceve da questo il *CTS*), non è informato in alcun modo della trasmissione corrente e trasmette l'*RTS* a B. Sul nodo B occorre una collisione tra il RTS di A e la trasmissione di C. Quindi B è inabilitato a rispondere ad A con il suo *CTS*. Anche se non occorresse la collisione, B non potrebbe rispondere in ogni caso ad A, dato che ha il NAV settato. A, non ricevendo il *CTS* di risposta, esegue l'algoritmo di backoff e poi riprova a trasmettere. Se dopo sette tentativi non riceve nessuna risposta, scarta il pacchetto ed inoltra una informazione di *link-failures* al livello superiore.
- Il nodo F subisce la stessa sorte di A. Non essendo informato in alcun modo della trasmissione tra C e D spedisce l'*RTS* ad E, non ricevendo nessuna risposta (E ha il *NAV* settato). F, non ricevendo il *CTS*, esegue l'algoritmo di backoff e poi riprova a trasmettere. Se dopo sette tentativi non riceve nessuna risposta, scarta il pacchetto ed inoltra una informazione di *link-failures* al livello superiore. Il nodo F non può trasmettere un dato al nodo E anche se non interferisce con la trasmissione tra C e D.
- Il nodo B, pur non andando in collisione con il dato che riceve D, non può spedire nessun dato ad A, avendo il NAV settato.

Da quest'esempio, si vede che il meccanismo di RTS/CTS non risolve il problema dell'*hidden node*. Tale handshake funziona correttamente solo sotto l'ipotesi che ogni stazione che può interferire con la ricezione di un pacchetto che va da C a D, sia dentro il Sensing Range di C o il Transmission Range di D. Sotto tale ipotesi, tutti i nodi della rete sono informati della trasmissione in corso e quindi si comportano in modo corretto. Chiaramente, in reti multi-hop tale ipotesi non è verificata, ed il problema dell'*hidden node* si ripresenta. Inoltre, l'*exposed node* problem, non essendo preso in considerazione nello standard, continua ad esistere indisturbato.

A peggiorare il tutto c'è il problema che lo standard 802.11 MAC è basato sul meccanismo del *carrier sensing*, il quale, per rispettare le proprietà fisiche del canale, è progettato in modo da avere tre diversi *radio range*:

- *Transmission range* (**TX_range**) - Raggio entro il quale un pacchetto che non subisce interferenze è ricevuto correttamente. Dipende principalmente dalla potenza di trasmissione e dalle proprietà del mezzo di comunicazione. Una stazione A riceve correttamente un pacchetto da B se è dentro il TX_range di B.
- *Carrier sensing range* (**CS_range**) - Raggio entro il quale può essere sentita una trasmissione che può entrare in collisione. Una stazione può sentire le trasmissioni di tutte le stazioni che si trovano dentro il suo CS_range.
- *Interference range* (**IF_range**) - Raggio entro il quale una stazione che sta trasmettendo può interferire con la trasmissione che sta avvenendo tra una coppia di nodi, causando collisione. Il raggio d'interferenza dipende dalla distanza tra la coppia di nodi che sta comunicando e risulta massimo (IF_range-max) quando le due stazioni si trovano ad una distanza pari al TX_range. In altre parole, quando la coppia di nodi che sta comunicando si trova ad una distanza di TX_range, un nodo che vuole eseguire una trasmissione senza causare collisione presso il Receiver della coppia, deve trovarsi ad una distanza maggiore di IF_range-max da tale nodo.

Questi tre range sono legati insieme dalla seguente relazione:

$$TX\_range \leq IF\_range \leq CS\_range$$

Solitamente il CS_range e l'IF_range-max sono circa il doppio del TX_range. Questo porta ad un'amplificazione dei problemi visti: l'aumento dell'IF_range rispetto al TX_range peggiora l'*hidden node problem* mentre l'aumento del CS_range peggiora l'*exposed node problem*. Prendiamo in esame lo scenario di figura 12:

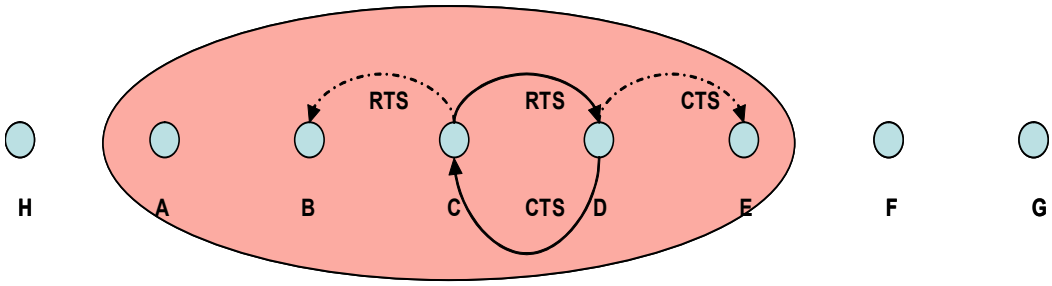

Figura 12 – L'aumento del CS-range e del IF-range peggiorano i problemi dell'hidden and exposed node.

A differenza di quanto accadeva in precedenza si può osservare che:

- A non può spedire nemmeno ad H (è nel CS_range di C).
- Se H spedisce ad A non riceve risposta, in quanto A si trova nell'IF_range di C e quindi si ha collisione. Anche in assenza di collisione, poiché A si trova nel CS_range di C, è inabilitato a spedire il CTS di risposta ad H.
- Se F spedisce un *RTS* ad E, si ha collisione (E si trova nell'IF_range di C) ed E non può replicare con il *CTS*. Inoltre si può andare in collisione anche con il dato che C sta spedendo a D (D è dentro il raggio d'interferenza di F).
- Se F spedisce un RTS a G, questo può replicare correttamente con il CTS. Però la trasmissione tra F ed G entra in collisione con il dato che sta ricevendo D (D è dentro il raggio d'interferenza di F).

I problemi dell'*exposed and hidden node*, insieme all'algoritmo di backoff del livello MAC ed all'interazione con il meccanismo di controllo della congestione del TCP, fanno sì che le performance del TCP in reti Ad-Hoc siano scarse. Analizziamo i tre problemi citati in precedenza facendo riferimento a [12], dove vengono effettuati una serie di test su una rete con topologia a stringa formata da nodi posti ad una distanza di 200 metri l'uno dall'altro e dove tra i vari raggi di trasmissione esiste la seguente relazione:

$$IF\_range = CS\_range = 2 \times TX\_range$$
$$TX\_range = 250 metri$$

Instability problem. In [12] viene instaurata una singola connessione TCP tra il nodo 1 e il nodo 5 e ne viene studiato il comportamento in termini di throughput. Il TCP, non essendoci nessun traffico di sottofondo nella rete, dovrebbe stabilizzare il trhoughput della connessione ad un certo valore, in funzione della capacità del mezzo trasmissivo. Invece, le simulazioni effettuate mostrano che il throughput della connessione TCP viene ad avere un andamento altalenante (avvicinandosi più volte allo zero). Cerchiamo di capire il perché di questo comportamento analizzando la figura 13.

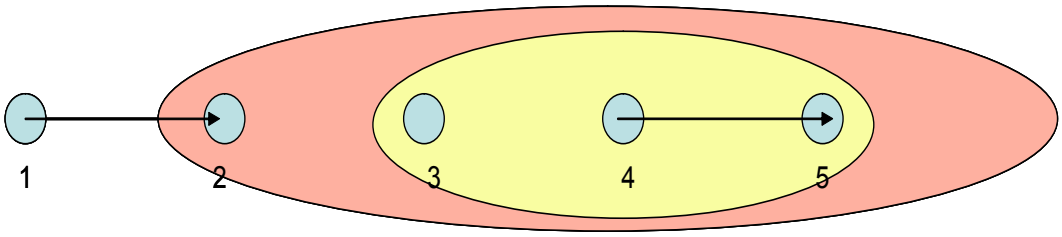

Figura 13 – Instability problem.

Supponiamo che ad un certo istante, il nodo 4, che ha dei pacchetti da spedire al nodo 5, riesca ad accedere al canale per effettuare la spedizione. Il nodo 4 invia

l'*RTS* a 5, il quale replica con il *CTS*. A questo punto, il nodo 4 comincia a spedire il pacchetto dati verso il nodo 5. Inoltre, supponiamo che durante la trasmissione tra 4 e 5, il nodo 1 voglia trasmettere un dato al nodo 2. Il nodo 1 testa il canale, lo sente libero, e spedisce l'*RTS* al nodo 2. Il *RTS* può subire collisione sul nodo 2, poiché si trova nel raggio d'interferenza del nodo 4 (*hidden node problem*), o può essere ricevuto correttamente. In questo caso, il nodo 2 testa il canale, lo sente occupato e non risponde al nodo 1 con il *CTS* (*exposed node problem*). La trasmissione da 1 a 2 non avviene, anche se non interferisce con quella tra 4 e 5. Il nodo 1, non ricevendo il *CTS* di risposta, deferisce la trasmissione in base all'algoritmo di backoff. L'algoritmo di backoff, insieme alla lunghezza del pacchetto dati che va da 4 a 5, favorisce il nodo 4 nella contesa per accedere al canale, in quanto sul nodo 1, ad ogni tentativo di ritrasmissione, la CWmax viene raddoppiata, diminuendo le chance del nodo 1 di vincere la contesa. Per quanto detto, se il nodo 4 ha molti pacchetti TCP da spedire, il canale viene tenuto occupato per molto tempo, causando al nodo 1 numerosi tentativi falliti di ricevere il *CTS* dal nodo 2. Dopo 7 tentativi, il nodo 1 riporta un *link-breakage* al livello superiore (si ha una *route-failures*) e comincia la ricerca di un nuovo path. Fino a quando la strada per andare a destinazione non è disponibile, non vengono trasmessi sul canale altri pacchetti da parte della sorgente, causando l'avvicinarsi a zero del throughput. Solitamente, il tempo per recuperare da una *route-failures* supera 1 sec. e sul nodo 1 possono presentarsi i problemi esaminati in precedenza causati dalla mobilità. Preso il TCP possono scattare i timeout per i pacchetti spediti e non acknoleggiati, facendo così partire il meccanismo di controllo della congestione: la finestra del TCP riparte dalla dimensione di 1 pacchetto e vengono ritrasmessi i pacchetti unacked con conseguente diminuzione del goodput. Tutto questo porta all'andamento altalenante del throughput. A peggiorare le cose c'è il fatto che il TCP non solo trasmette dati da 1 a 5, ma trasmette anche gli ACK di risposta che competono anche essi al canale. Tale competizione, essendo i pacchetti dati di dimensioni maggiori dei pacchetti di ack, viene vinta con maggiore probabilità dai pacchetti dati, causando una maggiore perdita di ack. Questo può portare il TCP a

considerare persi pacchetti che sono arrivati correttamente a destinazione e ad invocare il meccanismo di controllo della congestione. Il crescere del RTO ad ogni tentativo di ritrasmissione, degrada ancora di più il throughput.

Guardiamo che problemi possono dare gli ACK con un esempio. Supponiamo che il nodo 2 abbia un ACK da spedire al nodo 1. Essendo nella condizione di *exposed node* (sente la trasmissione del nodo 4), il nodo 2 non può trasmettere l'ACK fino a quando il canale risulta occupato. Se tale tempo è elevato può scattare il timeout relativo a tale pacchetto, scatenando la procedura di controllo della congestione.

In [9], [12] e [16], viene mostrato come la dimensione della finestra di controllo della congestione influenzi le performance del TCP. Se la dimensione della finestra di controllo viene tenuta bassa, il numero di pacchetti in volo nella connessione è basso, quindi il numero di pacchetti *back-to-back* che un nodo deve trasmettere al proprio vicino è basso, aumentando le chance dei vari nodi di accedere al canale in 7 tentativi. In tali studi, viene mostrato come una finestra di circa 4 pacchetti permetta di risolvere il fenomeno dell'instabilità e di raggiungere la massima utilizzazione del canale.

Incompatibily problem. In [12] si osserva che, con il livello MAC dato dallo standard 802.11, ci sono situazioni in cui due connessioni TCP non possono coesistere nella rete allo stesso tempo. Quando è attiva una connessione (spedisce pacchetti), l'altra non riesce ad accedere al canale. Poi ad intervalli di tempo random il ruolo delle connessioni si inverte (quella attiva si spenge e quella spenta diventa attiva). Per mostrare tale fenomeno, vengono utilizzate connessioni TCP con una dimensione massima della finestra di controllo pari ad 1 pacchetto, per evidenziare l'indipendenza di questo fenomeno dalla dimensione della finestra. Prendiamo la configurazione di rete di figura 14, dove vengono instaurate due connessioni TCP: una tra il nodo 3 d il nodo 1, ed una tra il nodo 4 ed il nodo 5.

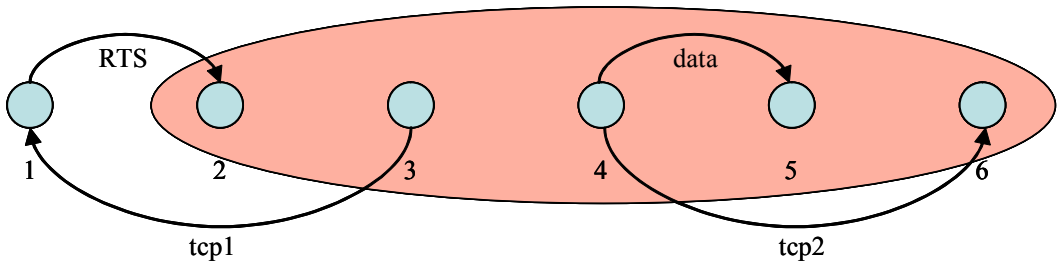

Figura 14 - Incompatibily problem.

Supponiamo che il nodo 4 stia spedendo un pacchetto al nodo 5, e che il nodo 1 voglia spedire un TCP-ack al nodo 3. Il nodo 1 testa il canale, lo sente libero e spedisce l'*RTS* a 2, non ricevendo nessuna risposta (il nodo 2 è nell'IF_range e nel CS_range del nodo 4). Il nodo 1, non ricevendo il *CTS* di risposta, deferisce la trasmissione in base all'algoritmo di backoff. L'algoritmo di backoff, insieme alla differente lunghezza del pacchetto dati che va da 4 a 5 rispetto al pacchetto di ack, fa si che il nodo 1 possa fallire 7 volte nel tentativo di raggiungere il nodo 2, causando una *link-failures* e l'esecuzione della routine di *route-discovery* da parte del protocollo di routing. La connessione 1 raggiunge un throughput pari a 0 mentre la 2 funziona correttamente. La situazione si ribalta non appena, per esempio, è il nodo 3 che riesce ad accedere al canale. Si ha la stessa situazione di prima però ribaltata. Lo scattare dei timeout e l'invocazione del meccanismo di controllo della congestione presso il TCP (con conseguente crescita del RTO), fa sì che la connessione off riesca ad acquisire il canale solo dopo un certo numero di tentativi.

Unfairness problem. In una rete Ad-Hoc in cui sono presenti più connessioni TCP, si può andare incontro al problema dell'unfairness. Le varie connessioni TCP possono non condividere in maniera equa la capacità del canale. Al limite, alcune connessioni possono essere completamente prive di traffico (cattura del canale da parte d'altre connessioni). Questo accade anche se la dimensione massima della finestra di controllo del TCP è di un pacchetto. In [12] viene fatto l'esempio del *one hop unfairness problem*. Esaminiamolo facendo riferimento alla Figura 15.

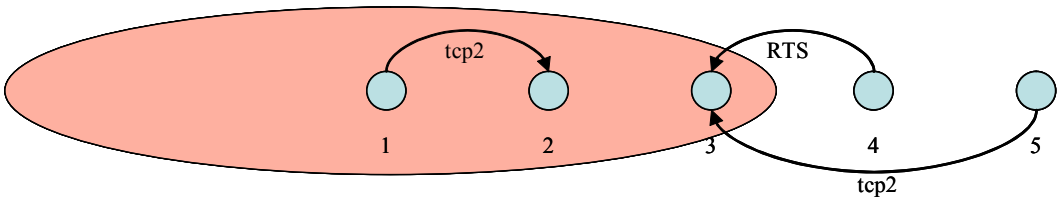

Figura 15 - one hop unfairness problem.

Inizialmente viene settata una connessione TCP tra il nodo 5 ed il nodo 3. Il throughput di questa connessione si stabilizza ad un certo valore. In seguito viene settata una connessione tra il nodo 1 ed il nodo 2. Non appena i pacchetti della connessione tra 1 e 2 riescono ad accedere al canale, succedono le seguenti cose: il nodo 4 riceve da 5 il pacchetto dati, testa il canale, lo trova libero e spedisce l'*RTS* al nodo 3. Il *RTS* del nodo 4, essendo il nodo 3 nel raggio di interferenza di 1, può subire collisione sul nodo 3 (*hidden node problem*). In questo caso il nodo 3 non può rispondere con il *CTS* al 4. Anche nel caso in cui il nodo 3 ricevesse correttamente il *RTS* di 4, non potrebbe rispondere con il *CTS*, poiché si trova nel IF_range di 2 (*exposed node problem*) e riceve il *CTS* dal nodo 2. Il nodo 4, dopo 7 tentativi falliti di ricevere il *CTS* dal nodo 3, riporta un *link-breakage* al livello superiore, scatenando l'invio di un messaggio di *route-failures* al nodo 5. Il nodo 5, a questo punto, inizia la routine di *route-discovery*, e il throughput della connessione, non venendo spediti pacchetti, arriva a zero. Il nodo 4 può trasmettere con successo al nodo 3 solo quando il nodo 2 non sta trasmettendo dati al nodo 1 e prima che il nodo 1 trasmetta il *RTS* al 2. Inoltre, il nodo 1 non appena riceve il TCP-ack dal nodo 2, si prepara subito a trasmettere un altro segmento TCP. In altre parole, il nodo 4 ha poche opportunità di accedere al canale. In più l'algoritmo di backoff del livello MAC, essendo esponenziale, riduce ulteriormente l'opportunità del nodo 4 di vincere la contesa (gli intervalli di tempo a cui si spediscono gli *RTS* aumentano in modo esponenziale). Di conseguenza il throughput della prima connessione raggiunge lo zero.

Un altro studio, fatto in [8], prende in considerazione una configurazione di rete come quella riportata in figura 16, e analizza il problema dell'*unfairness* e della *capture*.

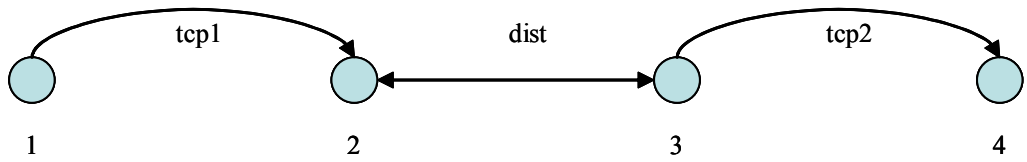

Figura 16 – Unfairness and capture problem.

In tale studio, vengono prese due connessione di 1 hop ciascuna (la distanza tra 1 e 2 e tra 3 e 4 è di 300 metri), e viene fatta variare la distanza tra le due connessioni (tra i nodi 2 e 3). I radio range utilizzati nella simulazione hanno i seguenti valori: TX_range = 376, IF_range = 533 e CS_range = 670. Il comportamento al variare della distanza è il seguente:

- dist < 76 - ogni nodo è nel CS_range di ciascun altro e il protocollo di livello MAC si comporta bene.
- 76 < dist < 233 - il nodo 1 è fuori dal TX_range del nodo 3 e il nodo 4 è fuori dal TX_range del nodo 2. Questi nodi non sono in grado di conoscere lo stato della rete. Fortunatamente, il nodo 1 è nell'IF_range di 3 e il nodo 4 è nell'IF_range di 2. Di conseguenza, le due connessioni interferiscono l'un l'altra, impedendo la cattura del canale da parte di una della due. Più precisamente, la trasmissione di un ack da parte del nodo 4 può causare collisione sul nodo 2 e la trasmissione di un pacchetto da parte del nodo 1 può causare collisione sul nodo 3.
- 233 < dist < 370 - la trasmissione da parte di 1 (4) non interferisce con la connessione 2 (1). Qui dominano i fenomeni dell'*exposed* e *hidden node*. Quando il nodo 4 spedisce un RTS a 3, se il nodo 1 sta spedendo un dato a 2, il nodo 3 non può replicare con il CTS (è nel TX_range di 2) causando backoff sul nodo 4 (a causa del protocollo di backoff l'intervallo può di-

ventare grande limitando le chance di 4 di acquisire il canale). Lo stesso discorso vale se il nodo 1 vuole spedire quando il nodo 4 sta spedendo a 3. Però, i pacchetti che vanno dal nodo 1 al nodo 2 sono TCP data packet (di grandi dimensioni) mentre i pacchetti che vanno da 4 a 3 sono TCP ack packet (di piccole dimensioni). Quindi la connessione 2 risulta svantaggiata rispetto alla connessione 1. Se sono i nodi 2 e 3 a spedire non ci sono problemi (gli altri nodi ricadono nel loro CS_range).

- 376 < dist <533 - il nodo 3 (2) è fuori dal TX_range del nodo 2 (3) e quindi, se il CS_range ed l'IF_range fossero uguali al TX_range, le due connessioni sarebbero indipendenti. Questo però non accade, poiché la maggiore dimensione del IF_range causa la cattura del canale. La trasmissione del nodo 3 (2), interferisce con ogni pacchetto ricevuto dal nodo 2(3). Dato che i pacchetti ricevuti sul nodo 2 sono TCP data packet e i pacchetti ricevuti dal nodo 3 sono TCP ack data, l'interferenza ha effetti peggiori sulla connessione 1. Inoltre, quando il nodo 1 spedisce un RTS al nodo 2, se il nodo 3 sta trasmettendo al nodo 4, il nodo 2 non riceve correttamente il RTS e non risponde con il CTS. La connessione 1 risulta molto penalizzata ed il suo throughput si avvicina a zero.
- 533 < dist <670 - le due connessioni sono indipendenti e non interferiscono l'un l'altra. Nonostante questo, il nodo 2 e 3 sono uno nel CS_range dell'altro. Quindi i problemi dell'*hidden* e *exposed node* continuano ad esistere. Quando è il nodo 3 a spedire, il 2 non può rispondere al nodo 1, il quale eseguirà l'algoritmo di backoff. Lo stesso problema si ha quando è il 2 a spedire. In ogni modo, il 3 spedisce un TCP-data mentre il 2 spedisce un TCP-ack (di dimensioni molto minori) e quindi la connessione 3-4 prende la frazione maggiore del canale.
- Dist > 670 - Le due connessioni sono indipendenti.

Cerchiamo di riassumere i vari problemi dello standard 802.11:

- La presenza dell'*hidden node problem* fa sì che un nodo possa fallire di raggiungere un proprio vicino ed invochi l'algoritmo di backoff, eventualmente scartando il pacchetto dopo 7 tentativi di ritrasmissione.
- La presenza dell'*exposed node problem* fa sì che un nodo possa ritardare in maniera eccessiva il suo accesso al canale.
- La differenza tra TX_range e IF_range peggiora la situazione amplificando i problemi dell'*exposed e hidden node*.
- L'algoritmo di backoff dello standard 802.11 non tiene in considerazione le condizioni della rete e forza eccessivi e non necessari ritardi nel tentativo d'accesso al canale, aumentando le chance del nodo che per ultimo ha eseguito una trasmissione con successo.

Gli esempi analizzati fino ad ora fanno inoltre notare come il TCP amplifichi i problemi del livello MAC a causa dell'interazione tra il suo meccanismo di controllo della congestione e i problemi presenti al livello MAC. In altre parole, l'invocazione del meccanismo di controllo della congestione, con conseguente aumento del RTO, porta un ulteriore svantaggio alla connessione che ha subito la perdita di pacchetti (causata dalla contesa per accedere al canale). Di conseguenza, per migliorare il comportamento del TCP in tali situazioni, si dovrebbe metterlo nella condizione di distinguere tra pacchetti persi a causa della contesa e pacchetti persi a causa della congestione, in maniera da invocare il meccanismo di controllo della congestione solo quando è necessario.

Analizziamo in breve alcune soluzioni che sono state proposte in letteratura per mitigare i problemi che si hanno al livello MAC.

In [8] viene studiato il problema dell'*unfairness* e vengono proposte delle modifiche da apportare al livello MAC dello standard IEEE 802.11. Il problema principale dello standard IEEE 802.11 in reti MANET, è che ai nodi della rete mancano le informazioni sulle attività dei propri vicini e non riescono a coordinare tra di loro le trasmissioni e ritrasmissioni. Quindi, il protocollo di backoff, non avendo a disposizione le informazioni che gli permettono di calcolare quanti nodi stanno

cercando di accedere al canale, è stato implementato secondo uno schema BEB. E' proprio tale protocollo ad essere messo sotto accusa come causa principale dei problemi di *unfairness* e viene proposto un nuovo algoritmo di backoff: l'*Active Neighbor Estimation Based Backoff* (***ANE***). L'*ANE* è basato sulle seguenti considerazioni: ogni nodo, attraverso i pacchetti di *RTS* e *data* che riceve, conta il numero di vicini che hanno pacchetti da spedire. I pacchetti MAC vengono inoltre modificati per contenere la stima del numero di vicini attivi di chi spedisce il pacchetto. Così facendo, ogni nodo conosce anche la stima fatta dai suoi vicini. Facendo l'assunzione che i nodi che competono al canale sono i nodi che si trovano ad un hop dal trasmettitore o dal ricevitore, una stima dei nodi che competono al canale (N) può essere fatta utilizzando la seguente relazione:

$$MAX(Nt, Nr) \leq N \leq (Nt + Nr)$$

dove la funzione MAX torna il massimo tra Nt e Nr e dove con Nr si indica la stima del numero di vicini attivi del ricevitore e con Nt si indica la stima del numero di vicini attivi del trasmettitore. Con queste considerazioni la finestra dell'algoritmo di backoff può essere calcolata con la seguente formula:

$$CW = a \times CW\min + F(N)$$

Dove per esempio si può prendere $F(N) = a \times CW\min \times N$, che significa che una stazione ha una probabilità su N di accedere al canale, con N numero di contendenti. Esaminiamo l'esempio di figura 17:

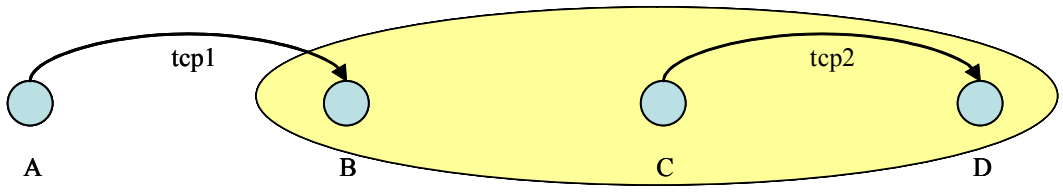

Figura 17 - Active Neighbor Estimation Based Backoff.

A impara da B che C è attivo e ha dati da spedire sul canale e, dopo ogni fallimento ad accedere al canale, la sua finestra per il calcolo del backoff time diventa aCWmin+1 e non cresce a dismisura come faceva prima. Così facendo sono aumentate le possibilità di A di vincere il contenzioso con C.

In [9] si cerca di migliorare le performance del TCP definendo la condizione di *network overload* (stato della rete in cui s'impone un traffico maggiore di quello che permette di dare la massima utilizzazione del canale) e stimando la dimensione ottimale per la finestra di controllo. Viene seguito il seguente approccio: supponendo che la trasmissione sui vari nodi possa essere fatta secondo una schedulazione da noi imposta, si cercano quali sono i nodi che possono spedire contemporaneamente senza provocare problemi. Ad esempio, in una configurazione a stringa come quella di figura 18, possono spedire contemporaneamente solo il nodo A ed il nodo E.

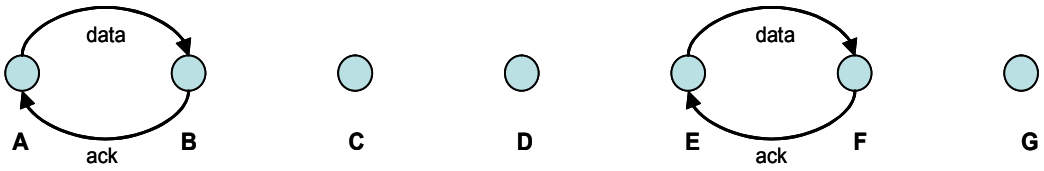

Figura 18 – Stima della dimensione ottimale per la finestra di controllo del TCP.

Quindi, in tale configurazione, la finestra ottimale per una connessione da A a G è di circa 2 pacchetti. Seguendo questo ragionamento, viene stimata la dimensione ottimale per la finestra delle connessioni TCP, con riferimento a varie topologie di rete ed al variare del numero di connessioni attive. Oltre alla stima della dimensione della finestra di controllo, in [9] vengono messi in evidenza i seguenti aspetti del comportamento del TCP in reti ad hoc:

- Mancata stabilizzazione della dimensione della finestra di controllo al valore ottimo trovato.
- Assenza di perdita di pacchetti dovuta ad un sovraccarico dei buffer (congestione) e perdita dei pacchetti dovuta alla contesa per accedere al canale (per dimensioni adeguate dei buffer sui vari nodi).

In tale articolo sono stati proposti due algoritmi (**LRED**, **Adaptive Pacing**) per migliorare il comportamento del TCP in reti Ad-Hoc. Il principio su cui si basa l'algoritmo *LRED* è quello di usare i primi segni della contesa sui vari nodi come indicazione di *network overload* e di far capire al TCP per tempo che il carico di lavoro che sta mandando in rete è eccessivo (porta ad overload). Per fare questo, come indicazione di *network overload* viene usato il numero di tentativi di ritrasmissione dei vari nodi. Più precisamente, appena il numero di tentativi di ritrasmissione su un certo nodo supera una certa soglia (il nodo è spesso *hidden*), viene informato di questo il TCP attraverso una notificazione esplicita. Quindi il TCP viene messo nella condizione di poter stimare lo stato effettivo della rete e di distinguere tra pacchetti persi a causa della congestione e pacchetti persi a causa del contesa al canale, riuscendo così ad adattare il traffico offerto alla effettiva capacità della rete. Tutto questo porta ad un aumento del throughput ed alla stabilizzazione della finestra di controllo. L'algoritmo di *Adaptive Pacing* ha come fine quello di migliorare l'utilizzazione del canale e di raggiungerne l'equa condivisione tra le varie connessioni. L'idea di fondo di tale algoritmo è quella di aggiungere al tempo di backoff calcolato da un nodo, dopo una trasmissione avvenuta con successo, un tempo extra pari al tempo di trasmissione di un pacchetto. Ad esempio, in relazione al LRED, quando un nodo ha trasmesso un pacchetto ed ha il numero di tentativi di ritrasmissione inferiore alla soglia, calcola il tempo backoff e ci aggiunge il tempo extra. Cosi facendo si migliora la coordinazione tra i nodi, permettendo ad un nodo che sta cercando di accedere al canale di avere più chance. In pratica si penalizzano quei nodi che non hanno problemi di contesa. Con questi due algoritmi, il TCP riesce a stabilizzare la dimensione della finestra al valore ottimo e a far condividere il canale in maniera equa tra le varie connessioni.

2.2 Soluzioni proposte in letteratura

In questo capitolo sono esaminate le varie proposte che si trovano in letteratura il cui fine è quello di migliorare il comportamento del TCP in ambiente MANET.

2.2.1 TCP-F

In [1] viene proposta una modifica del TCP basata sull'uso di un feedback esplicito da parte del livello network. Tale feedback ha il compito di informare il TCP sulla presenza di una strada che porta a destinazione. Il *TPC-F* si comporta nella seguente maniera: supponiamo che si stiano trasferendo dei pacchetti tra sorgente e destinazione. Appena il livello MAC di un nodo del percorso si accorge di una *link-failures*, invia questa informazione al livello network, il quale invia un pacchetto di *route-failures notification* (*RFN*) alla sorgente. Tutti i nodi intermedi che ricevono il pacchetto di *RFN,* prendono atto che quella particolare strada non è più buona e, se conoscono una strada alternativa per andare a destinazione, scartano il *RFN* ed usano il nuovo path per recapitare i pacchetti. Se tali nodi non conoscono nessuna strada alternativa, propagano il *RFN* verso la sorgente. La sorgente TCP si può trovare in due stati distinti: *snooze* ed *established*. Su ricezione del *RFN*, la sorgente si porta nello stato *snooze* e compie le seguenti azioni:

- Smette di spedire pacchetti.
- Congela tutte le sue variabili, come i timers e la dimensione della finestra.
- Fa partire un *route failure timer* che scatta se una strada non viene trovata entro un certo periodo di tempo.

Con questo meccanismo si evita che siano spediti pacchetti verso la destinazione quando la strada non è disponibile e che aumenti troppo il RTO a causa di successive ritrasmissioni. La sorgente rimane nello stato *snooze* finché non riceve la notifica del ristabilimento della strada, attraverso il pacchetto di *route re-estabilishment notification* (*RRN*). Il pacchetto di *RRN* viene spedito, dai nodi che in precedenza hanno spedito il *RFN*, non appena trovano una nuova strada per andare a destinazione. Su ricezione del *RRN* la sorgente scongela i timers, rispedisce tutti i pacchetti che non hanno ancora ricevuto un ack dalla destinazione ed entra nello stato di *established*. La trasmissione dei dati riprende allo stesso rate che aveva prima del cambiamento della strada. Dalle simulazioni effettuate viene mostrato che, per bassi valori del *route re-establishment delay* (*RRD*), e cioè del tempo impiegato per ritrovare una strada, TCP e TCP-F si comportano in modo

simile. Nel caso del TCP, essendo il *RRD* basso, scattano pochi timeout, il RTO non cresce a dismisura ed una nuova strada può essere sfruttata senza eccessivi ritardi dovuti alle dimensioni del RTO. Al crescere del *RRD* il TCP-F si comporta sempre meglio. L'allungarsi del *RRD* porta il TCP ad avere numerosi timeout e alla conseguente crescita del RTO (ad ogni ritrasmissione viene raddoppiato), rendendo il TCP incapace di sfruttare una strada appena viene ritrovata. Di converso, il TCP-F, entrando nello stato *snooze*, non ha quest'incremento del RTO e non appena la strada viene ritrovata e la sorgente riceve il messaggio di *RRN*, la trasmissione viene ripresa da dove era rimasta. Seguendo lo schema del TCP-F si ha un aumento del throughput e una riduzione del numero di ritrasmissioni. All'incrementare del *data rate* della trasmissione, le prestazioni del TCP-F rispetto al TCP migliorano sempre di più.

2.2.2 ENIC

L'ENIC (*explicit notification with enhanced inter-layer comunication and control*), descritto in [2], è una modifica del TCP che si basa sullo schema del TCP-F. Per realizzare il feedback da far pervenire alla sorgente TCP vengono sfruttate le caratteristiche del protocollo di routing *AODV*. Usano il protocollo di routing AODV si possono distinguere due tipi di *route failure*: *reparable route failure* (**RRF**) e *irreparable route failure* (**IRF**). Una *RRF* si ha quando il nodo che si accorge di una *route-failures* riesce a ritrovare una strada verso la destinazione (entro un predefinito intervallo di tempo), utilizzando i suoi nodi adiacenti. Una *IRF* si ha quando la strada non può essere recuperata dai nodi adiacenti al nodo che si è accorto della *route failure*. In questo caso la ricerca della strada viene affidata alla sorgente dei pacchetti. L'ENIC si appoggia sui seguenti segnali provenienti dal protocollo di routing: *route reply* (*RREP*) e *route error* (*RERR*). Appena un nodo si accorge di una *IRF* invia un messaggio di *RERR* alla sorgente ed alla destinazione del pacchetto che ha generato la *route-error*. La sorgente, su ricezione del *RERR*, comincia la ricerca di una nuova strada ed invia un messaggio di *explicit route error notification* (*EREN*) al livello TCP. Il comportamento del

TCP su ricezione del messaggio di *EREN* è uguale a quello del TCP-F. Non appena una strada viene ritrovata, il livello network spedisce un messaggio di *explicit route recovery notification* (*ERRN*) al TCP, il quale, su ricezione di tale messaggio, si comporta come il TCP-F con la differenza che, per la ritrasmissione dei pacchetti unacked usa un RTO modificato. La nuova strada può essere di dimensioni diverse da quella vecchia, quindi ritrasmettere i pacchetti con il vecchio valore del RTO non è una buona scelta. Per ritrasmettere tali pacchetti viene utilizzato un *TemporaryRTO* (*TRTO*) calcolato come segue:

$$TRTO = a \times oldRTO \times \frac{new\_route}{old\_route}$$

dove "a" è un coefficiente e *new_route* e *old_route* sono le lunghezze in hop della nuova e vecchia strada (ottenute dal routing protocol). Dopo che è stato ricevuto l'ack dell'ultimo pacchetto ritrasmesso, il Sender riprende il calcolo del RTO in modo normale. Il comportamento del ricevitore ENIC differisce da quello del TCP-F. Questo usa il meccanismo dei *DACK* (*delayed ack*) insieme con quello dei *SACK* (*selective ack*). Il meccanismo dei *DACK*, vale a dire lo spedire l'ack non subito dopo la ricezione del pacchetto ma dopo un predefinito intervallo di tempo, è usato per ridurre il numero di ack spediti lungo la rete. Così facendo viene ridotto il problema della contesa al canale, e viene alleviato il problema degli *out-of-order* packet. Il meccanismo dei *SACK* permette la ritrasmissione selettiva dei pacchetti. Appena il Receiver riceve un *EREN* stoppa le trasmissioni dei *DSACK*. Su ricezione di un *ERRN* il Receiver spedisce tutti i *SACK* non spediti e aspetta un nuovo pacchetto. Se il nuovo pacchetto è duplicato lo scarta, e continua a spedire i *DSACK* in modo normale.

2.2.3 TCP-BUS

In [3], il TCP viene modificato seguendo lo schema proposto col TCP-F e sfruttando la capacita dei vari nodi della rete di bufferare i pacchetti. Come protocollo

di routing viene usato il *ABR*. Tale protocollo di routing fornisce due messaggi di controllo: l'*explicit route disconnection notification* (*ERDN*) e l'*explicit route successful notifivation* (*ERSN*). Il messaggio di *ERDN* viene generato da un nodo (PN) appena si accorge di una *route-failures* e viene propagato fino alla sorgente. Il messaggio di *ERSN* viene generato dal PN non appena viene ritrovata una strada per andare a destinazione. La sorgente, su ricezione del messaggio del messaggio di *ERDN* congela il suo stato e si mette in attesa del messaggio di *ERSN*. Su ricezione del messaggio di *ERSN*, la sorgente ritrasmette solo i pacchetti persi che si trovavano nei nodi tra il PN e la destinazione (sfruttando le informazioni contenute nei messaggi di *ERSN* e *ERDN*) e raddoppia il timeout per quei pacchetti unacked e non ritrasmessi che si trovano tra la sorgente ed il PN (a causa di un aspettato incremento per il tempo di arrivo dei pacchetti causato dal cambiamento della strada). Il Receiver si comporta come quello del TCP classico, con l'aggiunta dell'opzione dei *SACK* e di una procedura che impedisca le richieste di ritrasmissione che non necessarie. Inoltre, il TCP-Bus fornisce una tecnica per la trasmissione affidabile dei messaggi di controllo.

2.2.4 TCP-ELFN

In [4] viene studiato il comportamento del TCP in reti Ad-Hoc, con riferimento al protocollo di routing *DSR*, e viene proposta una modifica da apportare al TCP al fine di metterlo nelle condizioni di rilevare le *route-failures*. Quando un nodo si accorge di una *route-failures*, spedisce il messaggio di *Explicit Link Failure Notification* (**ELFN**) verso la sorgente del pacchetto. Il messaggio di *route-failures* del *DSR* viene modificato in modo da contenere un payload che porta i campi corretti del TCP/IP header del pacchetto che porta la notizia (Sender e Receiver address e port) ed il TCP sequence number. Così facendo, il TCP, tramite il messaggio di ELFN, riesce a distinguere tra perdite di pacchetti causate dalla congestione e perdite causate dalla mobilità. Su ricezione del pacchetto di *ELFN*, il TCP disabilita i suoi timer ed entra in uno stato di *standby*. Quando si trova nello stato di *standby,* il TCP-ELFN spedisce ad intervalli regolari un pacchetto di *probe*, per vedere se è

stata ritrovata una strada che porta a destinazione. Se viene ricevuto l'ack del pacchetto di *probe*, significa che la strada è stata ripristinata e quindi lo stato di *standby* viene lasciato e la trasmissione viene ripresa da dove era rimasta. Le performance del TCP vengono confrontate in [4], con quelle del TCP-ELFN, facendone variare alcuni parametri quali la lunghezza dell'intervallo a cui vengono spediti i pacchetti di *probe*, il valore del RTO e facendo variare la dimensione della finestra di controllo della congestione. In tale studio viene evidenziato come il throughput viene a dipendere dall'intervallo tra i pacchetti di *probe*, aumentando al diminuire di questo. Intuitivamente i risultati sono giusti perché più aumenta l'intervallo più tardi viene rivelata la presenza di una strada. L'uso del TCP-ELFN con un intervallo di probe di 2s mostra un miglioramento del throughput medio dei vari modelli e migliora anche il throughput dei singoli modelli utilizzati nelle simulazioni. Per scegliere un intervallo di *probe* di dimensione giusta, può essere scelto di spedire il *probe* ad intervalli che variano in funzione del RTT. Anche il valore del RTO ha un impatto notevole sulle prestazioni del TCP-ELFN. Facendo tornare il RTO al valore di defoult, dopo la ricezione dell'ELFN, e tornando ad una dimensione della finestra di 1 pacchetto si vede che le prestazioni degradano. Invece, non toccare il RTO e portare la dimensione della finestra al valore di 1 pacchetto, ha un piccolo impatto sulle performance, in quanto la dimensione ottimale della finestra di controllo per reti Ad-Hoc è piccola e torna velocemente al valore di regime.

2.2.5 ATCP

L'approccio seguito in [11] è differente dai precedenti, poiché non richiede nessuna modifica al protocollo TCP. Per migliorare il comportamento del TCP viene implemento un livello intermedio tra network e transport layer (ATCP). L'ATCP si basa sull'uso del protocollo di *ICMP* per scoprire una partizione della rete e sul meccanismo di *Explicit Congestion Notification* (*ECN*) per rilevare la congestione. Il nuovo livello si comporta in maniera da evitare lo scattare del meccanismo

di controllo della congestione quando non è necessario. Più precisamente l'*ATCP* layer si può trovare in 4 stati:

- *Normal state*. L'*ATCP* esegue le seguenti azioni: i) conta il numero di ack duplicati ricevuti e valuta i valori dei timer dei pacchetti trasmessi dal livello TCP. Su ricezione del terzo ack duplicato o prima dello scattare del RTO mette il TCP in "*persist mode*" ed entra nel *loss state* (il TCP in questa maniera non si accorge della perdita dei pacchetti e non invoca il meccanismo di controllo della congestione); ii) Controlla il flag di *ECN* nei pacchetti di ack e nei pacchetti dati e se questo viene trovato settato, l'*ATCP* entra nel *Congested state*; iii) Su ricezione di un messaggio di *ICMP*, mette il Sender in "*persist mode*" ed entra in *disconnected state*.
- *Loss state*: l'*ATCP* ritrasmette i pacchetti unacked e mantiene i propri timer per ritrasmettere questi pacchetti. Su ricezione degli ack per i pacchetti ritrasmessi, l'*ATCP* propaga questi al TCP, che esce dal *persist mode* e ritorna in *normal state*.
- *Congested state*: l'ATCP non esegue nessuna azione e lascia che sia il livello TCP ad gestire la congestione. Dopo che il TCP trasmette un segmento l'ATCP ritorna in *normal state*.
- *Disconnected state*: il TCP in *persist mode* genera periodicamente dei pacchetti di *probe* per testare la disponibilità di una strada verso la destinazione. Su ricezione di una risposta da parte del Receiver, l'*ATCP* torna in *normal state* e il TCP lascia il *persist mode*. L'*ATCP*, inoltre setta ad uno il valore della dimensione della finestra di controllo del TCP (per venire incontro alle dimensioni variate del percorso tra la sorgente e la destinazione).

Il comportamento di base è quello del TCP-ELFN con la differenza che ora viene sfruttato anche il meccanismo di ECN per scoprire la congestione della rete.

2.2.6 Fixed RTO and out-of-order detection

In [5] e [6], per risolvere i problemi legati alla mobilità, viene seguito un approccio end-to-end, che non necessita di nessuna notifica esplicita da parte della rete. In [5], come indicazione di disconnessione viene usato lo scattare di due timer consecutivi e, per risolvere i problemi dovuti al calcolo del RTO (*exponential backoff*) usato nel TCP, viene modificato il TCP Sender in maniera da mantenere costante il valore del RTO allo scattare del secondo timeuot consecutivo (e successivi). In questa maniera, si viene ad avere un comportamento simile a quello del TCP-ELFN: la presenza di una strada per andare a destinazione viene testata ad intervalli regolari pari al RTO (dove RTO non cresce in maniera esponenziale ma rimane costante). Inoltre, viene usato il meccanismo dei *SACK* e dei *DACK* per ridurre il volume degli ack e l'effetto degli *out-of-order packet*. In [6], viene modificato il TCP per adattare il suo comportamento ai cambiamenti di percorso. Viene usata la presenza di pacchetti ricevuti fuori ordine (*out-of-order packet* (OOO)) come indicazione di un *route-changes*. L'analisi della presenza degli OOO viene fatta, sia dal Sender che dal Receiver (il Receiver lo notifica al Sender usando i pacchetti di ack), apportando delle modifiche ai pacchetti trasmessi per poterne rilevare l'ordine preciso di spedizione. Il Sender, su scoperta di un OOO, disabilita il meccanismo di controllo della congestione (per un tempo pari a T1) e, se durante il precedente periodo (T2) ha sofferto di congestione (timeout o tre ack duplicati), riporta lo stato della connessione a quello che aveva prima dello scattare del meccanismo di controllo della congestione (salta la fase di slow start).

2.2.7 ADTCP

In [7], viene seguito un approccio end-to-end per modificare il comportamento del TCP e risolvere i problemi legati alla mobilità dei nodi. In tale studio vengono identificate le varie situazioni che si possono venire a creare durante la vita di una connessione e vengono indicate le azioni che devono essere intraprese per ottenere un comportamento accettabile:

- *Congestion* (*CONG*) - quando la rete è congestionata, viene utilizzato lo stesso meccanismo di controllo della congestione usato dal TCP.

Nel caso in cui la rete non è congestionata si possono distinguere i seguenti stati:

- *Channel_Err* (*CHERR*) - questo stato indica la perdita di pacchetti dovuta ad un elevato valore del BER. Il Sender, in questa situazione, deve ritrasmettere solo i pacchetti persi a causa di errori di trasmissione.
- *Route_Change* (*RTCHG*) - nel caso di una route-changes, possono andare persi dei pacchetti e può nascere una sequenza di pacchetti out-of-order. Vanno rispediti i pacchetti persi e va stimata la dimensione della finestra di controllo relativa al nuovo percorso.
- *Disconnection* (*DISC*) - se la rete rimane partizionata per molto tempo, il Sender deve congelare il suo stato e testare periodicamente la disponibilità di una nuova strada. Su ritrovamento di una nuova strada, il Sender deve stimare la nuova dimensione della finestra di controllo e riprendere la spedizione dei pacchetti.

Nell'ADTCP, lo stato di *CONG* è quello che ha la più alta priorità. Gli altri tre stati sono considerati solo quando la rete non è congestionata. Per determinare lo stato della rete, vengono usate, in combinazione tra di loro, varie metriche, ciascuna delle quali è influenzata da differenti fattori:

- *Inter-packet delay difference* (*IDD*) - misura la differenza di ritardo di trasmissione tra due pacchetti consecutivi. Guardiamo come si calcola:

$$IDD = A(i+1) - A(i) - (S(i+1) - S(i))$$

dove A(i) è il tempo di arrivo del pacchetto i-esimo e S(i) è l'istante di spedizione del pacchetto da parte del Sender. Su ricezione di ciascun pacchetto, il reciver calcola l'*IDD*. Il valore dell'*IDD* cresce al presentarsi della congestione, non è affetto dal *random channel error* ed è indipendente dal comportamento del Sender. In ogni modo, l'*IDD* può essere influenza-

to da condizioni quali la nascita di out-of-order packet causati dalla mobilità.

- *Short-term throughput* (*STT*) - è un indicatore di congestione e vale

$$SRTT = \frac{N(T)}{T}$$

dove N(T) è il numero di pacchetti ricevuti nell'intervallo T. Rispetto all'*IDD* è meno sensibile rispetto agli *out-of-order packet*, è più robusto rispetto ai cambiamenti di strada, e dipende molto dal *rate* trasmissivo della sorgente e da altre cause di perdita dei pacchetti.

- *Packet out-or-order ratio* (*POR*) - indica la percentuale di pacchetti che arrivano fuori ordine

$$POR = \frac{N_0(T)}{N(T)}$$

dove No(T) è il numero di pacchetti out-of-order durante il periodo T. *POR* viene usato come indicazione di una *route-changes*.

- *Packet loss ratio* (*PLR*) - indica il numero di pacchetti mancanti nella finestra del ricevitore:

$$PLR = \frac{N_l(T)}{N(T)}$$

dove Nl(T) è il numero di pacchetti persi durante il periodo T.

I vari stati della rete vengono identificati tramite le metriche viste sopra. I parametri *IDD* e *STT* sono utilizzati per individuare la congestione della rete: valori alti del *IDD* e bassi di *STT* sono indice di congestione della rete, mentre le altre combinazioni dei loro valori indicano una assenza di congestione. Un insieme di valori di *POR* alti indica una *route-changes* ed un alto valore di *PLR* indica un alto livello di *channel error*. Lo stato di *DISC* viene identificato quando non si ha con-

gestione e scatta un timeout per un pacchetto spedito. L'ADTCP viene ottenuto operando delle opportune modifiche presso il Sender ed il Receiver TCP. Il Receiver, per ogni pacchetto ricevuto valuta i valori delle metriche viste (seguendo un opportuno algoritmo) e stima lo stato della rete. L'informazione sullo stato della rete viene poi inserita nei pacchetti di ack e passata al Sender. Il Sender procede normalmente fino a quando non riceve un terzo ack duplicato o fino allo scattare di un timeout. A questo punto, in base alle informazioni sullo stato della rete contenute nel pacchetto di ack, si comporta in maniera adeguata, invocando il meccanismo di controllo della congestione solo quando è necessario.

2.2.8 TCP-RCWE

L'obiettivo del TCP/RCWE [10] è quello di mettere il TCP nella condizione di poter distinguere tra pacchetti persi a causa della congestione, pacchetti persi a causa del movimento dei nodi e pacchetti persi a causa della contesa al livello MAC. Il TCP/RCWE si basa sul TCP/ELFN, apportandone le seguenti modifiche. Il TCP Sender si può trovare in tre stati: *normal*, *link-break* e *congestion*. Il Sender in normal state spedisce i pacchetti in modo normale. Su ricezione di un ELFN il Sender passa in *link-break state* (si comporta come il TCP/ELFN) e appena la strada diventa disponibile ritorna in *normal state*. Il meccanismo di controllo della congestione in questo stato è disabilitato. Per migliorare il meccanismo di controllo della congestione viene utilizzato un parametro che serve per stimare lo stato corrente della rete (*network state estimator* o *NSTATE*). Tale parametro viene calcolato su ricezione di un pacchetto secondo la seguente relazione:

$$NSTATE = \begin{cases} 1 & se\ RTOnew \le RTOold \\ 0 & altrimenti \end{cases}$$

Dove RTOnew è la stima del RTO ottenuta dal pacchetto ricevuto e RTOold è la stima precedente. Se *NSTATE* vale 0, la dimensione della finestra di congestione

non viene aumentata. Se *NSTATE* vale 1, la finestra di controllo viene incrementata normalmente (meccanismo classico del TCP). L'aumento del RTO viene preso come indicazione del fatto che nel percorso i nodi incontrano difficoltà ad accedere al canale (a causa della BER, dell'aumento del traffico o del cambiamento del percorso causato da un *link-failures*). Conseguentemente, il TCP non aumenta la finestra di controllo della congestione. Con questo meccanismo il TCP riesce a stimare in modo più efficiente lo stato della rete, non spedendo più pacchetti di quelli che il mezzo è in grado di ricevere e la finestra si stabilizza ad un valore ottimale. Inoltre, in [10] viene messo in evidenza come la formula classica per il calcola del RTO non vada bene per reti ad hoc. Il RTO viene calcolato dal TCP con la seguente formula:

$$SRTT = a \times RTT + b \times SRTT$$

dove SRTT è la stima del RTT ed RTT è il valore del RTT del pacchetto che ha ricevuto l'ack. Il SRTT viene stimato dando più peso alla storia della connessione (b > a). Nelle MANETs, essendo dinamiche, il peso della storia della connessione, nel calcolo del SRTT, deve assumere un valore differente.

Capitolo III.

Il protocollo TPA

Nel seguente capitolo vengono mostrate le limitazioni delle varie proposte di soluzione analizzate in precedenza e viene descritto il protocollo di trasporto TPA.

3.1 Limite delle soluzioni proposte in letteratura

I limiti delle varie soluzioni analizzate in precedenza possono essere così riassunti:

- Si basano tutte su una modifica del protocollo TCP, al fine di metterlo nella condizione di conoscere la causa della perdita di pacchetti.
- Nessuna proposta di soluzione riunisce in se i vari aspetti analizzati. In particolare, i protocolli che si occupano della mobilità non prendono in esame gli aspetti legati alla contesa (e viceversa). Inoltre, non ci sono protocolli che mettono insieme le varie indicazioni che possono essere utilizzate per determinare lo stato della rete.
- Il meccanismo di congestione del TCP risulta essere superfluo in ambiente MANETs, in quanto la dimensione ottima della finestra di controllo si aggira intorno ad un numero basso di pacchetti (4-5 pacchetti).

L'inutilità di un meccanismo di gestione della congestione come quello utilizzato dal TCP e la necessità di studiare dei meccanismi in grado di minimizzare il numero di ritrasmissioni non necessarie di pacchetti (per risparmiare energia e minimizzare il carico di lavoro offerto alla rete) e di rilevare lo stato della rete (al fine di adattarsi ad esso), ci hanno spinto a realizzare un protocollo di trasporto studiato appositamente per ambienti MANETs.

2.3 Descrizione del protocollo TPA

Il protocollo TPA (Transport Protocol for Ad-Hoc networks), fornisce un servizio di spedizione dei pacchetti affidabile e connection-oriented e introduce una serie di novità rispetto al TCP classico, quali la gestione delle condizioni di *route-failures* e di *route-changes* e l'introduzione di un nuovo meccanismo di controllo della congestione. Inoltre, il TPA implementa una nuova politica di ritrasmissione dei pacchetti andati in timeout, cercando di ridurre il numero di ritrasmissioni non necessarie. Di seguito vengono analizzati in dettaglio tutti gli aspetti del TPA.

3.2.1 Formato dei pacchetti

Esaminiamo il formato dell'header dei pacchetti TPA:

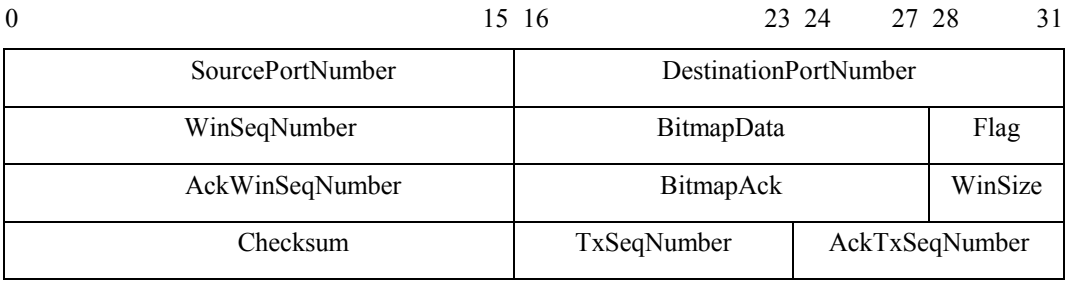

Figura 19 – Formato del pacchetto TPA.

Source e Destination Port Number (16 bit). Questa coppia di campi serve, in combinazione con l'IP-address, per identificare in modo univoco una connessione. Il port number ha un significato locale sull'host a cui si riferisce.

WinSeqNumber (16 bit). Il nostro protocollo TPA, prende i dati e li divide in blocchi di k pacchetti. Questo campo identifica il particolare blocco a cui appartiene il pacchetto che si sta spedendo. Dato che il nostro è un servizio di tipo *full-duplex,* ciascun capo della connessione deve mantenere il proprio *WinSeqNumber*.

BitmapData (12 bit). Questo campo è formato da k bit (con k numero di bit del gruppo) e serve per riconoscere in modo univoco un pacchetto all'interno del particolare blocco a cui appartiene. In combinazione con il *WinSeqNumber* identifica in maniera univoca un pacchetto. Un "1" in posizione i-esima della bitmap, signi-

fica che stiamo trasmettendo l'i-esimo pacchetto del blocco *WinSeqNumber*. Dato che il nostro è un servizio di tipo *full-duplex,* ciascun capo della connessione deve mantenere la propria bitmapData.

AckWinSeqNumber (16 bit). Identifica il gruppo a cui appartiene il pacchetto che si sta acknoleggiando.

BitmapAck (12 bit). Questo campo è formato da k bit (con k numero di pacchetti del gruppo) e contiene l'indicazione di tutti i pacchetti del blocco *AckWinSeqNumber* che sono stati ricevuti correttamente dalla destinazione. Per ogni pacchetto ricevuto correttamente è presente un "1" nella posizione corrispondente.

TxSeqNumber (8 bit). Questo campo viene incrementato ogni volta che si effettua la trasmissione di un pacchetto del blocco (comprese le ritrasmissioni). Ha significato locale all'interno del blocco.

AckTxSeqNumber (8 bit). Ha significato solo per pacchetti di ack. Indica il *txSeqNumber* del pacchetto che si sta acknoleggiando. Serve per poter identificare il pacchetto a cui si riferisce l'ack, poiché dalla *bitmapAck* non è possibile ricavarlo. Questo campo ha significato locale all'interno del gruppo.

Flag (4 bit). Il pacchetto contiene i seguenti flag:

- ACK - segnala che i campi AckWinSeqNumber e BitmapAck sono validi.
- RST - serve per resettare la connesione.
- SYN - indica che il pacchetto serve per iniziare la connessione.
- FIN - serve per la chiusura della connessione.

Checksum (16 bit). Campo utilizzato per rilevare eventuali errori di trasmissione.

WinSize (4 bit). Campo utilizzato per il *flow-control*. Serve per far capire al Sender quanti pacchetti è in grado di ricevere il Receiver.

3.2.2 Apertura della connessione

L'apertura della connessione si basa sul protocollo *three-way handshake*. Tale protocollo si basa sul seguente scambio di pacchetti (figura 20):

1. Appena il lato client vuole aprire una connesione, spedisce un segmento speciale, avente il bit di SYN settato ad uno ed il campo dati vuoto. Inoltre

sceglie il txSeqNumber ed il winSeqNumber da utilizzare per la spedizione del SYN (X,Y).

2. Su ricezione del segmento di SYN, il lato server alloca le strutture dati per la connessione che si sta aprendo e spedisce un segmento di risposta al client. Tale segmento ha il campo dati vuoto e deve contenere tre informazioni importanti: deve avere il flag di SYN settato ad uno; deve avere il flag di ACK settato ad uno e deve indicare che si sta acknoleggiando il segmento di SYN aventi txSeqNumber pari a X ed winSeqNumber pari a Y; deve scegliere un txSeqNumber ed un winSeqNumber per la spedizione del SYN da trasmettere al lato client.
3. Il client, su ricezione di tale segmento, considera la connessione stabilita e spedisce un segmento di ACK al server, indicandogli che ha ricevuto il suo SYN.
4. Il Server, su ricezione del segmento di ACK, considera aperta la connessione.

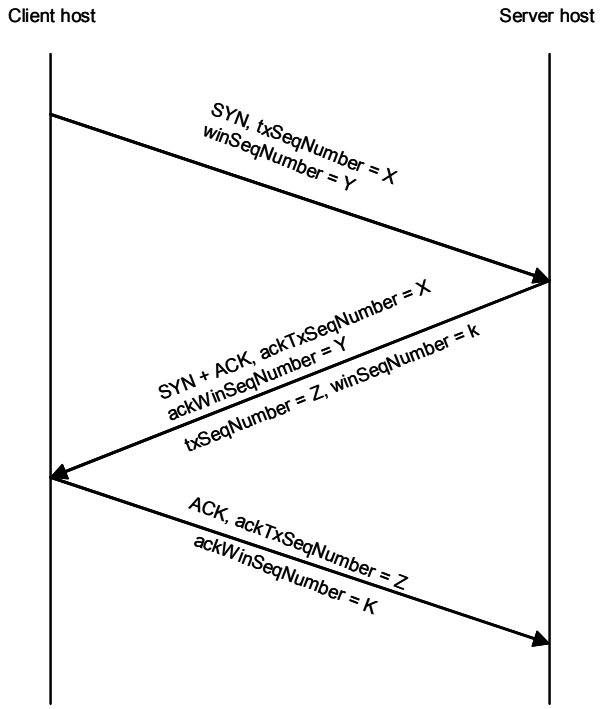

Figura 20 – Fase di apertura della connesione.

Una volta che i quattro passi visti sono stati completati dal Server e dal Client, la connessione risulta aperta e può cominciare lo scambio di dati. In tutti i futuri segmenti, il flag di SYN deve essere settato a zero. La spedizione dei segmenti di SYN (sia dal lato client che da lato server) deve essere affidabile. Appena il TPA spedisce un pacchetto di SYN, fa partire un timer. Se l'ACK per tale SYN non arriva prima dello scattare del timer, il SYN viene dato per perso e rispedito. Questo comporta che, sia il server che il client, devono essere preparati a ricevere dei pacchetti duplicati. Il client, su ricezione di un SYN duplicato, deve supporre che l'ACK a tale SYN sia andato perso e deve rispedirlo. Il server, su ricezione di un SYN duplicato, deve supporre che il suo pacchetto di SYNACK sia andato perso e deve rispedirlo. Ci sono altri casi particolari che si possono trovare in [18].

3.2.3 Chiusura della connessione

Essendo il servizio offerto dal TPA di tipo *full duplex*, la connessione deve essere chiusa sia dal lato client che dal lato server. Ad esempio, supponiamo che sia il lato client a chiudere per primo la connessione. Viene eseguito il seguente scambi di pacchetti (figura 21):

1. Il client spedisce un segmento con il flag di FIN settato ad uno al server, indicandogli che non ha più pacchetti da spedire.
2. Il server, su ricezione di tale pacchetto, spedisce un segmento di ACK, informando il client dell'avvenuta ricezione del pacchetto di FIN.

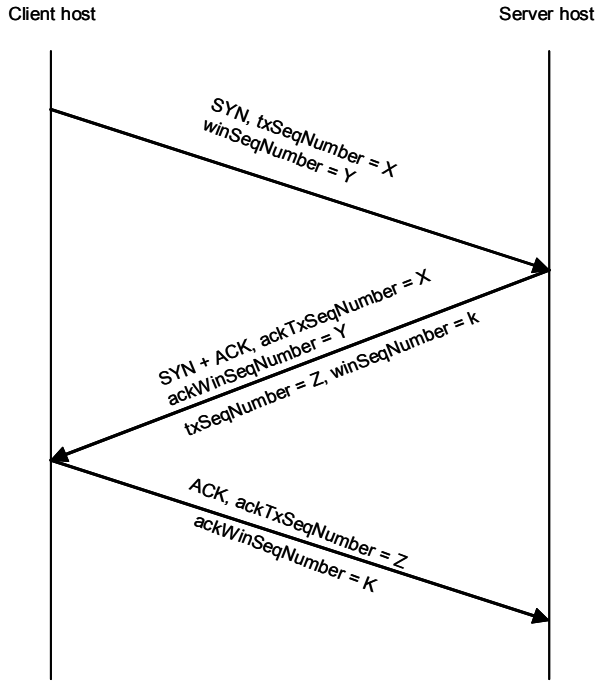

Figura 21 – Fase di chiusura della connesione.

Il server, dopo la ricezione del pacchetto di FIN, può ancora continuare a spedire pacchetti al client. Non appena anche lui decide di terminare la connessione, ripete le stesse operazioni del client, spedendo un segmento di FIN ed aspettando l'ACK di risposta. A questo punto la connessione è chiusa da entrambi i lati e possono essere deallocate le relative strutture dati (sia sul lato client che sul lato server). I segmenti di FIN devono essere spediti in modo affidabile. Su spedizione del FIN, il TPA setta un timer. Se non viene ricevuto l'ACK del FIN prima dello scattare del timer, questo viene dato per perso e rispedito.

3.2.4 Meccanismo di trasmissione dei pacchetti

Passiamo alla descrizione vera e propria del protocollo. Il TPA è basato su uno schema di tipo *sliding-window*, dove la dimensione della finestra di controllo varia in accordo agli algoritmi di *congestion-controll* e di *flow-controll.* Come visto in precedenza, in ambiente MANET è facile che si perdano pacchetti (soprattutto pacchetti di ack) e sono frequenti anche situazioni di sottostima del RTO. Questo porta a situazioni in cui alcuni pacchetti sono dati per persi presso la sorgente anche se sono arrivati correttamente a destinazione (come visto in precedenza per il

TCP). Al fine di minimizzare il carico di lavoro offerto alla rete e di risparmiare energia (riducendo il numero di ritrasmissioni non necessarie), il TPA usa un meccanismo di trasmissione di tipo circolare: i dati da trasmettere sono organizzati in blocchi di K pacchetti. Più precisamente, la sorgente TPA prende un certo numero di bytes (sufficiente a formare K pacchetti TPA) dal buffer[1] contenente i dati provenienti dall'applicazione, li incapsula in pacchetti TPA e tenta di trasmetterli in maniera affidabile a destinazione. L'header di ciascun pacchetto TPA, ha un campo *winSeqNumber* per identificare il blocco a cui appartiene il pacchetto ed un campo *bitmapData*, formato da K bit, che identifica la posizione del pacchetto dentro il blocco. Inoltre ci sono due campi per fare il piggybacking degli ack dentro i pacchetti dati: *ackWinSeqNumber* e *bitmapAck*. L'*ackWinSeqNumber* identifica il blocco a cui si riferisce l'ack, mentre, un bit settato ad uno dentro *bitmapAck* indica che il corrispondente pacchetto dentro il blocco è arrivato correttamente a destinazione. Ogni qual volta viene spedito un pacchetto, la sorgente TPA setta un timer e si mette in attesa di ricevere l'ack di risposta dalla destinazione. Su ricezione di un ack per i pacchetti in volo, il TPA si comporta nel modo seguente: i) valuta la dimensione della finestra di controllo in accordo con gli algoritmi di *flow* e *congestion control*; ii) spedisce i pacchetti successivi all'ultimo pacchetto acknoleggiato fino a raggiungere un numero di pacchetti in volo pari alla dimensione della finestra. Mettiamoci nella situazione descritta in figura 22:

[1] Il blocco potrebbe includere meno di K pacchetti se il buffer non contiene un numero sufficiente di bytes.

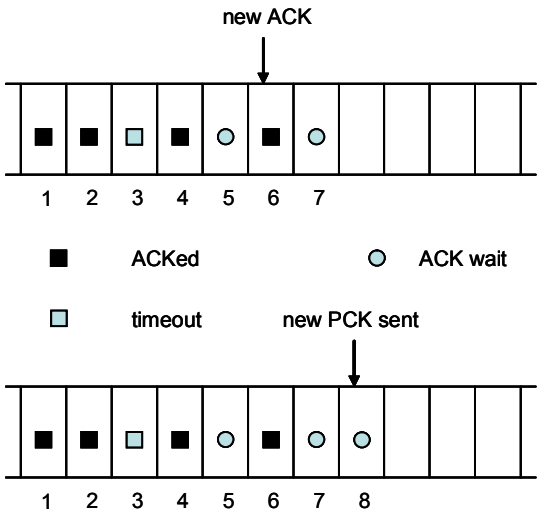

Figura 22 – Gestione della finestra di controllo su ricezione di un ack.

Supponiamo di trasmettere con una finestra di dimensioni pari a 3 e che siano in volo i pacchetti 5,6 e 7. Su ricezione dell'ack per il pacchetto 6, viene spedito il pacchetto 8. A questo punto i pacchetti in volo sono 5, 7 e 8.

Allo scattare del timer per un pacchetto, non si effettua la ritrasmissione del pacchetto andato in timeout, ma si va avanti nella trasmissione come visto in precedenza. Esaminiamo la situazione descritta in figura 23.

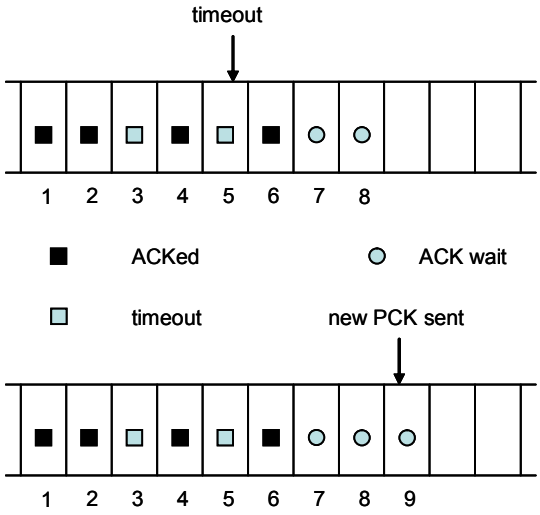

Figura 23 - Gestione della finestra di controllo allo scattare di un timeout.

Allo scattare del timeout per il pacchetto 5, non viene effettuata la ritrasmissione di 5 ma si passa a trasmettere il pacchetto 9. Questo procedimento si ripete fino a quando si giunge al k-esimo pacchetto dentro il blocco. A questo punto, si riparte con la ritrasmissione dei pacchetti non acknoleggiati appartenenti al blocco, seguendo sempre la tecnica circolare vista in precedenza. Esaminiamo la situazione presentata in figura 24:

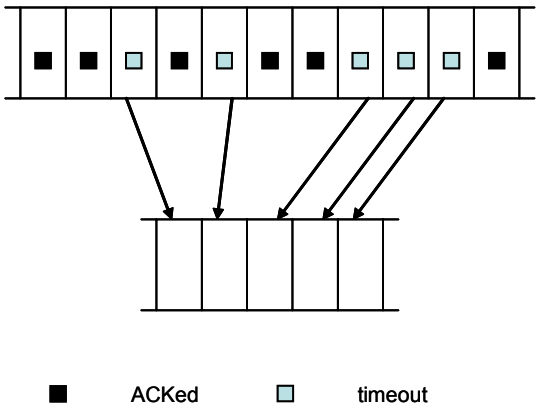

Figura 24 – Gestione delle ritrasmissioni.

Ad un certo punto della trasmissione si arriva alla fine del blocco. Si prendono i pacchetti non acknoleggiati e con essi si forma un nuovo blocco con cui riprendere la trasmissione. Se un pacchetto dentro il novo blocco viene acknoleggiato prima di essere ritrasmesso, viene eliminato dal blocco.

Questa procedura si ripete fino a quando tutti i pacchetti dentro il blocco di partenza non sono stati acknoleggiati. Solo a questo punto il Sender passa alla costruzione ed alla trasmissione del blocco successivo.

Al fine di raggiungere l'obiettivo di minimizzare il numero di ritrasmissioni non necessarie, bisogna che il Sender conosca tutti i pacchetti ricevuti correttamente dalla destinazione. Gli ack del TPA contengono, a tal fine, le informazioni su tutti i pacchetti ricevuti dalla destinazione, in modo che, anche se alcuni di essi vanno persi, i successivi compensino tale perdita. Il campo *bitmapAck* dei pacchetti di

ack contiene un bit settato ad 1 in corrispondenza di tutti i pacchetti del blocco pervenuti correttamente a destinazione.
Lo schema di trasmissione del TPA, presenta diversi vantaggi rispetto allo schema di trasmissione del TCP. Primo, la probabilità di ritrasmissioni non necessarie è ridotta. Il TCP allo scattare di un timeout, rispedisce immediatamente il relativo pacchetto. Il TPA invece, va avanti nella trasmissione dei pacchetti. Questo, come abbiamo gia detto, è importante nelle MANET, dove la mobilità può portare ad una mancata corrispondenza tra RTO e RTT del pacchetto e dove possono essere persi numerosi pacchetti di ack. Secondo, l'uso della *bitmapAck* fa si che un singolo ACK sia sufficiente a notificare al Sender la situazione sulla trasmissione del blocco, rendendo il TPA poco sensibile alla perdita degli ACK. Terzo, il Sender non soffre del fenomeno degli out-of-order packet che si presentava con il TCP. Questo rende il TPA capace di operare efficientemente anche in MANETs che usano protocolli di routing *multi-path* [20], al contrario del TCP che in tali reti si comporta in modo scadente [11].

3.2.5 Flow-control

Il meccanismo di controllo di flusso del TPA è simile a quello utilizzato dal TCP. Il ricevitore, in funzione della sua disponibilità di spazio nel buffer, indica al Sender quanti pacchetti può spedire attraverso il campo *WinSize* dell'header TPA. Il Sender deve mantenere il numero dei pacchetti in volo (non acknoleggiati e non in timeout) al disotto del numero indicato in *WinSize*.

3.2.6 Gestione delle route-failures

La gestione delle *route-failures* da parte del TPA segue lo schema del TCP-ELFN [4], e si basa sull'utilizzo di un feedback esplicito da parte del livello network. La scelta di utilizzare il messaggio di ELFN è stata basata sulla osservazione che i principali protocolli di routing sono in grado di fornire tale messaggio e quindi conviene sfruttarlo al fine di mettere il TPA nelle condizioni di avere il maggior numero possibile di informazioni sullo stato della rete. Se un nodo intermedio ca-

pisce che un pacchetto non può essere spedito al nodo successivo a causa di una *link-failures* (causata dalla mobilità o dalla contesa) e non esiste una strada alternativa per andare a destinazione, spedisce (sfruttando le caratteristiche del protocollo di routing) un messaggio di *Explicit Link Failure Notification* (ELFN) al Sender del pacchetto. Presso la sorgente del pacchetto, il livello network inizia la ricerca di una nuova strada per andare a destinazione e informa il TPA della ricezione messaggio di ELFN. Come nel caso del TCP-ELFN, si assume che la rete non informi il livello trasporto del ritrovamento della strada che porta a destinazione. Quindi bisogna che la sorgente TPA testi la rete ad intervalli regolari per accorgersi del ritrovamento di una strada.

In una *link-failures* possono incappare, indipendentemente, sia i pacchetti di ack che i pacchetti dati. Inoltre, la presenza di path asimmetrici (i pacchetti dati ed i pacchetti di ack possono seguire strade diverse) può portare a situazioni in cui i dati (ack) subiscono *link-failures*, mentre gli ack (dati) no, causando una mancanza di strada tra sorgente e destinazione (tra destinazione e sorgente) e non viceversa. Una *link-failures* tra Sender e Receiver, provoca la spedizione del messaggio di ELFN verso il Sender, mentre una *link-failures* tra Receiver e Sender provoca la spedizione del messaggio di ELFN verso il Receiver.

Guardiamo il comportamento del Sender e del Receiver TPA. Il Sender, su ricezione del messaggio di ELFN, entra congela il suo stato (entra in LF-state) e stoppa la trasmissione dei pacchetti. In tale stato, il Sender TPA spedisce un pacchetto di *probe* ogni T secondi, per testare la disponibilità di una strada per andare a destinazione. Su ricezione dell'ack per tale pacchetto, il Sender capisce che una strada è stata ritrovata dal protocollo di routing e intraprende le seguenti azioni: i) abbandona il *LF-state*; ii) setta la dimensione della *congestion window* al suo valore massimo; iii) riparte dalla spedizione del pacchetto che ha generato il messaggio di ELFN. In *LF-state* (a causa della possibile presenza di path asimmetrici) possono arrivare degli ack per i pacchetti spediti prima di quello che ha subito la *link-failures*. Il Sender, non deve prendere tali arrivi come indice di un ritrovamento di una strada.

Il Receiver TPA, si comporta in maniera analoga al Sender: su ricezione del messaggio di ELFN, il Receiver setta un timer ad un valore pari a T secondi, riprendendo la spedizione degli ack solo allo scattare di tale timer. Durante questo periodo di tempo, il Receiver può ricevere alcuni pacchetti dati. Su ricezione di questi, deve aggiornare il buffer di ricezione ma non deve spedire gli ack di risposta. Il Sender, si accorge della *link-failures* avvenuta tra Receiver e Sender con un meccanismo implicito. Essendo il Receiver impossibilitato a mandare ack fino al ritrovamento di una strada, una disconnessione tra R e S si presenta presso il Sender con l'assenza di ack per un tempo di circa T secondi. L'intervallo di tempo T è una stima del tempo impiegato solitamente dai protocolli di routing per recuperare da una *route-failures*.
Nella casistica delle *route-failures* rientra anche la disconnessione della rete. Testando la rete ad intervalli regolari e pari a T, non si hanno i problemi a cui va incontro il TCP [1], e la trasmissione riprende non appena la disconnessione si è risolta.

3.2.7 Gestione delle route-changes

Il TPA, per il calcolo del *Retransmission Timeout* (RTO) dei pacchetti, usa la stima del *Round Trip Time* (RTT) della connessione. Tale stima viene ricavata utilizzando le stesse formule del TCP:

$$ERTT(n) = (1-g) \times ERTT(n-1) + g \times SRTT(n)$$

$$DEV(n) = h \times |SRTT(n) - ERTT(n)| + (1-h) \times DEV(n-1)$$

$$RTO(n) = ERTT + 4 \times DEV(n)$$

dove ERTT (n) e DEV (n) sono rispettivamente il valore medio e la deviazione standard della stima del RTT all'n-esimo passo, SRTT(n) è il valore dell'n-esimo campione del RTT, RTO(n) è il retransmission timeout calcolato all'n-esimo passo, g e h ($0 \le g, h \le 1$) sono dei parametri reali [18] settati in modo da avere

l'ERTT sbilanciato verso la storia della connessione (g <<1, h << 1). Come visto nei capitoli precedenti, le *route-failures* causate dalla mobilità o dalla contesa, scatenano nel protocollo di routing la ricerca di un nuovo percorso per andare a destinazione. Tale ricerca può produrre un cambiamento nel percorso seguito dai pacchetti (data o ack). Questo cambiamento si può verificare anche quando una *link-failures* viene recuperata localmente, senza scatenare presso la sorgente del pacchetto, una nuova ricerca di un path per andare a destinazione.
Quando occorre una *route-changes*, può modificarsi il RTT dei pacchetti. Dato che il calcolo del RTO e del RTT è sbilanciato verso la storia della connessione, il RTO può risultare non essere più appropriato per il nuovo path per numerose trasmissioni successive e si può andare incontro a fenomeni di sovra e sottostima del retransmission timeout. Il TPA è in grado di accorgersi delle *route-changes* non appena si verificano, modificando di conseguenza la stima del RTT e del RTO al fine di velocizzare la convergenza di questi ai valori corretti. Il TPA individua una *route-changes* dopo i seguenti eventi:

- Non appena una strada è nuovamente disponibile dopo la ricezione del messaggio di ELFN.
- In assenza di cambiamenti di percorso, l'ack per l'n-esimo pacchetto dovrebbe arrivare (con molta probabilità) entro un intervallo di tempo pari a:

$$ERTT(n) - 4 \times DEV(n) \leq \ldots \leq ERTT(n) + 4 \times DEV(n)$$

Una *route-changes* può far si che gli ack per i pacchetti spediti arrivino con un eccessivo ritardo o anticipo. Si fissa un intervallo pari a:

$$ERTT(n) - a \times DEV(n) \leq \ldots \leq ERTT(n) + a \times DEV(n)$$

con a parametro reale. Su ricezione di N pacchetti ritardati o anticipati rispetto a tale intervallo, si assume una *route-changes*.

Su individuazione di una *route-changes*, si sostituiscono i parametri g e h con i parametri g1 e h1. Tali parametri (g1 > g, h1 > h) fanno si che la nuova stima del RTT e del RTO sia maggiormente influenzata dai nuovi campioni del RTT rispetto alla storia (RTT e RTO vecchi). Così facendo si velocizza la convergenza dei valori del RTT e RTO a quelli relativi al nuovo path. Per finire, su ricezione di N1 pacchetti dentro l'intervallo considerato in precedenza, i valori dei parametri vengono ripristinati con g e h.

3.2.8 Gestione della congestione

L'utilizzo di una finestra di controllo di flusso di dimensioni ridotte (3-4 pacchetti) fa si che non si renda necessario l'utilizzo di un meccanismo di controllo della congestione come quello del TCP. Questo è messo in evidenza anche in [9], dove viene osservato che la perdita di pacchetti a causa di buffer-overflow è rara e la perdita di pacchetti è dominata dalla contesa a livello link-layer. In reti Ad-Hoc, per congestione si intende una condizione di *network overload* e cioè uno stato della rete in cui si impone un traffico maggiore di quello che permette di dare la massima utilizzazione del canale. La condizione di network overload porta al sorgere della contesa al livello link layer e alla perdita di numerosi pacchetti. L'invio del messaggio di ELFN viene scatenato non solo da un cambiamento di path, ma anche dalla perdita di pacchetti (dati o ack) causata dalla contesa ([17]). Quando un nodo non riesce a raggiungere il proprio vicino a causa della contesa, invia un messaggio di *link-failures* al protocollo di routing, il quale si comporta in modo analogo a quello che accade nel caso di mobilità dei nodi. In altre parole, il protocollo di routing inizia la fase di ricerca di un nuovo path per andare a destinazione. La congestione causata dalla contesa a livello link-layer si manifesta, al livello trasporto, in due modi:

- Un nodo, durante la spedizione di un *data packet* può incappare in una *link-failures* causata dalla contesa (*data inhibition)*. Di conseguenza, spedisce il messaggio di ELFN alla Sender TPA del pacchetto. Questo evento non può essere distinto da una *route-failures*.

- Un nodo può fallire nel trasmettere un *ack packet* a causa di una *link-failures* causata dalla contesa (*ack inhibition*). Il messaggio di ELFN viene ricevuto dal Receiver TPA e il Sender esperimenta una serie di timeout consecutivi (gli ack non hanno una strada disponibile per andare a destinazione) senza ricevere nessun messaggio di ELFN. Allo scattare di *UnackedPck* timeout consecutivi, il Sender assume che ci sia stata una *ack inhibition* e reagisce entrando nel *congestion state*.

Il meccanismo di controllo della congestione del TPA è molto semplice: non appena si entra nel *congestion state*, si passa a trasmettere i dati con una finestra di 1 pacchetto, per ridurre il carico di lavoro offerto alla rete (in rete c'era troppo traffico rispetto a quello che poteva sostenere senza causare collisioni). Il *congestion state*, viene abbandonato non appena la sorgente riceve *contAck* ack consecutivi dalla destinazione.

Una situazione analoga alla precedente, e indistinguibile da questa, si presenta nel caso di una *route-failures*, dovuta alla mobilità, nel percorso tra destinazione e sorgente. Pertanto, una volta che si passa nel *congestion state*, si modifica la formula per il calcolo del RTO e del RTT.

3.2.9 Gestione dei percorsi Multi-Path

Può capitare che tra sorgente e destinazione esistano due strade alternative (figura 25):

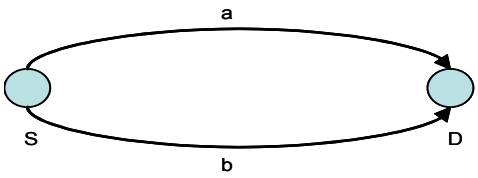

Figura 25 – Multi-Path.

Ad esempio, per andare da S a R esistono le strade alternative 'a' e 'b'. La strada 'a' viene seguita dai pacchetti con probabilità 'p', mentre la strada 'b' viene seguita con probabilità 1-p. Se le due strade sono molto differenti, per quanto riguarda

il RTT, può capitare che si presentino degli ack ritardati o anticipati. Supponiamo che la strada ‘a’ sia più corta (lunga) di ‘b’. Essendo il percorso ‘a’ il più probabile, il RTO viene calcolato in modo più sbilanciato verso la connessione ‘a’. Come conseguenza, i pacchetti che percorrono la strada ‘b’ possono andare incontro ai fenomeni di sotto o sovrastima del RTO. Ogni tanto si presenteranno alcuni pacchetti sotto o sovrastimati. Di questo si deve tenere di conto quando si va a stimare il numero di pacchetti ritardati o anticipati (capitolo 3.3.1).

3.2.10 Gestione dei duplicati

Con il meccanismo di trasmissione utilizzato, il Receiver non può ricevere dei pacchetti duplicati, poiché tutte le volte che si spedisce un pacchetto, si incrementa il campo *txSeqNumber* dell’header TPA. Il Receiver può però ricevere più volte lo stesso pacchetto dati. Questo avviene, ad esempio, quando il Sender rispedisce i pacchetti unacked del blocco. Il Receiver deve supporre che gli ack spediti siano andati persi e quindi su ricezione di un pacchetto già ricevuto, costruisce la bitmap e spedisce l’ack verso il Sender, scartando il pacchetto dati ricevuto. Allo stesso modo il Sender non può ricevere dei pacchetti ack duplicati, poiché è presente il campo *ackTxSeqNumber* dell’header TPA. Gli unici casi in cui si possono avere dei duplicati sono i seguenti:

- **Duplicati prima della chiusura della connesione -** problemi con questo meccanismo si hanno quando il tempo che un pacchetto può passare dentro la rete a girare è superiore al tempo impiegato dal *WinSeqNumber* a ciclare (per pacchetti appartenenti alla stessa connessione). In questo caso, si può andare incontro all’indistinguibilità dei pacchetti con lo stesso *WinSeqNumber* e scartare un pacchetto che non deve essere scartato. Nell’esempio di figura 26, il pacchetto 1 rimane a giro per la rete per un tempo maggiore di quello impiegato dal *WinSeqNumber* per riciclare, e arriva al Receiver prima del nuovo pacchetto 1. Il Receiver, non sapendo quale dei due pacchetti è un duplicato, scarta il secondo. Questo comportamento è da evitare.

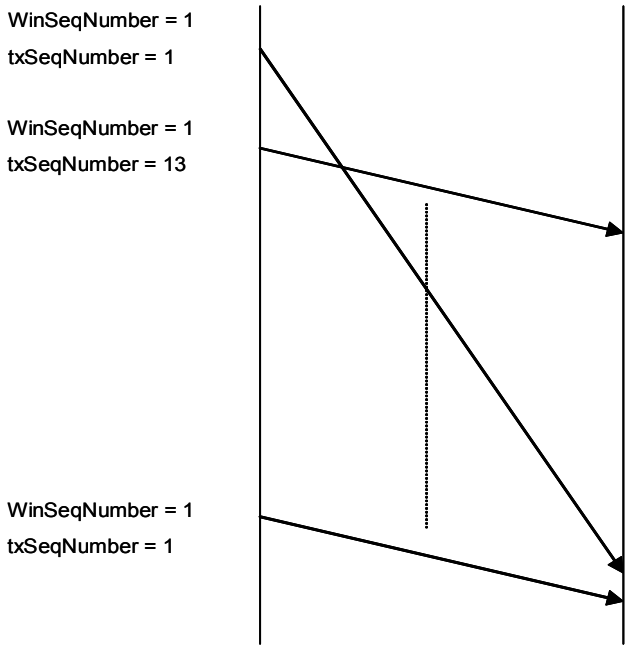

Figura 26 – Duplicati prima della chiusura della connessione.

Nel nostro caso, il *WinSeqNumber* si riferisce ad un blocco di k pacchetti e quindi ricicla molto lentamente (dopo la spedizione di 2^16 * k pacchetti). Di conseguenza, non dovrebbe mai verificarsi la situazione in cui un pacchetto ritardato si viene a ripresentare dopo il riciclaggio. Inoltre, in reti di questo tipo, i pacchetti non hanno una vita molto lunga.

- **Duplicati dopo la chiusura della connesione -** il problema dei duplicati si può presentare anche tra due connessioni successive. Ad esempio, tra i pacchetti della nuova connessione possono arrivare dei pacchetti appartenenti alla vecchia connessione (che ancora girano nella rete), con un *winSeqNumber* identico. Prendiamo l'esempio di figura 27. In questo caso, il pacchetto con *winSeqNumber* pari a 3 della vecchia connessione, è indistinguibile dal pacchetto con *winSeqNumber* pari a 3 della nuova connessione. In questa situazione il receiver non è in grado di decidere correttamente su quale dei due pacchetti è il duplicato. Per evitare che si verifichi la circostanza esaminata, il Client TPA, dopo la chiusura della connessio-

ne, aspetta un periodo di tempo che va dai 30 secondi ai 2 minuti prima di riaprire una connessione. Nel caso in cui il sistema va in crash, ci dobbiamo astenere dallo spedire pacchetti per un certo periodo di tempo pari al massimo tempo di permanenza in rete dei pacchetti.

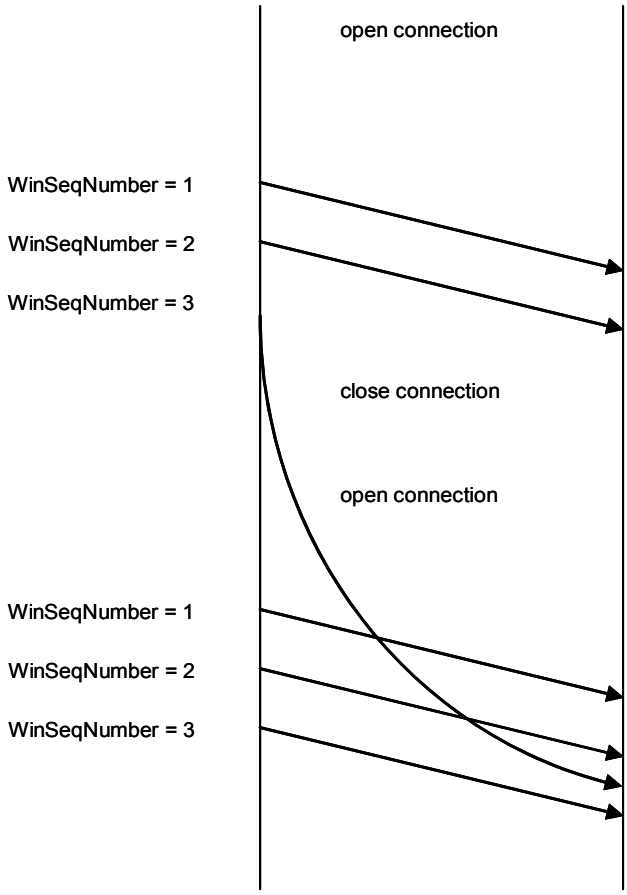

Figura 27 - Duplicati dopo la chiusura della connessione.

3.3 Formalizzazione del protocollo tramite macchina a stati finiti

Nel seguente capitolo viene fornita una descrizione del TPA tramite una rappresentazione a macchina a stati finiti. In figura 28 viene mostrato il diagramma a

stati che si riferisce all'intera vita di una connessione TPA, comprensivo della fase di apertura e chiusura delle connessioni.

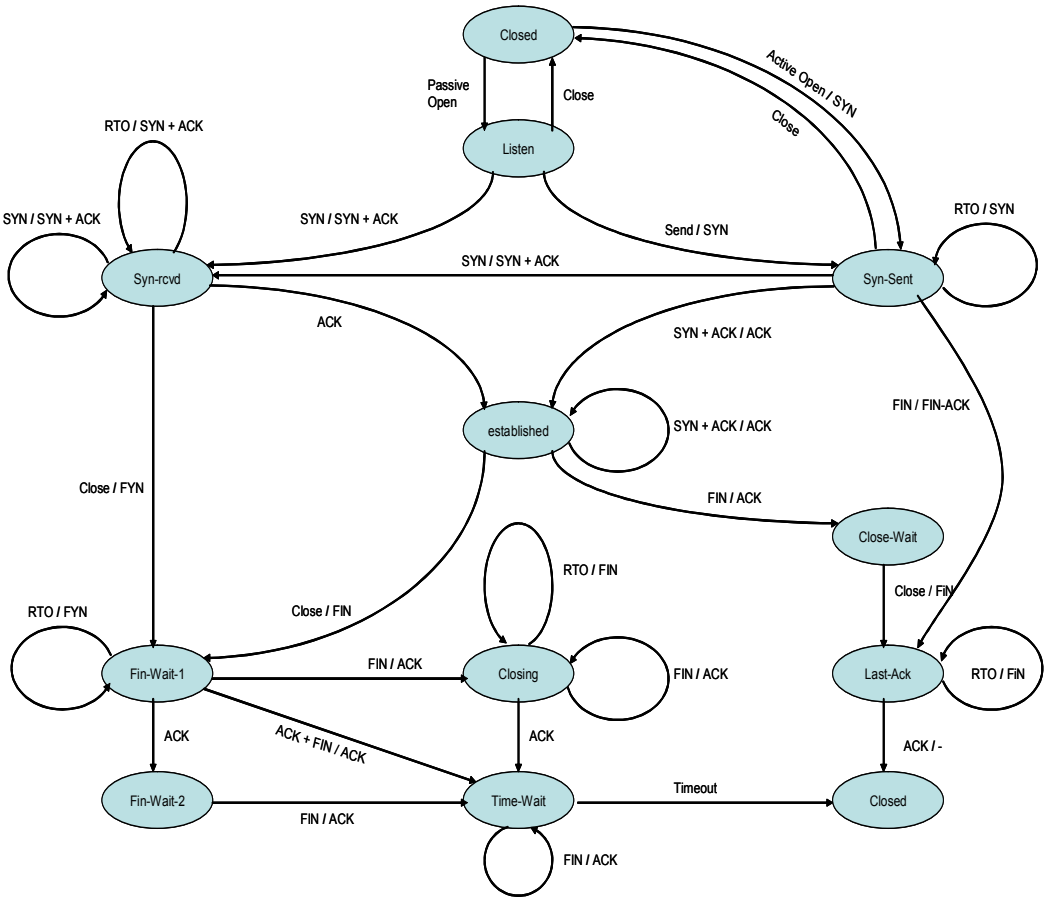

Figura 28 – Diagramma a stati della vita di una connessione TPA.

Descriviamo i vari stati in cui si può venire a trovare il TPA.

Listen-state - In tale stato, si viene a trovare il lato server della connessione su esecuzione di una *passive-open*. Il server si mette in attesa di ricevere un SYN da parte del lato client. Su ricezione di questo, si ha il passaggio nel *syn-received-state* e la trasmissione del pacchetto di SYN-ACK.

Syn-received-state – In tale stato possono presentarsi i seguenti eventi: i) ricezione di un SYN duplicato: il server considera perso il SYN-ACK e lo rispedisce; ii) ricezione dell'ACK del SYN spedito: si ha il passaggio in *established-state*; iii) scatto del RTO del pacchetto di SYN-ACK spedito: si suppone il SYN-ACK per-

so e si rispedisce; iv) se viene eseguita la *close*, viene spedito il messaggio di FIN al client.

Syn-sent-state – In tale stato possono presentarsi i seguenti eventi: i) invocazione della primitiva di *close*: si passa in *closed-state*; ii) ricezione del pacchetto di SYN-ACK: si spedisce l'ACK del SYN e si passa in *established-state*; iii) scatto del RTO del SYN spedito: viene dato per perso il SYN e viene rispedito; iv) ricezione di un SYN: siamo nel caso di apertura contemporanea della connessione e si spedisce un pacchetto di SYN-ACK; v) ricezione di un pacchetto di FIN: si spedisce il pacchetto di FIN-ACK.

Established-state – In tale stato, il TPA considera aperta la connessione tra sender e receiver e può cominciare la sua attività di spedizione dei pacchetti. Oltre alla ricezione di pacchetti dati e di ack, possono presentarsi i seguenti eventi: i) ricezione di un SYN-ACK duplicato: viene dato per perso il pacchetto di ACK relativo al SYN e viene rispedito; ii) ricezione di un pacchetto di FIN: viene spedito l'ack relativo a tale pacchetto e si passa in *close-wait-state*; iii) esecuzione di una close: viene spedito il pacchetto di FIN e si passa in *fin-wait-1-state*.

Fin-wait-1-state - In tale stato possono presentarsi i seguenti eventi: i) ricezione dell'ACK del FIN: si passa in *fin-wait-2-state*; ii) ricezione del FIN: si passa in *closing-state*; iii) ricezione di un FIN-ACK: la connessione si può considerare chiusa e si entra in *time-wait-state*; iv) allo scattare del RTO per il pacchetto di FIN questo viene dato per perso e rispedito. In tale stato il TPA può continuare a ricevere pacchetti dati e deve continuare a spedire gli ack per tali pacchetti.

Fin-wait-2-state - Su ricezione del pacchetto di FIN, si spedisce l'ACK relativo e si passa in *time-wait-state*. In tale stato il TPA può continuare a ricevere pacchetti dati e deve continuare a spedire gli ack per tali pacchetti.

Closing-state - In tale stato, il TPA è in attesa di ricevere l'ACK del FIN per poter chiudere la connessione. Possono esibirsi i seguenti casi: i) scatta il RTO per il pacchetto di FIN: viene dato per perso e rispedito; ii) ricezione di un FIN duplicato: viene dato per perso l' ACK del FIN e viene rispedito; iii) viene ricevuto l'ACK del FIN. Si passa in *time-wait-state*.

Time-wait-state - In tale stato, si setta un timer ad un valore che va dai 30 sec ai 2 minuti. Allo scattare di tale timer, si passa in *closed-state*. La funzione di tale timer è quella di impedire il fenomeno dei duplicati dopo la chiusura della connessione. In *time-wait-state* possono essere ricevuti anche dei pacchetti di FIN duplicati. Su ricezione di questi si spedisce il relativo ACK.

Close-wait-state - In tale stato, il TPA si mette in attesa di ricevere la *close* dalla applicazione. Quindi si può ancora avere la spedizione dei pacchetti verso l'altro lato della connessione. Su esecuzione della *close* da parte della applicazione, viene spedito il pacchetto di FIN e si passa in *last-ack-state*.

Last-ack-state - In tale stato ci si mette in attesa di ricevere l'ACK del FIN spedito. Allo scattare del timer questo viene rispedito. Su ricezione dell'ack si va in *closed-state*.

Di seguito viene riportata la descrizione del comportamento del Sender e del Receiver TPA nello stato di *established*.

3.3.1 Sender TPA

Come detto in precedenza, il Sender utilizza, per la trasmissione affidabile dei pacchetti, un protocollo a finestra di tipo go-back-n modificato e sfrutta il segnale di ELFN per avere informazioni sulle *route-failures* che avvengono tra S (Sender) e R (Receiver). Riassumiamo le varie situazioni gestite dal TPA e i meccanismi utilizzati per gestirle.

- Una **route-failures** viene rilevata utilizzando il messaggio di ELFN (3.2.5), oppure viene rilevata con lo scattare di un certo numero di timeout consecutivi (invalidazione del percorso tra Receiver e sorgente). In entrambi i casi, su ritrovamento di una strada vengono modificati i parametri per il calcolo del RTO, in accordo con quanto avviene per una *route-changes*. Questo perché una *route-failures* porta il protocollo di routing a ricercare una nuova strada per recapitare i pacchetti a destinazione e quindi, su ritrovamento di una strada, dobbiamo assumere che ci sia stata una *route-changes*.

- Come indicazione di **route-changes** si utilizza l'arrivo di un certo numero *delayAck* (*quickAck*) di ack ritardati (anticipati) rispetto all'intervallo di variazione del RTT (3.2.6). Nel calcolo di tale numero, non importa che i pacchetti ritardati (anticipati) siano consecutivi. La presenza dei cammini multi path può far sì che arrivi un ack giusto pur essendoci il bisogno di modificare il RTO. Prendiamo l'esempio di figura 24. Supponiamo che il percorso 'a' sia il più probabile e anche il più lungo (La > Lb). Il percorso 'a', ad un certo istante cambia (*route-changes*) e diventa di lunghezza L, con L > La > Lb. A questo punto, i pacchetti che percorrono 'a' si trovano ad essere ritardati. Bisogna modificare la legge per il calcolo del RTO. Allo stesso tempo, può capitare che qualche pacchetto prenda la strada 'b' ed arrivi entro l'intervallo (o in anticipo). Il calcolo degli ack ritardati (anticipati), deve essere riazzerato dopo un certo numero di ack arrivati entro l'intervallo di variazione del RTT o anticipati (ritardati) e non subito dopo la ricezione di un ack arrivato entro l'intervallo o in anticipo (ritardo). Inoltre, i valori di *delayAck* e *quickAck* non devono essere troppo piccoli, poiché la possibile presenza di cammini multi path può portare ad avere qualche pacchetto di ack ritardato o anticipato, senza che sia necessario modificare la formula del RTO. Dopo che è stata rilevata una *route-changes*, si modifica la legge per il calcolo del RTO. Su ricezione di *rightAck* ack entro l'intervallo di variazione del RTT, si ripassa al calcolo classico del RTO.
- Per rilevare la **contesa** si possono prendere due indicazioni(3.2.7). La contesa tra S e R si manifesta con la ricezione del messaggio di ELFN da parte del Sender. La contesa tra R e S, si manifesta con un numero pari ad *unackedPck* di pacchetti non acknoleggiati ed in timeout. I sintomi che abbiamo preso per la contesa, si possono presentare anche nel caso di una *route-failures* o di una eccessiva sottostima del RTO. In questo caso, la ricezione di un ack per i pacchetti unacked, implica che cominciano ad arrivare gli ack ritardati. Su ricezione di tale ack, si dovrebbe riportare la fi-

nestra di controllo al suo valore massimo. Questa non è una buona scelta, in quanto ci sono delle situazioni che danno fastidio, come il caso in cui si ha congestione ma in qualche maniera arriva un ack a destinazione. Ad esempio, se dopo che la contesa è stata rilevata arriva un ack ritardato (magari c'era stata, prima della congestione, una *route-changes*), questo non ci deve indurre a considerare la congestione come terminata. Come indice di fine congestione, si utilizza la ricezione di un numero di ack pari a *contAck*. Su ricezione di *contAck* ack per i pacchetti unacked, si ripassa a trasmettere con una finestra di controllo della congestione settata al suo valore massimo e si azzera il conteggio dei pacchetti unacked. Dato che la contesa porta ad un probabile cambiamento di percorso, su rilevazione di questa si modificano i parametri per il calcolo del RTO. Al calcolo classico del RTO si ripassa, come nel caso di *route-changes*, su ricezione di *RightAck* ack ricevuti entro l'intervallo di variazione del RTT.

In figura 29 è mostrato il diagramma a stati del Sender, che si riferisce allo stato di **Established** (cioè ad una connessione aperta).

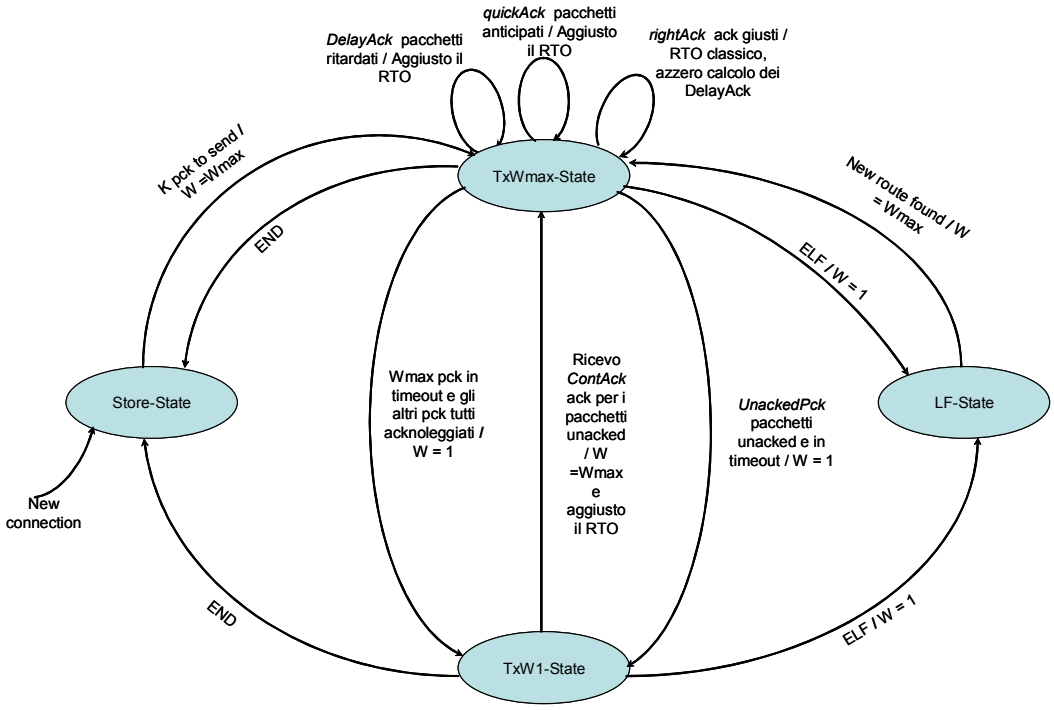

Figura 29 – Stato di Established del sender TPA.

Descriviamo i singoli stati presenti nel diagramma.

Store-State - In questo stato, descritto in figura 30, il Sender si mette in attesa di ricevere K pacchetti per formare il blocco da spedire.

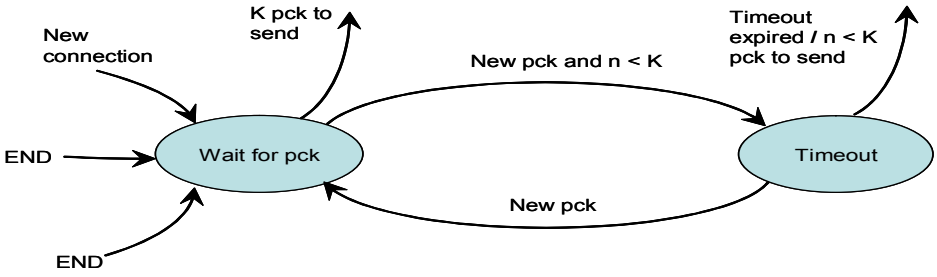

Figura 30 - Store-State.

Su ricezione di un pacchetto, se n < k (con n numero di pacchetti ricevuti), viene settato un timer ed si entra nello stato di timeout. Se non viene ricevuto nessun pacchetto, allo scattare del timer viene passata la finestra di n pacchetti al

TxWmax-state. Se invece viene ricevuto un nuovo pacchetto prima dello scattare del timer, si ripassa nello stato *wait-for-pck*. Se, a questo punto, n è uguale a k, la finestra di pacchetti viene passata al *TxWmax-state*, altrimenti si risetta il timer e si ripassa nello stato *Timeout*.

TxWmax-State - Il Sender, quando si trova in questo stato, trasmette i dati con il protocollo a finestra visto in precedenza, con una finestra pari a Wmax pacchetti. Si possono avere i seguenti eventi:

- Ricezione del messaggio di ELFN: viene settata la dimensione della finestra di controllo di flusso al valore di 1pacchetto e si passa nel *LF-State*.
- Se arrivano *delayAck* (*quickAck*) pacchetti fuori dall'intervallo, si suppone che ci sia stata una *route-changes* e si modifica la legge per il calcolo del RTO. E' possibile che, tra i pacchetti in ritardo (anticipo), ne arrivi uno giusto o in anticipo (ritardo). In questo caso si assume lo stesso una *route-changes*.
- Dopo la ricezione di un numero pari a *rightAck* di ack ricevuti dentro l'intervallo di variazione del RTT, si ripassa a calcolare il RTO con la formula classica.
- Se ci sono *unackedPck* pacchetti non acknoleggiati e in timeout, si suppone che ci sia uno dei seguenti problemi: disconnessione, congestione o eccessiva sottostima del RTO. In questo stato, non si riesce a distinguere le tre situazioni e quindi si assume che ci sia contesa. Si porta la finestra di controllo al valore di 1 pacchetto, si modifica la legge per il calcolo del RTO (la contesa, come visto in precedenza, porta ad una *route-changes*) e si passa in *TxW1-state*.
- Se si stanno trasmettendo gli ultimi Wmax pacchetti del blocco (i restanti pacchetti sono acknoleggiati), e scattano per questi i timeout, la finestra viene settata al valore di 1 pacchetto e si passa nel *TxW1-state*.
- Trasmessi correttamente i pacchetti del blocco, si passa nello *Store-State*.

TxW1-state - Il Sender, quando si trova in questo stato, trasmette i pacchetti con il protocollo a finestra visto in precedenza, con una finestra pari a 1 pacchetto, ed

utilizza, per il calcolo del RTO, la formula modificata. Il Sender passa in questo stato nel caso se ci sono *unackedPck* pacchetti non acknoleggiati e in timeout. Come detto in precedenza, questa situazione viene presa come sintomo di qualche problema lungo la connessione. Quindi, la trasmissione viene effettuata con una finestra di 1 pacchetto, per mitigare una possibile congestione. Le situazioni di sottostima del timeout e di contesa, si distinguono tra di loro per la durata del periodo che il Sender passa senza ricevere ack. In ogni modo, una volta che gli ack cominciano ad arrivare, le azioni da intraprendere sono le stesse: portare la finestra di controllo al valore di Wmax pacchetti e modificare la formula per il calcolo del RTO. La finestra di controllo, su ricezione degli ack, viene riportata al valore di Wmax pacchetti anche dopo la contesa, in quanto, una volta che il percorso è stato ritrovato dal protocollo di routing, si suppone che la congestione sia terminata. Si possono avere i seguenti eventi:

- Su ricezione del messaggio di ELFN si passa nel *LF-State*.
- Su ricezione di *contAck* ack si riporta la finestra al valore di Wmax pacchetti e si passa nel TxWmax-state. In presenza di congestione, la strada che porta gli ack rimane interrotta per diverso tempo. La ricezione degli ack, implica che la strada è stata ripristinata e quindi si prova a rispedire al massimo rate consentito, usando la formula modificata per il calcolo del RTO. In presenza di una *route-changes*, l'arrivo di un ack implica che stanno cominciando ad arrivare gli ack che percorrono il nuovo percorso e quindi si prova a spedire i pacchetti al massimo rate consentito, usando la formula modificata per il calcolo del RTO.
- Trasmessi correttamente i pacchetti del blocco, si passa nello *Store-State*.

LF-State - Questo stato (descritto in figura 31) serve per testare la disponibilità di una strada, dopo la ricezione del messaggio di ELFN che ci informa di una *route-failures*.

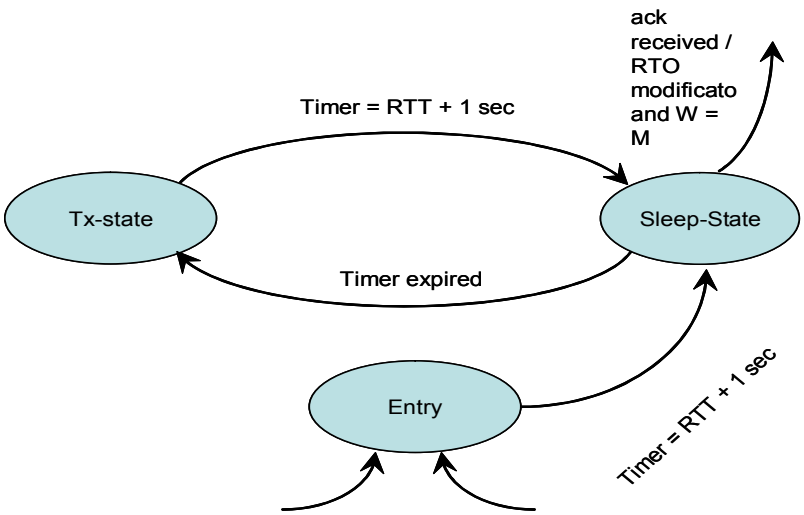

Figura 31 – Link failure state.

Si entra in questo stato ogni volta che il protocollo di routing comincia la ricerca di una nuova strada per andare a destinazione. Solitamente, i protocolli di routing non impiegano meno di T sec per recuperare da una *route-failures*. Quindi, il comportamento del Sender in questo stato è quello di settare un timer al valore di T sec + il RTT dell'ultimo pacchetto spedito e testare la connessione con un pacchetto di *probe* ogni timer secondi, fino alla ricezione di un ack per tale pacchetto. Essendo i path di andata e ritorno asimmetrici, può essere che dopo l'arrivo del messaggio di ELFN continui ad arrivare qualche ack. Questi pacchetti non vanno presi come indicazione del ritrovamento di una strada: si ripassa nel *TxWmax-state*, ponendo la finestra di controllo al valore di Wmax pacchetti e calcolando il RTO con la formula modificata, su ricezione dell'ack per i pacchetti di *probe*.

3.3.2 Receiver TPA

Il compito del receiver è sicuramente più semplice di quello del Sender. Il Receiver si mette in attesa di ricevere i pacchetti dal Sender. Su ricezione di un pacchetto, aggiorna lo stato dei pacchetti ricevuti, costruisce la nuova bitmap (contenente l'indicazione su tutti i pacchetti ricevuti) e spedisce il pacchetto di ack verso la sorgente. Su ricezione del messaggio di ELFN, il Receiver si astiene dallo spedire

gli ack per un periodo di circa Td secondi. Il comportamento del Receiver viene mostrato in figura 32.

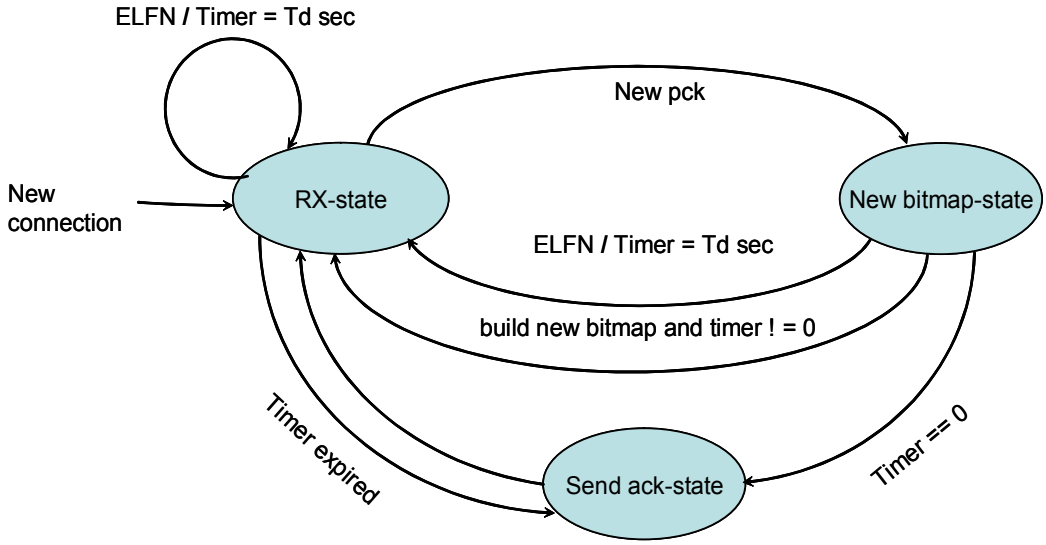

Figura 32 – Diagrammi a stati del Receiver.

Rx-State – In questo stato, il Receiver si mette in attesa di ricevere un pacchetto dati dal Sender. Si possono verificare i seguenti eventi:

- Su ricezione di un pacchetto dati, si passa nel *NewBitmap-State*.
- Allo scattare del Timer si passa nel *SendAck-State* per spedire l'ultimo pacchetto di ack costruito.
- Su ricezione del messaggio di ELFN, viene settato un timer al valore di Td sec.

NewBitmap-State - in questo stato, il Receiver prende il pacchetto, aggiorna la situazioni dei pacchetti ricevuti e calcola la bitmap. Vengono compiute le seguenti azioni:

- Se non è stato ricevuto nessun messaggio di ELFN (cioè il timer è nullo), il Sender passa nel *SendAck-State*.
- Su ricezione del messaggio di ELFN, viene settato un timer al valore di Td secondi e si ripassa nel *Rx-State*.

- Una volta che il timer è stato settato, dopo il calcolo della bitmap per il pacchetto ricevuto, non si ha più il passaggio nel *SendAck-State* (fino a quando il suo valore non torna a zero) e, una volta calcolata la bitmap, si ripassa nel *Rx-State*.

SendAck-State - In questo stato, il pacchetto di ack costruito su ricezione di un pacchetto dati, viene preso e spedito, dopo di che si ripassa nel *Rx-State*.

Capitolo IV.

L'ambiente QualNet

QualNet Developer fornisce un ambiente per progettare protocolli di rete, e mette a disposizione strumenti per creare, visualizzare (anche in maniera animata) ed eseguire esperimenti e per analizzarne i risultati. QualNet Developer è formato da vari elementi, tra cui i principali sono:

- QualNet Simulator
- QualNet Animator
- QualNet Tracer
- QualNet Importer
- QualNet Analyzer
- QualNet Designer

Diamo una breve descrizione dei suddetti componenti.

4.1 QualNet Simulator

Nel seguente capitolo viene data una descrizione dei principi base di funzionamento di QualNet Simulator e vengono presentati in maniera dettagliata i livelli applicazione e trasporto (i due livelli utilizzati per implementare il TPA e l'applicazione che lo utilizza). Per finire viene mostrato come si può settare un esperimento, esaminando il file di configurazione utilizzato da QualNet.

4.1.1 Principi di base

QualNet è un simulatore ad eventi discreti, il cui funzionamento si basa sugli eventi. Un evento è definito come un avvenimento che causa un cambiamento di stato del sistema o l'esecuzione di un'azione specifica da parte di esso. I protocolli progettati per tale simulatore, devono operare come una macchina a stati finiti, re-

agendo in modo opportuno all'occorrenza di un evento. Un esempio d'evento può essere l'arrivo di un pacchetto o lo scattare di un timer.
QualNet Simulator è implementato con un modello a stack del tipo rappresentato in figura 33, dove ciascun protocollo è inserito in uno o più livelli adiacenti dello stack:

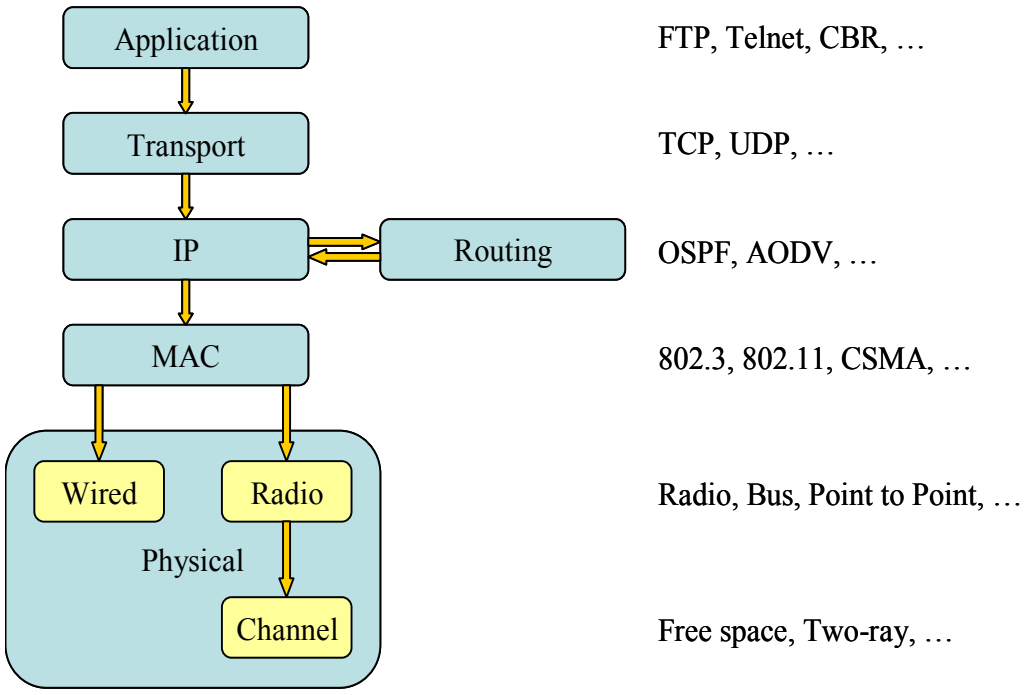

Figura 33 – Modello a stack di QualNet.

L'interfacciamento tra livelli è basato anch'esso sugli eventi: per passare dati o richiedere un servizio ad un livello adiacente, il sistema schedula un evento a quel livello. Un protocollo può creare eventi che vengono gestiti da lui stesso, causandone il cambiamento di stato o l'esecuzione di specifiche funzioni, o eventi che vengono gestiti da altri protocolli di altri livelli dello stack. Il codice dei protocolli, deve essere implementato come un gestore d'eventi che, riceve un messaggio (la struttura dati associata ad un evento) contenente il tipo di evento e i dati associatogli e, in funzione del tipo di evento ricevuto, esegue una particolare azione (ad esempio rispedizione di un pacchetto andato in timeout). La figura 34 mostra

la schematizzazione di un protocollo progettato per essere utilizzato con QualNet, sotto forma di diagramma a stati.

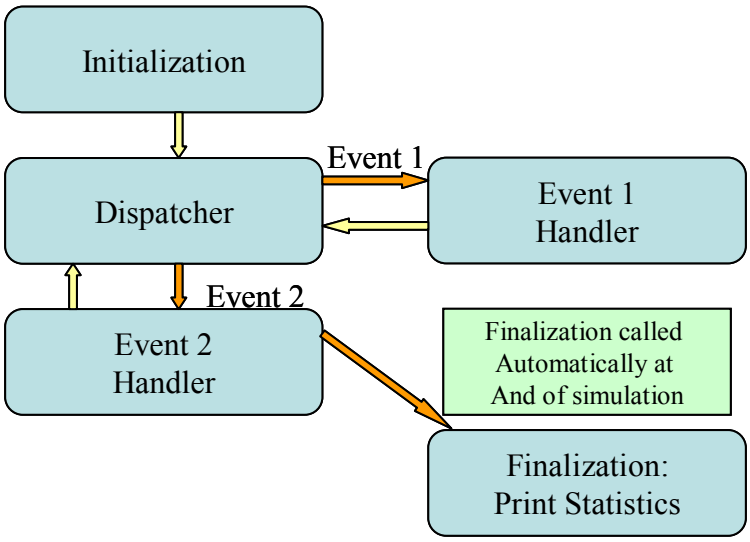

Figura 34 – diagramma a stati di un protocollo progettato per QualNet.

Ciascun protocollo inizia la sua esecuzione con una funzione d'inizializzazione. Tale funzione legge dei parametri esterni che servono per configurare il protocollo. Successivamente, il controllo passa ad un *Event Dispatcher*. Non appena un evento arriva al livello a cui è destinato, QualNet Simulator determina a quale protocollo è indirizzato e chiama l'*Event Dispatcher* di tale protocollo. L'*Event Dispatcher* determina il tipo dell'evento e chiama un appropriato *Event Handler* per processarlo. La *Event Handler function* è una funzione che compie l'azione da intraprendere su ricezione di un particolare evento. Ogni evento ha la propria *event function*. Alla fine della simulazione, viene chiamata la funzione di finalizzazione, il cui compito principale è quello di stampare le statistiche della simulazione. L'evento che causa la transizione nel *finalize state* viene generato automaticamente dal simulatore alla fine della simulazione.

Da quanto visto, ciascun protocollo deve implementare le seguenti tre funzioni:

1 Inizialize.

2 Event Dispatcher.

3 Finalize.

Quando scatta un evento, per farlo pervenire all'*event handling* giusto, il flusso di controllo deve passare attraverso una serie di *Event Dispatcher* function. In particolare, all'occorrenza di un evento, il kernel di QualNet chiama la *dispatcher function* del nodo verso il quale l'evento è scattato. Tale funzione, si trova nel file QUALNET_HOME/addons/seq/node.c e si chiama NODE_ProcessEvent( ). Il suo compito è quello di determinare il livello dello stack a cui è diretto l'evento e di chiamare la funzione LAYER_ProcessEvent( ) di tale livello (ad esempio per il livello applicazione c'è la funzione APP_ProcessEvent( )). La funzione LAYER_ProcessEvent( ), a sua volta, chiama la *event dispatcher* function relativa al protocollo a cui e diretto l'evento. L'*event dispatcher* del protocollo (così come gli altri *event dispatcher)* è formato da una serie di statement switch che servono per distinguere le varie situazioni specifiche che possono capitare durante la vita del protocollo (nello statement switch sono elencati tutti i possibili eventi che possono essere ricevuti dal protocollo in questione). Su ricezione di un messaggio, la *event dispatcher* function chiama l'appropriata *event handler function*.

La struttura dati usata per definire un evento è chiamata *Message* e mantiene le varie informazioni riguardanti l'evento, quali il tipo dell'evento e i dati a lui associati. Tale struttura dati è definita nel file QUALNET_HOME/addons/seq/message.h. In tale file sono contenute anche le dichiarazioni delle funzioni che servono per gestire i messaggi.

In QualNet ci sono due tipi di eventi:

- Packet Events
- Timer Events

Tali eventi sono definiti usando la stessa struttura dati (QUALNET_HOME/addons/seq/message.h) ma si differenziano per il loro scopo e per il modo in cui sono gestiti da QualNet. Esaminiamo in maniera più dettagliata i due tipi d'eventi. I *Packet events* sono usati per simulare la spedizione di pacchetti attraverso la rete. Un pacchetto è definito come un'unità dati, virtuali o reali, appartenenti a qualsiasi livello dello stack. Quando in un nodo della rete deve essere

spedito un pacchetto da un livello dello stack ad un livello adiacente, viene schedulato un *packet event* a tale livello. Ad esempio, la spedizione di un pacchetto dal livello applicazione verso il livello trasporto, genera nel livello trasporto l'evento MSG_TRANSPORT_FromAppSend. Il verificarsi del *packet events* MSG_TRANSPORT_FromAppSend presso il livello in questione, simula l'arrivo del pacchetto. Quando un pacchetto viene spedito da un'applicazione, attraversa in discesa lo stack di protocolli del sending node, attraversa la rete e poi risale lo stack di protocolli del receiving node fino ad arrivare all'applicazione ricevente. Tale procedimento è schematizzato in figura 35:

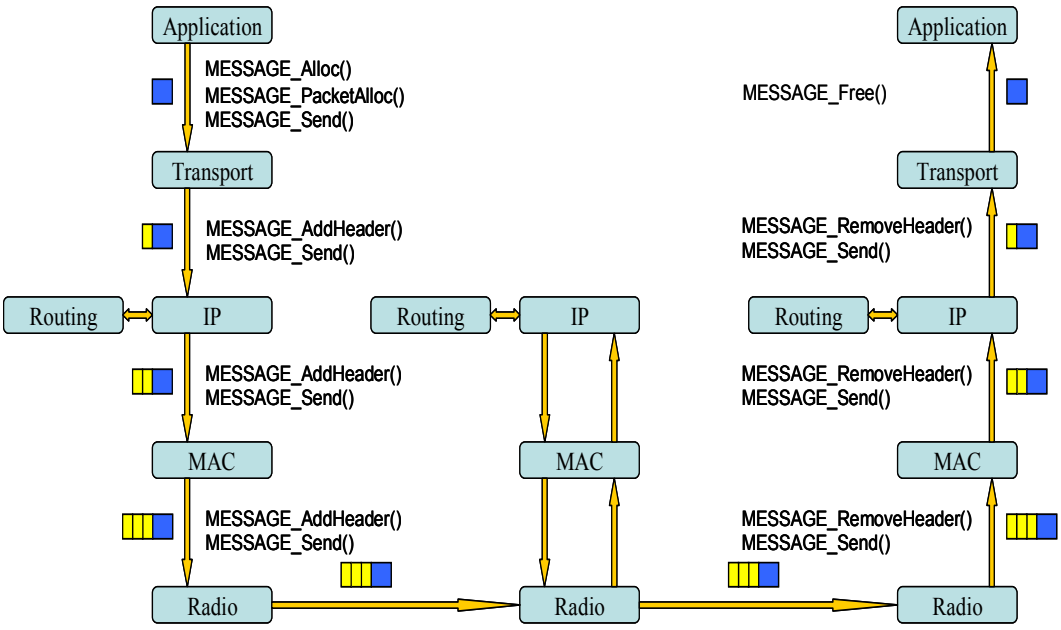

Figura 35 – Ciclo di vita di un pacchetto dati spedito da una applicazione.

A ciascun livello dello stack, al pacchetto spedito viene aggiunto l'header del protocollo attraversato (es: header TCP, header IP....). Ciascun livello è responsabile della spedizione dei pacchetti al livello adiacente. Sul receiving node, gli header aggiunti vengono eliminati dai relativi livelli, fino a ritornare al pacchetto originario, che viene così reso disponibile alla applicazione. Esaminiamo le API che permettono di gestire la spedizione dei pacchetti all'interno della rete (presenti in figura 35).

Per settare un evento (packet o timer), il primo passo da compiere è di allocare il messaggio che lo deve contenere, usando la funzione MESSAGE_Alloc( ). Tale funzione ha i seguenti parametri:

- *node* - il nodo che sta allocando il messaggio.
- *layerType* - il tipo di livello a cui è diretto il messaggio.
- *protocol* - il protocollo a cui è diretto il messaggio.
- *eventType* - il tipo dell'evento a cui si riferisce il messaggio.

I valori per il campo *eventType* sono definiti nel file QUALNET_HOME/include/api.h. A tali valori se ne possono aggiungere di nuovi (è possibile definire degli eventi), inserendo nell'enumerato che elenca tutti i tipi di evento presenti in QualNet, il proprio evento, facendo attenzione ad assegnargli un valore intero non utilizzato da altri eventi. I parametri della funzione MESSAGE_Alloc( ) sono usati dal kernel di QualNet per chiamare la funzione appropriata per gestire il messaggio.

La funzione MESSAGE_PacketAlloc( ) alloca lo spazio per il pacchetto dentro il messaggio (il campo *payload* del messaggio). I suoi parametri sono i seguenti:

- *node* - nodo che sta allocando il messaggio.
- *msg* - puntatore al messaggio per il quale deve essere allocato il pacchetto.
- *packetSize* - la dimensione del pacchetto che deve essere allocato.

La funzione MESSAGE_Send( ) spedisce il messaggio dentro QualNet. I suoi parametri sono i seguenti:

- *node* - nodo che sta spedendo il messaggio.
- *msg* - puntatore al messaggio che deve essere spedito.
- *delay* - ritardo che deve essere sofferto dal messaggio.

Usando tale meccanismo di spedizione, solo il puntatore al messaggio viene spedito attraverso il sistema. Quindi bisogna stare attenti a non modificare il puntatore al messaggio dopo aver chiamato la send.

La funzione MESSAGE_AddHeader( ) aggiunge l'header al pacchetto che si sta spedendo. L'header viene aggiunto in testa al campo *payload* del messaggio. I suoi parametri sono i seguenti:

- *node* - nodo che sta aggiungendo l'header.
- *msg* - puntatore al messaggio a cui va aggiunto l'header.
- *hdrSize* - dimensione dell'header che deve essere aggiunto.

La funzione MESSAGE_RemoveHeader( ) è la funzione chiamata per rimuovere l'header di un pacchetto contenuto in un messaggio. I suoi parametri sono:

- *node* - nodo che sta rimuovendo l'header.
- *msg* - puntatore al messaggio relativo al pacchetto a cui si sta rimuovendo l'header.
- *hdrSize* - dimensione dell'header che si sta rimuovendo.

La funzione MESSAGE_Free( ) libera il messaggio, deallocando tutta la memoria relativa. I suoi parametri sono:

- *node* - nodo che sta cancellando il messaggio.
- *msg* - puntatore al messaggio da cancellare.

Oltre a tali funzioni c'e anche la funzione MESSAGE_ReturnPacket( ), che torna il puntatore al campo *packet* del messaggio.

Un pacchetto può essere creato e spedito a qualsiasi livello. Vediamo, con un esempio, la sequenza d'operazioni da compiere per creare e spedire un messaggio destinato al protocollo UDP del livello trasporto.

```
Message* msg;
msg = MESSAGE_Alloc (node);
MESSAGE_PacketAlloc (node,TRANSPORT_LAYER,TransportProtocol_UDP,
                                MSG_TRANSPORT_FromAppSend);
memcpy (MESSAGE_ReturnPacket (msg), payload, payloadSize);
MESSAGE_Send (node, msg, delay);
```

Dove *payload* è il dato da inserire nel pacchetto e *payloadSize* è la dimensione di tale dato. Nel caso in questione, l'invio del messaggio scatena, dopo delay secondi, l'arrivo presso il protocollo UDP del nodo *node* del messaggio MSG_TRANSPORT_FromAppSend. Per inserire un eventuale header al pacchetto

bisogna compiere le seguenti azioni (supponendo di operare con il protocollo UDP):

```
header* headerPtr;
MESSAGE_AddHeader(node, msg, sizeof (header), TRACE_UDP);
headerPtr = (header*) MESSAGE_ReturnPacket (msg);
hedarPtr = <valore appropriato>;
```

Dove l'ultima istruzione è il riempimento dell'header con i valori appropriati.
La struttura dati *Message* contiene, oltre al campo *payload* e *packet*, un campo chiamato *info* usato per memorizzare informazioni aggiuntive circa il messaggio, oltre a quelle contenute nel campo *payload*. Ogni protocollo, ha le proprie strutture dati da inserire come campo *info* per i messaggi a lui destinati, contenenti tutte le informazioni richieste per gestire il messaggio. Ad esempio, per il messaggio MSG_TRANSPORT_FromAppSend, ci sono le seguenti strutture dati (definite nel file api.h) da inserire come campo *info*: se la spedizione del pacchetto è destinata al protocollo UDP deve essere usata la struttura dati *AppToUdpSend*, altrimenti, se il messaggio è indirizzato al protocollo TCP, deve essere usata la struttura dati *AppToTcpSend*. A tali strutture se ne possono aggiungerne delle altre per i protocolli che si implementano. Ad esempio per il protocollo TPA è stata creata la struttura dati *AppToTpaSend*. Le informazioni memorizzate nel campo *info* attraversano solo un livello dello stack. Guardiamo le API che servono per gestire tale campo.
MESSAGE_InfoAlloc( ): alloca il campo *info* del messaggio. Se il campo *info* era gia stato allocato, il campo nuovo prende il posto di quello vecchio. I suoi parametri sono:

- *node* - nodo che sta allocando il campo info.
- *msg* - puntatore al messaggio per il quale va allocato il campo info.
- *infoSize* - dimensione del campo info.

MESSAGE_ReturnInfo( ): ritorna il puntatore al campo *info*.
MESSAGE_ReturnInfoSize( ): ritorna la dimensione in byte del campo *info*.
Come abbiamo visto, per comunicare tra livelli adiacenti si utilizzano i messaggi e l'insieme d'istruzioni esaminato in precedenza. Per facilitare il compito del programmatore, sono rese disponibili, ai vari livelli, delle funzioni che svolgono il compito di spedire i pacchetti ai livelli adiacenti (permettono all'utente di non scrivere l'insieme d'istruzioni analizzate in precedenza). Ad esempio, al livello applicazione, nel file QUALNET_HOME/apllication/application_util.c sono presenti delle funzioni che permettono di spedire pacchetti al livello trasporto. In particolare ci sono le funzioni che servono per spedire i pacchetti al protocollo TCP e UDP.
I *Timer Events* possono essere usati per implementare dei timers, poiché permettono di schedulare eventi in un tempo futuro. Tali eventi sono settati e ricevuti dallo stesso protocollo e non possono attraversare lo stack di protocolli (come fanno i *Packet Events*). Per settare un *event timer* si deve allocare un nuovo messaggio usando la MESSAGE_Alloc():

```
Message* msg;
msg = MESSAGE_Alloc(...);
```

Dopo aver allocato il messaggio, si deve definire una variabile (di tipo clocktype) a cui va assegnato il numero di secondi, dal tempo corrente della simulazione, dopo il quale il timer deve scattare:

```
clocktype delay;
delay = 5 * SECOND;
```

Eventualmente si deve allocare il campo *info* ed assegnargli il valore voluto:

```
info* timer;
MESSAGE_InfoAlloc(node,msg,sizeof(info);
```

```
timer = (info*) MESSAGE_ReturnInfo(msg);
timer = <valore appropriato>;
```

Dove *info* è una struttura dati che serve per gestire il timer. Come detto in precedenza, tale struttura può essere definita da noi ed inserita nel file api.h e deve contenere tutte le informazioni necessarie al protocollo per gestire correttamente i timer. Una volta creato il messaggio, si deve invocare la funzione MESSAGE_Send( ) per spedirlo dentro QualNet. Tale funzione serve per schedulare l'evento.

```
MESSAGE_Send(node,msg,delay);
```

Se T è l'istante di spedizione del messaggio, a T + delay secondi scatta il timer event allocato. Per cancellare un *Timer Events* dallo schedulatore degli eventi, è possibile utilizzare la funzione MESSAGE_CancelSelfMsg( ). I parametri di tale funzione sono i seguenti:

- *node* - nodo che vuole cancellare il messaggio.
- *msg* - puntatore al messaggio da cancellare.

Dopo aver usato tale funzione non si deve cancellare o riusare il messaggio, poiché la memoria associatagli viene liberata automaticamente.

4.1.2 Livello applicazione

L'application layer risiede al top dello stack di protocolli di QualNet. Tale livello si interfaccia con il transport layer, passando e ricevendo messaggi a questo. Esaminiamo i principali file e cartelle necessarie al funzionamento di questo livello:

- QUALNET_HOME/include/api.h - questo file definisce gli eventi e le strutture dati a loro associate, che vengono utilizzati durante la simulazione. Inoltre, contiene i seguenti tipi di funzioni, definite per i vari livelli dello stack: intialize, finalize, process event.

- QUALNET_HOME/include/application.h - questo file contiene definizioni comuni ai vari protocolli dell'application layer, e le varie strutture dati per la gestione dell'application layer.
- QUALNET_HOME/application - questa cartella contiene i source e header file per le applicazioni definite in QualNet. Il nome del file indica a quale applicazione si riferisce.
- QUALNET_HOME/application/app_util.h - questo file contiene i prototipi delle funzioni contenute nel file app_util.c.
- QUALNET_HOME/include/fileio.h - questo file contiene le strutture dati e le API usate per leggere dal file di input del simulatore.
- QUALNET_HOME/application/application.c - questo file contiene la funzione d'inizializzazione, la funzione che processa i vari eventi e la funzione di finalizzazione dell'application layer.
- QUALNET_HOME/application/app_util.c - questo file contiene le varie utilities usate dalle applicazioni per settare i timer, spedire i pacchetti al transport layer, e aprire le connessioni. Ogni protocollo di livello trasporto (TCP, UDP, …) fornisce in questo file le proprie funzioni per spedire i pacchetti e aprire le connessioni.

Guardiamo adesso come si fa a scrivere e ad aggiungere una nuova applicazione. Il primo passo da fare è quello di creare due file, un header e un source file, contenenti il codice della nostra applicazione. Tali files devono essere messi nella cartella QUALNET_HOME/application. E' buona regola dare ai due file creati un nome che indichi il protocollo a cui si riferiscono: ad esempio l'implementazione della applicazione CBR si trova nei files cbr.c e cbr.h.

Il secondo passo è quello di andare a modificare i files che gestiscono l'application layer. Guardiamo come.

QUALNET_HOME/include/application.h - La applicazione da noi creata deve essere aggiunto alla lista delle varie applicazioni presenti nello stack. Questo va fatto operando sul file QUALNET_HOME/include/application.h. Tale file, contiene l'enumerato *AppType* che elenca tutte le applicazioni presenti. Un'applicazione è

formata normalmente da due parti: il client che genera il traffico e il server che lo riceve. Sia il client che il server dell'applicazione, devono essere aggiunti all'enumerato *AppType*. E' consigliabile usare il seguente formato per inserire le due parti dell'applicazione:

APP_yourAppName_CLIENT: per il cliente
APP_yourAppName_SERVER: per il server

I nuovi nomi vanno aggiunti alla fine della lista.
QUALNET_HOME/application/application.c - In tale file si deve aggiungere la dichiarazione del nuovo protocollo e vanno modificate alcune funzioni. La funzione APP_ProcessEvent( ) implementa uno statement switch sul nome del protocollo, letto dal messaggio ricevuto, e va modificata in maniera che sia in grado gestire gli eventi del nuovo protocollo. A tale statement va aggiunto del codice per chiamare, su ricezione di un evento a lei destinato, l'*event dispatcher* della nostra applicazione. Questo va fatto separatamente per il server ed il client. Per modificare tale funzione si può seguire l'esempio dell'applicazione FTP. La funzione APP_InizializeApplications() legge i parametri di configurazione dell'applicazione, specificati nel file di configurazione, e li passa alla funzione d'inizializzazione del protocollo a cui sono destinati. A tale funzione dobbiamo aggiungere il codice che permette di leggere i parametri di configurazione della nostra applicazione e di passarli alla sua funzione d'inizializzazione. Guardiamo tale funzione, con riferimento all'applicazione CBR: inizialmente viene fatta una sscanf( ) per leggere i parametri della applicazione. Successivamente viene chiamata la funzione IO_AppParseSourceAndDestStrings( ) per recuperare l'indirizzo del client e del server. Poi, tramite la funzione MAPPING_GetNodePtrFromHash( ) vengono ricavate le strutture dati del nodo dove risiede il server e del nodo dove risiede il client e vengono chiamate le funzioni di inizializzazione di entrambi. Per inserire il nostro protocollo bisogna seguire l'esempio del CBR, modificandolo in funzione dei parametri della nostra applicazione. Nel file di configurazione (*.app) contenente i parametri dell'applicazione, va indicato il nome

dell'applicazione (es: CBR). Tale nome è quello che deve essere usato nello statement else if( ). La funzione APP_Finalize() va modificata per metterla in grado di chiamare la funzione di finalizzazione del nostro protocollo. Basta aggiungere allo statement switch, il *case* relativo al nuovo protocollo e chiamare dentro di esso le funzione di finalizzazione del client e del server. Basta seguire l'esempio del protocollo FTP.

L'ultimo passo da fare per il corretto inserimento del protocollo in QualNet è la modifica del file QUALNET_HOME/main/Makefile-common, per includervi i files creati. Questo assicura la compilazione del nuovo protocollo.

Per rendere funzionante il protocollo aggiunto, bisogna ricompilare QualNet, eseguendo i seguenti comandi:

- Andare nella directory QUALNET_HOME/main
- Lanciare il comando make –f Makefile-linux

Diamo ora un rapido sguardo a come va strutturato il file sorgente di una applicazione. Per un esempio concreto di come organizzare un'applicazione, vedere il capitolo 5.1.

Il primo passo da fare per creare il file sorgente della nostra applicazione, è quello di includere i seguenti files:

```
#include <stdio.h>
#include "api.h"
#include "app_util.h"
#include"ip.h"
```

In seguito devono essere implementate le funzioni che simulano il comportamento dell'applicazione. Come gia detto, ci sono tre funzioni che vanno implementate per far funzionare un protocollo: la init( ), la event dispatcher( ) e la finalize( ). Esaminiamole in dettaglio.

Init( ). In generale, vanno implementate due funzioni d'inizializzazione: una per il client ed una per il server. I compiti principali delle due init( ) sono i seguenti:

- Inizializzare lo stato dell'applicazione e registrare i parametri di configurazione.
- Schedulare un evento per far partire l'applicazione.
- Inizializzare le varie strutture dati e variabili, allocare la memoria necessaria, settare i valori di defoult, etc.

Nel file di header dell'applicazione, deve essere dichiarata la struttura dati che descrive lo stato dell'applicazione. Per ogni istanza di una applicazione, deve essere allocata la struttura dati che ne descrive lo stato. Nella init( ) viene creata una istanza della applicazione, allocando la memoria per contenere la sua struttura dati (MEM_malloc()). Tale struttura dovrà essere inizializzata usando i parametri della init( ) e i valori di defoult definiti per essi. Una volta creata la struttura dati, bisogna registrare l'istanza della applicazione nella lista delle applicazioni che girano sul nodo dove l'istanza della applicazione risiede. Questo è fatto tramite la funzione APP_RegisterNewApp() (file app_util.h). Tale funzione ha tre parametri: il puntatore al nodo, il tipo d'applicazione e il puntatore alla struttura dati della applicazione. Quando bisogna accedere alla struttura dati di una specifica istanza della applicazione, bisogna ripescarla dalla lista in cui è stata inserita. Tale lista, è mantenuta nella struttura dati *AppInfo*, definita nel file QUALNET_HOME/include/applicatio.h. Per ripescare la struttura dati dell'applicazione che ci interessa, bisogna identificarla in qualche maniera, ad esempio attraverso il numero di porta o l'ID della connessione appartenente alla applicazione. La variabile node->appData.appPtr contiene la lista suddetta.

Guardiamo come creare e inizializzare un timer da utilizzare in un'applicazione. Dato che ciascun nodo può avere più istanze contemporanee della stessa applicazione, l'*event timer* del livello applicazione, deve avere il campo *info* per determinare a quale istanza della applicazione va recapitato il messaggio. Nel file QUALNET_HOME/include/apllication.h è definita la struttura dati AppTimer che serve per gestire i timer. Tale struttura ha i seguenti campi:

- *type* - tipo di timer.
- *connectionId* - identificativo della connessione.

- *sourcePort* - numero di porta.

I tipi di timer per l'applicazione, sono definiti nell'enumerato AppTimerType del file QUALNET_HOME/include/apllication.h. La struttura dati AppTimer deve essere usata come campo *info* del messaggio di timer. Il tipo di tale messaggio va definito come MSG_APP_TimerExpired. Per facilitare il compito di settare un timer (evitare le fasi viste per settare un timer) c'è l'API APP_SetTimer( ). Tale funzione è implementata nel file ./application/app_util.c.

Event Dispatcher**.** Tale funzione ha il compito di gestire i vari messaggi che arrivano alla nostra applicazione. Ci saranno, in generale, due event dispatcher: uno per il server e uno per il client. Entrambi si devono comportare nel modo seguente: su ricezione di un messaggio, devono determinare a quale istanza dell'applicazione è destinato. Questo può essere fatto attraverso il campo *info* del messaggio (conterrà l'identificativo dell'istanza a cui è destinato). Una volta determinata l'istanza dell'applicazione di destinazione, si deve esaminare il tipo di evento del messaggio, e gestirlo di conseguenza. Questo viene fatto attraverso uno statement switch che elenca tutti gli eventi presi in considerazione dall'applicazione. Una volta gestito il messaggio ricevuto, bisogna liberare la memoria da lui occupata, tramite la funzione MESSAGE_Free( ).

Finalize. In questa funzione, deve essere chiamato il codice per stampare le statistiche della simulazione. Ciascun protocollo ha le sue statistiche da definire. Per fare questo, vanno create delle apposite variabili per contenere tali statistiche. Ad esempio, si può creare una struttura dati (nel file header) avente per campi le statistiche che si vogliono collezionare, ed inserirla come campo nella struttura dati che definisce lo stato del protocollo (ogni istanza del protocollo deve avere le proprie variabili per le statistiche). La variabile che definisce le statistiche, deve essere inizializzata nella init( ) function. Durante la vita del protocollo, le statistiche vanno gestite nel modo opportuno: ad esempio, se si definisce la statistica "pacchetti ricevuti", su ricezione di ogni pacchetto va incrementata la relativa variabile. Va inoltre creata una funzione in grado di stampare le statistiche collezionate. Questa funzione, deve essere chiamata nella finalize function(). Per stampare

le statistiche bisogna creare una stringa contenente le informazioni da stampare (nome e valore della statistica). Successivamente, la statistica va stampata nel file .stat (prodotto al termine della simulazione), facendo una chiamata alla funzione IO_PrintStat( ) (file QUALNET_HOME/main/fileio.c). Questa funzione, richiede i seguenti parametri:

- Puntatore al nodo che sta stampando le statistiche.
- Livello a cui appartengono le statistiche (nel nostro caso va settata ad Application).
- Stringa contenente il nome del protocollo.
- Indirizzo dell'interfaccia: se non necessario va settato ad ANY_DEST.
- Identificatore dell'istanza del protocollo.
- Stringa contenente la statistica.

Per ultimo, guardiamo come gestire la spedizione e la ricezione dei messaggi. Per gestire i messaggi, si possono usare due strade alternative: utilizzare le API definite nel file app_util.c o usare le API che gestiscono i messaggi (message.c). Ad esempio, per spedire un dato al protocollo UDP, possiamo mandare un messaggio al livello trasporto indirizzato al protocollo UDP (il tipo del messaggio deve essere MSG_TRANSPORT_FromAppSend), settando il campo *info* del messaggio con una istanza della struttura dati AppToUdpSend, oppure possiamo usare la funzione APP_UdpSendNewData( ).

4.1.3 Livello trasporto

Il transport layer fornisce il servizio di trasportare i messaggi tra il client ed il server dell'applicazione e si interfaccia con il livello applicazione ed il livello network, permettendo lo scambio di messaggi tra questi due livelli. Esaminiamo i principali file e cartelle necessari al suo funzionamento:

- QUALNET_HOME/include/api.h - questo file definisce gli eventi, e le strutture dati ad essi associate, che vengono utilizzati durante la simulazione. Inoltre, contiene i seguenti tipi di funzioni, definite per vari livelli dello stack: intialize, finalize, process event.

- QUALNET_HOME/include/transport.h - questo file contiene le definizioni comuni ai vari protocolli di livello trasporto e le strutture dati necessarie per gestire il livello trasporto presso i vari nodi.
- QUALNET_HOME/include/fileio.h - questo file contiene le strutture dati e le API usate per leggere dal file di input del simulatore.
- QUALNET_HOME/transport - questa cartella contiene i source e header files relativi ai vari protocolli di trasporto (come UDP e TCP). Il nome del file indica a quale protocollo si riferisce il file. Ad esempio, l'implementazione del protocollo UDP si trova nei files udp.c e udp.h.
- QUALNET_HOME/transport/transport.c - questo file contiene la funzione d'inizializzazione, l'event dispatcher e la funzione di finalizzazione del transport layer.

Guardiamo adesso i passi principali da seguire per implementare ed inserire in QualNet un nuovo protocollo di livello trasporto. Il primo passo consiste nel creare i files che devono contenere il codice del nuovo protocollo e nell'inserirli nella directory QUALNET_HOME/transport/. Tali file vanno nominati in modo da indicare il protocollo a cui si riferiscono.

Il secondo passo è quello di modificare i files che gestiscono il livello trasporto. Guardiamo come.

QUALNET_HOME/include/transport.h - Tale file contiene l'enumerato TransportProtocol che elenca tutti i protocolli di livello trasporto. A tale enumerato, va aggiunto anche il nostro protocollo, in ultima posizione, usando il nome TRANSPORT_PROTOCOL_newProtocol. In tale file c'è anche la struttura dati struct_transport_str, contenente i puntatori alle strutture dati relative ai vai protocolli presenti su un particolare nodo. A tale struttura va aggiunto il puntatore alla struttura dati relativa al nuovo protocollo. Inoltre, se il protocollo non è obbligatorio, e vale a dire se si può decidere di attivarlo o disattivarlo mediante il file di configurazione, va aggiunto un flag booleano che indica se il relativo protocollo è attivo o meno. Come esempio, si può seguire il protocollo SRM.

QUALNET_HOME/transport/transport.c - Tale file contiene le funzioni d'inizializzazione, di event dispatcher e di finalizzazione del livello trasporto. Tali funzioni vanno modificate per inserirvi il nuovo protocollo. I protocolli possono essere configurati attraverso dei parametri specificati dall'utente all'interno del file di configurazione del simulatore. Per implementare correttamente il protocollo, bisogna decidere il suo formato per l'input (per specificare i parametri di configurazione), e il suo nome. I parametri del protocollo, devono essere inseriti nel file di configurazione in modo da indicare il protocollo a cui si riferiscono. Il formato è il seguente:

<keyword> <value>

dove *keyword* identifica il protocollo (nome del protocollo) e *value* il parametro che si vuole settare. Se il protocollo non è obbligatorio, bisogna aggiungere la seguente entrata nel file di configurazione:

TRANSPORT-PROTOCOL-<protocol-name> <stato>

Dove *protocol-name* è il nome del protocollo che si vuole attivare, e *stato* indica se il protocollo deve essere abilitato (YES) o meno (NO). La funzione TRANSPORT_Initialize( ) è responsabile di leggere e memorizzare, in appropriate strutture dati, i parametri di configurazione per l'attivazione dei vari protocolli, e di chiamare le loro funzioni d'inizializzazione. Per inserire un nuovo protocollo, tale funzione va modificata. Se il nuovo protocollo è obbligatorio, basta seguire l'esempio del TCP e chiamare la funzione d'inizializzazione. Altrimenti, va letto il file di configurazione (con la funzione IO_ReadString( )) per vedere se il protocollo deve essere attivato e, se necessario, va chiamata la sua funzione d'inizializzazione. Inoltre, va aggiornato lo stato dei vari protocolli di livello trasporto, contenuto nella variabile node->transportData. Per fare questo basta seguire l'esempio del protocollo SRM.

La funzione TRANSPORT_ProcessEvent( ) è l'event dispatcher del livello trasporto. Questa va modificata per metterla nella condizione di gestire gli eventi relativi al nuovo protocollo. I messaggi, contengono il nome del protocollo a cui sono destinati. Tale nome deve essere elencato nell'enumerato TransportProtocol visto prima. La processEvent implementa uno switch su tale nome e chiama l'event dispatcher relativo al protocollo a cui è destinato il messaggio. Basta inserire il case relativo al nuovo protocollo, seguendo l'esempio del TCP nel caso in cui il protocollo sia obbligatorio, o l'esempio di SRM in caso contrario.
La funzione TRANSPORT_Finalize( ) è eseguita al termine della simulazione e ha il compito di chiamare le funzioni di finalizzazione dei vari protocolli. La chiamata alla funzione di finalizzazione del nuovo protocollo deve essere inserita in tale funzione, seguendo, come nei casi precedenti, l'esempio del TCP o di SRM.
QUALNET_HOME/network/ip.c - In tale file va aggiunta una funzione per mettere il livello IP in grado di inoltrare, verso il nuovo protocollo di trasporto, i pacchetti a lui destinati. Per fare questo, si può seguire l'esempio della funzione SendToUdp( ), definita per spedire pacchetti verso il protocollo UDP. Tale funzione, implementa tutte le operazioni da fare per spedire il messaggio al protocollo UDP e ha i seguenti parametri:

- *node* - puntatore alla struttura dati del nodo che sta eseguendo la *sendToUdp*.
- *msg* - puntatore al messaggio da passare all'UDP.
- *priority* - priorità del messaggio.
- *sourceAddress* - indirizzo IP della sorgente del messaggio.
- *destinationAddress* - indirizzo IP della destinazione del messaggio.
- *incomingInterfaceIndex* - indice della interfaccia di ingresso del messaggio.

La funzione DeliverPacket( ) del file ip.c, determina il tipo di protocollo a cui è destinato il messaggio e, tramite uno statement switch, chiama l'apposita funzione per recapitarlo a destinazione. Basta inserire il case relativo al nostro protocollo (case IPPROTO_newProtocol), seguendo l'esempio dell'UDP. Per ultimo va mo-

dificato il file QUALNET_HOME/network/ip.h, inserendovi la seguente definizione:

```
#define IPPROTO_newProtocol          17
```

Bisogna stare attenti ad assegnare ad IPPROTO_newProtocol un numero che non è ancora stato assegnato a nessun altro protocollo.
Per finire va modificato il file QUALNET_HOME/main/Makefile-common per includervi i files creati. Questo assicura che il nuovo protocollo verrà compilato.
Per rendere funzionante il protocollo aggiunto, bisogna ricompilare QualNet, eseguendo i seguenti comandi:

- Andare nella directory QUALNET_HOME/main
- Lanciare il comando make –f Makefile-linux

Diamo ora uno sguardo a come va strutturato il file sorgente di una applicazione. Per un esempio concreto sì può esaminare il capitolo 4.2.
Il primo passo da fare per creare il file sorgente della nostra applicazione, è quello di includere le seguenti dichiarazioni:

```
#include <stdio.h>
#include “api.h”
#include “ip.h”
```

Successivamente, vanno implementate le tre funzioni principali che permettono il funzionamento di un protocollo in QualNet: la init(), la event dispatcher() e la finalize(). Esaminiamole in dettaglio.
La init( ) è responsabile della inizializzazione del protocollo. Comunemente deve eseguire le seguenti azioni:

- Creare un’istanza della struttura dati del protocollo. La struttura dati di un protocollo, serve per memorizzare i parametri di configurazione e altre informazioni necessarie al corretto funzionamento (TransportDataNewProtocol).

- Inizializzare lo stato del protocollo e registrare i parametri di configurazione. La init deve leggere dal file di configurazione, le varie informazioni di configurazione del protocollo e registrarle nella istanza della struttura TransportDataNewProtocol. Per leggere i parametri è resa disponibile la funzione IO_ReadString( ).

Esaminiamo il comportamento della funzione d'inizializzazione del protocollo UDP (file QUALNET_HOME/transport/udp.c). Tale funzione (TransportUdpInit()) svolge le seguenti azioni:

- Crea un'istanza della struttura dati del protocollo UDP, usando la funzione MEM_malloc( ) e la fa puntare dalla variabile node->transportData.udp.
- Guarda se le statistiche sono abilitate per il protocollo.
- Aggiorna la struttura dati relativa all'istanza dell'UDP, inizializzando la struttura dati relativa alle statistiche.

La Event dispatcher( ) implementa una serie di statement switch per chiamare la funzione appropriata per gestire l'evento ricevuto. Tale switch deve contenere tutti gli eventi gestiti dal protocollo. Prima di gestire un evento, si deve determinare a quale istanza del protocollo è destinato. Per un esempio di tale funzione, si può esaminare l'event dispatcher del protocollo UDP (TransportUdpLayer( ) definita nel file udp.c).

La Finalize( ) function è responsabile di stampare le statistiche relative al protocollo e di deallocare la eventuale memoria non ancora deallocata e relativa al nostro protocollo. Ad esempio, la funzione di finalizzazione dell'UDP (TransportUdpFinalize( )) svolge le seguenti funzioni: recupera la struttura dati dell'UDP e guarda se le statistiche sono abilitate o meno. Se sono abilitate, stampa i valori delle due statistiche definite per l'UDP: numPckToApp e numPckFromApp.

Guardiamo anche i meccanismi che si possono utilizzare per lo scambio di messaggi al livello trasporto. Per spedire i messaggi verso il livello applicazione, si usano le API che gestiscono i messaggi, definite nel file QUALNET_HOME/main/message.c. Come esempio, si può prendere la funzione TransportUdpSendToApp( ) dell'UDP, responsabile di spedire i pacchetti verso

l'applicazione. Tale funzione, prende il messaggio ricevuto dal livello IP e ne setta il livello di destinazione, il protocollo di destinazione e il tipo di evento, dopo di che alloca il campo *info* del messaggio, usando la struttura dati UdpToAppRecv. Tale struttura contiene i campi necessari per identificare il socket del pacchetto. A questo punto, il messaggio viene spedito a destinazione.

Per spedire i pacchetti al livello network, possono essere usate o le API che gestiscono i messaggi, definite nel file message.c, o le API definite nel file ip.c, come la NetworkIpReceivePacketFromTransportLayer( ). Per vedere come si utilizza tale funzione si può esaminare la funzione TransportUdpSendToNetwork( ), usata dall'UDP per spedire pacchetti al protocollo IP (udp.c). Tale funzione, prende il messaggio proveniente dall'applicazione, recupera il campo *info*, aggiunge l'header al messaggio e lo spedisce al livello network usando l'API fornita dal livello IP.

Come ultima cosa, guardiamo come collezionare le statistiche per il protocollo creato. Il protocollo, deve allocare lo spazio necessario per contenere le statistiche nella propria struttura dati, in modo da potervi salvare le informazioni raccolte durante il corso della simulazione. Nel caso dell'UDP, viene definita la struttura dati TransportUdpStat (contenente le varie statistiche), e nella struttura dati che descrive lo stato dell'UDP viene inserito un puntatore a tale struttura dati. Nella funzione d'inizializzazione, si deve allocare lo spazio per la struttura TransportUdpStat e si devono aggiornare in modo opportuno i campi dell'istanza della struttura TransportDataUdpStruct. Per finire va creata ed inserita nella finalize function, una funzione per stampare le statistiche.

4.1.4 Settare un esperimento

Guardiamo adesso come configurare un esperimento in QualNet. Per configurare un esperimento, si deve usare un file di configurazione (nomeFile.config) contenente tutte le informazioni necessarie per la sua esecuzione. Il file di configurazione, è un file di testo formato da varie righe, ciascuna delle quali contenente un

parametro della simulazione. Tali parametri possono essere specificati nella seguente forma:

[Qualifier] VARIABLE [Instance] VALUE

I campi *VARIABLE* e *VALUE* rappresentano il nome del parametro ed il suo valore. Il campo *qualifier* è usato per indicare i nodi della rete per i quali il valore del parametro è valido. Il campo *instance* permette di specificare valori multipli per il parametro in questione. Sia il campo *qualifier* che il campo *instance* sono opzionali. Il valore di un parametro, può essere di vari tipi: *string*, *integer*, *double*, *float*, e *clocktype*. Un esempio di file di configurazione, contenente tutti i parametri per i protocolli forniti con QualNet, si può trovare nel file QUALNET_HOME/bin/default.config.

Esaminiamo adesso alcuni dei principi base che si devono conoscere per poter costruire un file di configurazione.

Per inserire un commento nel file basta far precedere la riga di commento dal carattere **#**.

Il tempo, nel file di configurazione, deve essere specificato nel *QualNet Time Format*. Più precisamente, il tempo deve essere espresso mediante un valore numerico fatto seguire da dei caratteri che specificano l'unita di tempo del numero in questione. La tabella seguente mostra come specificare le varie unità di tempo:

QualNet Time Format	significato
100NS	100 nanosecondi
100US	100 microsecondi
100MS	100 millisecondi
100S	100 secondi
100M	100 minuti
100H	100 ore
100D	100 giorni

Un nodo, in QualNet, rappresenta ogni dispositivo che si connette ad una rete, come radio devices, desktop computers, routers, e laptop computers. Ciascun nodo, ha un identificatore (*Node Identifier* o *nodeId*) rappresentato da un intero positivo. Il *NodeId* è usato, nei vari file di configurazione, per riferirsi ad un particolare nodo. I nodi possono avere una o più interfacce di rete, ciascuna con il proprio indirizzo IP e con la propria *subnet mask*.
Un esperimento è composto da varie reti di nodi. Per creare tali reti, l'utente deve descrivere nel file di configurazione le SUBNET e i vari LINK. Un LINK, è una SUBNET con due nodi collegati ad essa tramite un link full duplex (point-to-point link). Per definire il *network address* e la *subnet mask* di una rete si usa la seguente sintassi (*QualNet Subnet Shorthand*):

N<host bits>-<network address with significant digits only>

Per esempio, la definizione N8-1.0 specifica che l'*host address* è formato 8 bit (la subnet mask vale 255.255.255.0) e il *network address* è 0.0.1.0 (N8-1.0 e N8-0.0.1.0 sono equivalenti). Il metodo di specifica *QualNet Subnet Shorthand* può essere usato per applicare ad una rete specifica un certo parametro di configurazione.
Il parametro SUBNET serve per definire una subnet ed ha due argomenti:

- IL *QualNet Subnet Shorthand* per specificare il network address e la subnet mask.
- Il *nodeIds* dei nodi che compongono la rete.

Prendiamo l'esempio seguente:

SUBNET N8-1.0 { 1, 2, 3 thru 10 }

Con tale parametro viene creata una subnet avente il *network address* e la *subnet mask* definita da N8-1.0, e formata dai nodi aventi il *nodeId* che va da 1 a 10. QualNet, assegna automaticamente l'*IP address* ai vari nodi, in funzione del

network address della *subnet*. Ai dieci nodi sono assegnati gli indirizzi *network-address*.1 fino a *network-address*.10. L'utilizzo della parola chiave *thru*, permette di non enumerare tutti i nodi da 1 a 10. In quest'esempio, al nodo 1 viene assegnato l'*IP address* 0.0.1.1, al nodo 2 viene assegnato l'*IP address* 0.0.1.2 e al nodo 10 viene assegnato l'*IP address* 0.0.1.10.

Esaminiamo adesso i *point-to-point link*. La parola chiave LINK, è usata per descrivere un *ponit-to-point link* che connette esattamente due nodi. Inoltre, è un parametro che serve per settare le interfacce di rete dei nodi da connettere al link. Ha due argomenti:

- Il *QualNet Subnet Shorthand* per specificare il network address.
- I due *nodeId* dei nodi che verranno connessi tramite il point-to-point link.

Il formato per tale parametro è lo stesso di quello utilizzato per il parametro SUBNET.

Associati con la dichiarazione del LINK, per definirne le proprietà, ci sono vari parametri di configurazione. Esaminiamone alcuni con un esempio:

```
LINK-BANDWIDTH     1544000
LINK-PROPAGATION-DELAY     1MS
LINK-RETURN-BANDWIDTH     1000000
LINK-RETURN-PROPAGATION-DELAY     5MS
```

Il LINK-BANDWIDTH definisce la banda, in bits per secono, del link. Se il LINK-RETURN-BANDWIDTH non è specificato, il link viene considerato simmetrico. Se è specificato, il LINK-RETURN-BANDWIDTH definisce la banda, in bits per secondo, dal secondo *nodeId* al primo, creando un link asimmetrico.

Il LINK-PROPAGATION-DELAY specifica il tempo, in *QualNet Time Format*, che impiega un bit per attraversare il link. Se il LINK-RETURN-PROPAGATION-DELAY non è specificato, il link ha lo stesso propagation delay in entrambi i sensi di propagazione. Altrimenti, il LINK-RETURN-PROPAGATION-DELAY specifica il propagation delay dal secondo nodo al primo. Un esempio d'utilizzo della parola chiave LINK è dato qui sotto:

```
LINK N2-1.0 { 1 , 2 }
LINK N2-2.0 { 2 , 3 }

LINK-BANDWIDTH     1544000
LINK-PROPAGATION-DELAY     1MS

[N2-2.0]     LINK-BANDWIDTH     112000
[N2-2.0]     LINK-PROPAGATION-DELAY     50MS
```

Con i parametri sopra elencati, vengono creati due link: il primo collega i nodi 1 e 2. Il secondo collega i nodi 2 e 3. QualNet assegna automaticamente l'*IP address*, in funzione del *network address*, ai vari nodi. Per il primo link, al nodo 1 viene assegnato l'*IP address* 0.0.1.1 e al nodo 2 viene assegnato l'*IP address* 0.0.1.2. Per il secondo link, al nodo 2 viene assegnato l'*IP address* 0.0.2.1 e al nodo 3 viene assegnato l'*IP address* 0.0.2.2.

Il LINK-BANDWIDTH e LINK-PROPAGATION-DELAY, senza campo *qualifier*, specificano i valori di defoult per le proprietà dei point-to-point links.

Le linee "[N2-2.0] LINK-BANDWIDTH 112000" e "[N2-2.0] LINK-PROPAGATION-DELAY 50MS", indicano che il link avente la subnet 0.0.2.0 opera a 112 kbps con 50 ms di *propagation delay* (ISDN speed), in entrambe le direzioni.

L'utente può settare, per ogni protocollo utilizzato (*protocol-based parameter*) nella simulazione (MAC-PROTOCOL, ROUTING-PROTOCOL, MULTICAST-PROTOCOL, etc.), i relativi parametri. Per restringere la specifica dei *protocol-based parameter* ad una particolare *subnet*, deve essere settato il campo *qualifier* del parametro, utilizzando l'indirizzo della *subnet*. Guardiamo un esempio:

```
[N8-1.0] MAC-PROTOCOL MAC802.11
MAC-PROTOCOL CSMA
```

Con tale configurazione, la *subnet* N8-1.0 usa il MAC-PROTOCOL MAC802.11, mentre tutte le altre *subnets* usano il CSMA. Un parametro senza il campo *Qualifier* è usato come valore di default.
Per restringere la specifica dei *protocol-based parameter* ad un nodo singolo o ad un insieme di nodi, bisogna seguire l'esempio seguente:

```
[0.0.1.1   0.0.1.2   3   4] IP-QUEUE-TYPE RED
IP-QUEUE-TYPE FIFO  First-In First-Out;
```

Qui vengono settate le interfacce dei nodi aventi l'*IP address* 0.0.1.1, e 0.0.1.2, e i nodi con *nodeId* 3 e 4, per usare il meccanismo di gestione delle code denominato *Random Early Detection*. Tutti gli altri nodi usano il meccanismo *FIFO*.
Il parametro VERSION specifica la versione del file di configurazione utilizzato.

```
VERSION 3.6
```

Il parametro EXPERIMENT-NAME specifica il nome dell'esperimento descritto dal file di configurazione. Tale nome è usato come prefisso per il file contenente le statistiche prodotte alla fine della simulazione (experiment-name.stat). Vediamo un esempio di utilizzo di tale parametro.

```
EXPERIMENT-NAME myExperiment
```

Il parametro SIMULATION-TIME serve per definire il tempo massimo di simulazione. Deve essere specificato nel *QualNet Time Format*. Dopo che la simulazione è durata per il tempo indicato, termina e vengono registrate tutte le statistiche collezionate. Vediamo un esempio di utilizzo di tale parametro.

```
SIMULATION-TIME 100M
```

Il parametro SEED è un random number seed per generare numeri random. Specificare gli stessi parametri seed per due simulazioni differenti, garantisce che i risultati del simulatore saranno esattamente gli stessi, indipendentemente da quante volte viene eseguito lo scenario specificato. Variando il seed e mediando le metriche ottenute su più esecuzioni, si può ridurre il numero di outliers nell'analisi statistica del comportamento della rete. Vediamo un esempio di utilizzo di tale parametro.

SEED 1

Il parametro COORDINATE-SYSTEM, specifica il sistema di coordinate da utilizzare nell'esperimento. Le coordinate possono essere numeri in floating-point. Facciamo un esempio di come si può specificare la posizione di un nodo:

(0,0)

(0,0,0)

(-34.00, 115.00, 220.5)

L'altitudine è assunta pari a zero se omessa.
I sistemi di coordinate presenti in QualNet sono:

1. LATLONALT- sistema di coordinate standard per la latitudine, la longitudine e l'altitudine.
2. CARTESIAN- Con questo sistema le coordinate devono essere date come valori positivi, e rappresentano la distanza in metri dall'origine (0, 0, 0).

Per entrambi i sistemi di coordinate, la coordinata Z (altitudine) è specificata in metri sopra il livello del mare (valori negativi indicano che siamo sotto il livello del mare) e può essere omessa. Vediamo un esempio di utilizzo di tali parametri.

COORDINATE-SYSTEM CARTESIAN

COORDINATE-SYSTEM LATLONALT

I seguenti parametri specificano le coordinate degli angoli southwest e northeast del terreno usato nella simulazione, quando si usa il sistema di coordinate LAT-LONALT. Tutte le posizioni di un nodo avranno dei valori compresi tra l'angolo southwest e l'angolo northeast, per le loro coordinate di latitudine e di longitudine.

TERRAIN-SOUTH-WEST-CORNER (30.00, 40.00)
TERRAIN-NORTH-EAST-CORNER (30.01, 40.01)

Il parametro TERRAIN-DIMENSIONS specifica la dimensione del terreno dove avviene la simulazione, usando le coordinate cartesiane. L'unita di misura sono i metri.

TERRAIN-DIMENSIONS (2000, 2000)

Il parametro NODE-PLACEMENT serve per stabilire dove vanno posizionati i nodi che prendono parte all'esperimento. Per un posizionamento automatico dei nodi bisogna selezionare il valore RANDOM, UNIFORM, o GRID. Per posizionare i nodi manualmente bisogna scegliere il valore FILE e specificare il parametro NODE-PLACEMENT-FILE usando il nome del file contenente le coordinate iniziali di tutti i nodi. Guardiamo le varie opzioni:

1 RANDOM - i nodi sono posizionati in modo random sul terreno.
2 UNIFORM - in funzione del numero di nodi della simulazione il terreno è diviso in un certo numero di celle. Dentro ciascuna cella il nodo viene posizionato in modo random. Questo produce una topologia di rete casuale, ma con una densità piuttosto uniforme di nodi.
3 GRID - il posizionamento dei nodi parte dalla posizione (0,0). I nodi sono poi posizionati su una griglia, distanziati l'uno l'altro di una quantità pari a GRID-UNIT. GRID-UNIT deve essere specificato numericamente in me-

tri. Il numero di nodi specificati per la simulazione deve essere il quadrato di un intero (4, 9, 16, 25, ...)

4 FILE - La posizione dei nodi, nel caso di posizionamento manuale, viene letta dal file specificato dal parametro NODE-PLACEMENT-FILE. Tale file è un file di testo dove ciascuna linea ha il seguente formato:

<nodeId> <time> (<destination x coord>, <dest y>, <dest z>)

dove nodeId è l'identificatore di un nodo, time deve assumere il valore 0 (è un tempo) e tra le parentesi sono indicate le coordinate della posizione iniziale del nodo. Ogni nodo ha la propria riga.

Come detto in precedenza è possibile specificare quali protocolli si devono usare nell'esperimento. Tale selezione può essere fatta per ciascun nodo. Guardiamo un esempio:

```
MAC-PROTOCOL          802.11
TCP                   LITE
NETWORK-PROTOCOL      IP
IP-QUEUE-TYPE         FIFO
IP-QUEUE-SCHEDULER    STRICT-PRIORITY
ROUTING-PROTOCOL      BELLMANFORD
MULTICAST-PROTOCOL    DVMRP
```

Esaminiamo adesso come poter selezionare le varie statistiche che vogliamo ottenere dalla simulazione. Basta specificare YES nel relativo parametro. Guardiamo un esempio:

```
APPLICATION-STATISTICS          YES
TCP-STATISTICS                  YES
UDP-STATISTICS                  YES
RSVP-STATISTICS                 NO
```

```
ROUTING-STATISTICS                 YES
IGMP-STATISTICS                    NO
EXTERIOR-GATEWAY-PROTOCOL-STATISTICS  YES
NETWORK-LAYER-STATISTICS           YES
QUEUE-STATISTICS                   YES
MAC-LAYER-STATISTICS               YES
PHY-LAYER-STATISTICS               YES
MOBILITY-STATISTICS                NO
```

Esaminiamo adesso i parametri che riguardano la mobilità. Il parametro MOBILITY ha le seguenti opzioni:

1 NONE - I nodi rimangono in posizione fissa per tutto il tempo della simulazione.

2 RANDOM-WAYPOINT - i nodi scelgono in modo random la posizione di destinazione, e si muovono in direzione di questa ad una velocità uniforme scelta tra MOBILITY-WP-MIN-SPEED e MOBILITY-WP-MAX-SPEED. Dopo aver raggiunto la destinazione, i nodi vi rimangono per un tempo pari a MOBILITY-WP-PAUSE, dopo di che il procedimento viene ripetuto.

3 TRACE - il modello secondo cui si muovono i nodi è contenuto in un file specificato tramite il parametro MOBILITY-TRACE-FILE. Ciascuna linea di tale file ha il seguente formato: <nodeId> <time> (<destination x coord>, <dest y>, <dest z>). Tutte le righe che si riferiscono ad un nodo devono essere ordinate in base al tempo. Facciamo un esempio:

```
10 100S (200.0, 150.0, 0.2)
10 200S (200.0, 150.0, 0.2)
10 500S (250.0, 250.0, 0.1)
10 800S (200.0, 280.0, 0.5)
10 900S (300.0, 310.0, 1.0)
```

Facciamo un esempio di RANDOM-WAYPOINT.

```
MOBILITY                    RANDOM-WAYPOINT
MOBILITY-WP-PAUSE           30S
MOBILITY-WP-MIN-SPEED       1
MOBILITY-WP-MAX-SPEED       10
MOBILITY-POSITION-GRANULARITY  0.5
```

Il parametro MOBILITY-POSITION-GRANULARITY (definito in metri), determina la risoluzione alla quale la posizione dei nodi viene aggiornata. La velocità viene specificata in metri al secondo, e il tempo di pausa è dato in *QualNet Time Format*.
Facciamo un esempio di TRACE:

```
MOBILITY                    TRACE
MOBILITY-TRACE-FILE         default.mobility
MOBILITY-POSITION-GRANULARITY  0.5
```

Dove default.mobility è il file contenente il modello della mobilità dei vari nodi.
Per settare le applicazioni da usare nell'esperimento, bisogna specificare il nome del file che contiene i parametri di configurazione delle applicazioni.

```
APP-CONFIG-FILE default.app
```

Il file .app deve contenere varie righe, una per ogni applicazione, contenenti il nome dell'applicazione ed i nodeId della coppia di nodi tra cui l'applicazione deve lavorare. Oltre a questo, tali righe devono contenere i parametri specifici di ogni applicazione. Ogni applicazione che prende parte alla simulazione, ha i propri parametri specificati sulla riga.
QualNet fornisce delle API per leggere i parametri dal file di configurazione. Ad esempio, per leggere un particolare parametro dal file di configurazione si può usare la seguente API:

```
void IO_ReadString( const NodeAddress nodeId,
                    const NodeAddress interfaceAddress,
                    const NodeInput* nodeInput,
                    const char* index,
                    const BOOL* wasFound,
                    const char* readVal);
```

Il *nodeInput* solitamente rappresenta il file di configurazione. Il campo *index* rappresenta il nome del parametro e *readVal* rappresenta il valore letto nel file di configurazione per tale parametro. Il parametro *wasFound* informa chi ha chiamato la funzione della presenza nel file di configurazione del parametro specificato.

4.2 Interfacce grafiche

In questo capitolo sono esaminate le interfacce grafiche presenti in QualNet 3.6. Tali interfacce hanno il compito di semplificare l'utilizzo di QualNet Simulator. Per maggiori informazioni circa questi strumenti, riferirsi al manuale fornito con QualNet.

4.2.1 QualNet Animator

QualNet Animator è uno strumento scritto in Java che esegue QualNet simulator in un processo separato e comunica con esso tramite sockets e funzioni standard di I/O. Si presenta come una classica interfaccia per il disegno grafico, con una grande tela centrale dove è possibile settare un esperimento (posizionamento dei nodi, creazione dei link, etc.). Le sue funzioni principali sono le seguenti:

- Permette il settaggio di un esperimento. Tale settaggio viene scritto nel file di configurazione di QualNet Simulator.
- Permette di eseguire l'esperimento settato e di animarlo. Inoltre, consente il controllo in tempo reale di QualNet Simulator.

4.2.2 QualNet Designer

QualNet Designer è uno strumento che facilita la scrittura e l'inserimento in QualNet Simulator di un nuovo protocollo, e si presenta con l'aspetto di una tipica interfaccia grafica. I protocolli, in QualNet Designer, devono essere creati usando una rappresentazione a diagramma di stato (il diagramma che andrà a rappresentare il protocollo usa una notazione UML).

Per realizzare un protocollo usando QualNet Designer bisogna descrivere i vari stati in cui esso si può venire a trovare, gli eventi gestiti da tali stati, e le varie transizioni di stato. Questo può essere fatto mediante gli strumenti grafici forniti, che permettono di disegnare i vari stati e di tracciare le varie transizioni di stato che si possono avere. Ogni stato include un entry code, vale a dire del codice che viene eseguito ogni qual volta si accede a tale stato. Tale codice, deve essere scritto in linguaggio C, utilizzando l'editor messo a disposizione da QualNet Designer o un editor scelto dall'utente.

Quando si sceglie di realizzare un nuovo protocollo, il sistema crea automaticamente due stati:

- Initial State.
- Final State.

Questi due stati hanno delle caratteristiche speciali rispetto agli altri stati inseriti dall'utente. Esaminiamole:

i) Designer non permette che si creino transizioni che partono o che arrivano al *final state* e non permette che si creino neanche transizioni che arrivano all'*initial state*;

ii) Dall'*initial state* possono partire solo transizioni automatiche o che si basano su condizioni booleane;

iii) Nell'*initial state* il protocollo si viene a trovare solo all'inizio della simulazione (l'*entry code* relativo viene eseguito una sola volta durante la simulazione), dopo di che il protocollo deve passare in un altro stato. In molti casi, l'*initial state* ha una transizione automatica verso il *dispatcher state*, ma in casi come le applicazioni client-server, per uscire

dall'*initial state* vengono usate delle *guarded transitions*. Questo permette di passare, a seconda del valore di una variabile, negli stati relativi al client o negli stati relativi al server.

iv) Il protocollo passa nel *final state* automaticamente al termine della simulazione.

In QualNet si possono definire i seguenti tipi di transizione:

- Triggered: è una transizione che avviene su ricezione di un particolare evento.
- Guarded: è una transizione che avviene quando siamo già entrati in uno stato e risulta vera una particolare condizione booleana.
- Automatic: è una transizione che si verifica automaticamente su completamento dell'entry code dello stato.
- Triggered and Guarded: mette insieme i due aspetti.

Durante la fase di sviluppo è importante conoscere le API necessarie per implementare il protocollo (come quelle che gestiscono i messaggi). QualNet Designer facilita tale compito elencando, in un apposito menu, le varie API messe a disposizione per il livello a cui appartiene il protocollo che si sta implementando, e permettendo all'utente di selezionare quella desiderata con un click. Le varie API sono suddivise in due categorie:

- General API: una lista di tutte le funzioni e routines disponibili a tutti i livelli (Sending message, setting timers, etc.).
- Layer API: una lista di tutte le funzioni e routines disponibili solo per il livello a cui appartiene il protocollo che si sta implementando.

Oltre a fornire l'elenco delle API disponibili, QualNet Designer inserisce automaticamente del codice al protocollo che si sta creando. In particolare, inserisce il codice necessario per leggere i parametri di configurazione del nostro protocollo, e il codice per gestire le statistiche (crea la struttura dati per contenere le statistiche e la funzione che serve per stamparle).

Una volta completata la descrizione del protocollo tramite macchina a stati, ed una volta definiti gli entry code per i vari stati, può essere generato il codice (in lin-

guaggio C) che dovrà essere inserito in QualNet per rendere attivo il protocollo. Questo può essere fatto utilizzando un apposito comando. Questo comando organizza il codice su due file: un header file ed un body file. In seguito alla generazione del codice, il protocollo può essere aggiunto a QualNet usando l'apposito comando. Questo comando va a modificare i files sorgenti di QualNet nel modo analizzato nei precedenti capitoli. Per finire, c'è il comando che permette di compilare QualNet per rendere attivo il protocollo.
Per adesso, QualNet Designer supporta la creazione dei seguenti protocolli: client-server applications, application-routing, network-routing, e MAC layer protocols. Comunque è possibile modificarlo per permetergli di generare protocolli di qualsiasi livello (guardare il QualNet Developer's Guide fornito con QualNet).

4.2.3 QualNet Analizer

QualNet Analizer Analizer è uno strumento che permette di visualizzare le statistiche generate da un esperimento. Basta scegliere il file .stat da analizzare per visualizzare le varie statistiche, suddivise per livelli. Se vengono selezionati più file .stat che si riferiscono a più esecuzioni dello stesso esperimento, le statistiche vengono mostrate mettendo a paragone quelle ottenute per ciascuna esecuzione.

Capitolo V.

Realizzazione del TPA

Il protocollo TPA è stato implementato per essere utilizzato nell'ambiente di sviluppo e simulazione QualNet Developer. Per la realizzazione del TPA ed il suo inserimento in QualNet è stato seguito quanto detto nel capitolo 4.1. Di seguito viene riportata la descritta dell'implementazione del TPA (contenuta nei file TPA.c e TPA.h) e dell'applicazione FTP/TPA, realizzata per utilizzare il protocollo TPA.

5.1 Realizzazione del protocollo TPA

In questa sezione viene descritta l'implementazione del TPA in QualNet, seguendo il seguente schema: i) vengono elencati gli eventi scambiati dal TPA e le relative strutture dati; ii) vengono elencate le principali strutture dati del TPA; iii) vengono descritte le principali funzioni che modellano il comportamento del TPA. Per quanto riguarda l'inserimento del TPA in QualNet, bisogna rifarsi al capitolo 4.1.3.

Rispetto alla descrizione del TPA data nel capitolo 3, l'implementazione è priva di alcuni aspetti. Vediamo quali: la fase d'apertura è stata semplificata, consentendo solo al lato client di richiedere l'apertura della connessione; non è stato preso in considerazione il messaggio di ELFN proveniente dal livello network; il TPA deve ricevere dalla applicazione dei pacchetti dati di dimensione pari a K * MSS, dove K è il numero di pacchetti che possono essere contenuti nel send buffer e MSS è la dimensione massima dei segmenti che il TPA spedisce al livello network; il TPA non fa il piggybacking degli ack dentro i pacchetti. In altre parole, gli ack vengono spediti su pacchetti distinti rispetto ai pacchetti di dati; non viene utilizzato il campo checksum del pacchetto per rilevare eventuali errori di trasmissione al livello trasporto.

Esaminiamo gli eventi gestiti dal TPA e le strutture dati da usare come campo *info* nei messaggi relativi a tali eventi (file api.h).

MSG_TRANSPORT_FromNetwork**:** è l'evento schedulato dal livello network quando deve passare un pacchetto al livello transport. La struttura dati passata come campo *info* del messaggio associato a tal evento è la NetworkToTransportInfo. I campi di questa struttura gestiti dal TPA sono i seguenti:

- *sourceAddr* - indirizzo IP della sorgente del pacchetto.
- *destinationAddr* - indirizzo IP della destinazione del pacchetto.
- *Priority* - priorità del pacchetto.

Queste informazioni, devono essere passate al livello transport attraverso il campo info, poiché sono necessarie al funzionamento del TPA ma non fanno parte del pacchetto.

MSG_TRANSPORT_FromAppListen**:** è l'evento schedulato dal lato server dell'applicazione quando vuole eseguire una *listen* per mettersi in attesa di ricevere le varie richieste di connessione su di una specifica porta. La struttura dati passata come campo *info* del messaggio associato a tale evento è la AppToTpaListen e contiene tutte le informazioni necessarie al TPA per implementare la *listen*. Tale struttura ha i seguenti campi:

- *appType* - specifica il tipo di applicazione che ha effettuato la listen.
- *localAddr* - specifica l'indirizzo IP locale della applicazione che ha effettuato la listen
- *localPort* - specifica il numero di porta su cui è in ascolto l'applicazione.
- *priority* - specifica la priorità della connessione che si vuole aprire.

MSG_TRANSPORT_FromAppOpen**:** è l'evento schedulato dal client quando vuole effettuare una *open* per aprire una connessione. La struttura dati passata come campo *info* del messaggio associato a tale evento è la AppToTpaOpen e contiene le informazioni necessarie al TPA per intraprendere la fase di apertura della connessione. Tale struttura ha i seguenti campi:

- *appType* - specifica il tipo di applicazione che ha richiesto l'apertura.

- *localAddr* - specifica l'indirizzo IP locale della applicazione che ha richiesto l'apertura.
- *localPort* - specifica la porta su cui è stata fatta la open.
- *remoteAddr* - specifica l'indirizzo IP dell'host verso cui si vuole effettuare la connessione.
- *remotePort* - specifica la porta su cui è in ascolto il server.
- *uniqueId* - campo che specifica una particolare coppia <client,server>.
- *Priority* - priorità della connessione.

MSG_TRANSPORT_FromAppSend: è l'evento schedulato dall'applicazione per spedire dati lungo la connessione. La struttura dati passata nel campo *info* del messaggio associato a tale evento è la AppToTpaSend e contiene i seguenti campi:

- *connectionId* - identifica la connessione a cui appartiene il pacchetto spedito.

MSG_TRANSPORT_FromAppClose: è l'evento schedulato dall'applicazione per effettuare la chiusura del suo capo della connessione (non ha più dati da spedire). La struttura dati passata nel campo *info* del messaggio associato a tale vento, è la AppToTpaClose e contiene i seguenti campi:

- *connectionId* - identifica la connessione che deve essere chiusa.

MSG_TRANSPORT_TPA_Timer: questo evento, a differenza dei precedenti, è stato aggiunto ai vari eventi disponibili in QualNet. Per fare questo è stato inserito l'enumerato MSG_TRANSPORT_TPA_Timer = 511 alla lista dei vari eventi presente nel file api.h. La struttura dati passata nel campo *info* del messaggio associato a tale evento è la TpaTimerPacket e contiene i seguenti campi:

- *timerType* - indica il tipo di timer a cui si riferisce il messaggio. Nel TPA sono definiti i seguenti tipi di timer: TM_SYNPERSO, TM_SYNACKPERSO, TM_FINPERSO, TM_CLOSE, TM_FINACKPERSO, TM_DATOPERSO. Questi si riferiscono ai vari pacchetti che possono andare persi durante la vita della connessione, tranne TM_CLOSE, che serve per schedulare la chiusura della connessione.

- *timerId* - ha significato solo quando il timer è di tipo TM_DATOPERSO e serve per indicare a quale pacchetto all'interno del buffer si riferisce il timer.
- *connectionId* - identifica la connessione a cui si riferisce il timer.

Esaminiamo adesso i messaggi che il TPA manda verso il livello applicazione.

MSG_APP_FromTransListenResult: è l'evento schedulato dal TPA per informare l'applicazione del risultato della *listen*. La struttura dati come campo *info* del messaggio associato a tale evento è la TransportToAppListenResult e contiene i seguenti campi:

- *localAddr* - rappresenta l'indirizzo IP locale.
- *localPort* - identifica la porta su cui è stata fatta la listen.
- *connectionId* - tale campo ha il seguente significato a seconda del suo valore: i) un valore di -2 indica che è già stata fatta una listen su quella porta; ii) un valore di -1 indica che non si è in grado di allocare le strutture dati che servono per gestire la connessione; iii) un valore maggiore o uguale di 0 indica che la listen è stata fatta con successo.

MSG_APP_FromTransOpenResult: è l'evento schedulato dal TPA per informare l'applicazione sul risultato dell'apertura della connessione. La struttura dati come campo *info* del messaggio associato a tale evento è la TransportToAppOpenResult e contiene i seguenti campi:

- *type* - indica se si tratta di una *active open* (0) o di una *passive open* (1). Il lato client riceverà tale messaggio con il campo type settato a 0, mentre il lato server riceverà tale messaggio con il campo type settato a 1.
- *localAddr* e *localPort* - indicano l'indirizzo IP e la porta del lato locale della connessione.
- *remoteAddr* e *remotePort* - identificano l'indirizzo IP e la porta del lato remoto della connessione.
- *connectionId* - ha il seguente significato, a seconda del suo valore: i) un valore di -1 indica che non è stato possibile aprire la connessione; ii) un

valore maggiore o uguale a zero indica che la connessione è aperta e la identifica in maniera univoca.

- *uniqueId* - identifica la coppia <client,server>.

MSG_APP_FromTransCloseResult: è l'evento schedulato dal TPA per informare l'applicazione della chiusura della connessione. La struttura dati usata come campo *info* del messaggio associato a tale evento è la TransportToAppCloseResult ed ha i seguenti campi:

- *type* - indica se tale messaggio si riferisce ad una *active close* (0) o ad una *passive close* (1). Più precisamente, il lato che decide di chiudere la connessione invia il FIN verso l'altro capo della connessione. Il TPA di tale capo della connessione, su ricezione del FIN, informa l'applicazione tramite l'evento MSG_APP_FromTransCloseResult. Se l'applicazione che riceve tale evento non ha ancora eseguito la *close*, il campo *type* del messaggio relativo a tale evento risulta essere settato ad 1. Invece, il messaggio di chiusura con il campo *type* settato a 0 viene ricevuto da entrambi i capi della connessione non appena la connessione risulta effettivamente chiusa (entrambi i capi della connessione risultano chiusi).
- *connectionId* - è l'identificatore della connessione che si sta chiudendo.

MSG_APP_FromTransDataReceived: è l'evento schedulato dal TPA per spedire i pacchetti ricevuti dal livello IP al livello applicazione. La struttura passata nel campo *info* del messaggio associato a tale evento è la TransportToAppDataReceived e contiene i seguenti campi:

- *connectionId* - identifica la connessione a cui appartiene il pacchetto ricevuto.

MSG_APP_FromTransDataSent: è l'evento che informa l'applicazione sul risultato della spedizione dei pacchetti. Più precisamente, un blocco di dati provenienti dall'applicazione viene spostato nel send buffer per essere spedito, viene informata di questo la applicazione, che così può passare al TPA un altro blocco di dati. La struttura passata nel campo *info* del messaggio associato a tale evento è la TransportToAppDataSent e contiene i seguenti campi:

- *connectionId* - identifica la connessione a cui appartengono i pacchetti spediti.
- *length* - identifica la lunghezza in byte dei dati spediti correttamente a destinazione.

Esaminiamo adesso le strutture dati del TPA (contenute in QUALNET_HOME/transport/tpa.h).

La struttura dati inpcbtpa rappresenta un'istanza di una connessione TPA. Esaminiamone i campi principali:

- *inp_next*, *inp_prev*, e *inp_head* - le struttura dati relative alle varie connessioni sono organizzate sottoforma di una lista. Questi campi servono per gestire tale lista.
- *app_proto_type* - rappresenta il tipo di applicazione a cui appartiene la connessione.
- <*inp_remote_addr*, *inp_remote_port*>, <*inp_local_addr*, *inp_local_port*> - tali coppie di campi rappresentano il socket del processo di destinazione e del processo sorgente.
- *inp_ppcb* - è il puntatore alla struttura dati che rappresenta l'istanza del TPA associata alla connessione.
- *con_id* - identifica in maniera univoca le varie connessioni appartenenti alla lista, facilitando le operazioni di ricerca.
- *sendBuffer* e *receiveBuffer* - sono i puntatori alle strutture dati che rappresentano il buffer di ricezione ed il buffer spedizione della connessione.
- *appendBuffer* – puntatore alla struttura dati che rappresenta il buffer dove vengono inseriti i pacchetti provenienti dalla applicazione.
- *userreq* - serve per identificare il tipo di richiesta dell'utente. Può assumere i seguenti valori: INPCBTPA_USRREQ_NONE indica che nessuna richiesta è stata fatta dall'utente; INPCBTPA_USRREQ_OPEN indica che l'utente ha richiesto l'apertura della connessione; INPCBTPA_USRREQ_CONNECTED indica che la connessione dell'utente

è attiva; INPCBTPA_USRREQ_CLOSE indica che l'utente ha effettuato una richiesta di chiusura della connessione.

- *unique_id* - è un campo usato per determinare una coppia client/server.
- *priority* - è la priorità dei pacchetti che circolano nella connessione.

La struttura dati circularBuffer serve per implementare il buffer di spedizione del TPA. Esaminiamone i campi che hanno bisogno di qualche spiegazione:

- *data* - è un vettore di puntatori a carattere di dimensione DIM_BUFFER che serve per memorizzare i pacchetti ricevuti dalla applicazione. Su ricezione di un pacchetto, viene allocata la memoria per contenerlo e viene fatta puntare dal primo campo libero di *data*.
- *pckAcked* - è un vettore di booleani di dimensione DIM_BUFFER. Se *pckAcked*[i] è TRUE, significa che il pacchetto *data*[i] è arrivato correttamente a destinazione.
- *indicePcks* - è un vettore di interi di dimensione 256 che serve per indicare a quale pacchetto si riferisce un certo ACK ricevuto. Se viene trasmesso l'i-esimo pacchetto (*data*[i]) con un *txSeqNumber* pari a x, viene inserito nell'x-esima entrata di *indicePcks* (*indicePcks*[x]) il valore "i". Quando si riceve un ACK, si prende il suo *ackSeqNumber* e si accede con tale valore al campo *indicePcks*, ottenendo così l'indice del pacchetto dentro il buffer a cui si riferisce l'ACK. Se *ackSeqNumber* vale x, *data*[*indicePck*[x]] è il pacchetto a cui si riferisce l'ACK. Tutto questo serve per poter calcolare il RTT dei pacchetti.
- *bit* - serve per analizzare il RTT dei pacchetti di una finestra che arrivano in ritardo. In pratica, se vale 0 significa che il buffer è stato tutto acknoleggiato ma non è ancora cominciata la spedizione di una nuova finestra. Si sfrutta questa conoscenza per continuare a processare eventuali pacchetti di ACK che possono continuare ad arrivare per la vecchia finestra (per una stima più precisa del RTT).
- *datiTimers* - è una struttura dati che serve per gestire i timers.

La struttura dati gestioneTimers serve per la gestione dei timers per i pacchetti appartenenti al blocco che si sta spedendo. Esaminiamo i suoi campi:

- *istanteTx* - è un vettore di dimensione DIM_BUFFER contenente l'istante di trasmissione dei vari pacchetti.
- *estremoInfDev* - è un vettore di dimensione DIM_BUFFER contenente l'estremo inferiore della varianza del RTT dei pacchetti.
- *estremoSupDev* - è un vettore di dimensione DIM_BUFFER contenente l'estremo superiore della varianza del RTT dei pacchetti.
- *timers* - è un vettore di dimensione DIM_BUFFER contenente i puntatori ai messaggi usati per i timers dei pacchetti.
- *pckTimerSet* - è un vettore di dimensione DIM_BUFFER che indica se per un certo pacchetto il timer è in funzione.
- *numPckTimeoutUnacked* - rappresenta il numero di pacchetti in timeout.

La struttura dati ReceiveBuffer rappresenta il buffer di ricezione dei pacchetti. Esaminiamo i campi che hanno bisogno di qualche spiegazione:

- *data* - è un vettore di caratteri contenente i puntatori ai pacchetti ricevuti.
- *bitmap* - è un campo di bit di dimensione DIM_BUFFER che indica quali pacchetti sono stati ricevuti correttamente. Se l'i-esimo bit vale 1 significa che l'i-esimo pacchetto del buffer è stato ricevuto.

La struttura dati AppendBuffer rappresenta il buffer dove sono inseriti i pacchetti provenienti dall'applicazione. Possiede i seguenti campi:

- data - buffer di dimensione MSS * DIM_BUFFER bytes in grado di contenere un blocco di dati provenienti dalla applicazione.
- Full - indica se il buffer data è pieno.

La struttura dati TransportDataTpa è la struttura dati del protocollo TPA. Questa viene istanziata dalla funzione d'inizializzazione del protocollo non appena la simulazione parte. Esaminiamo i suoi campi:

- *head* - rappresenta la testa della coda relativa alle varie connessioni.
- *tpaIss* e *tpaIWss* - rappresentano il valore iniziale da dare al *txSeqNumber* e alla *winSeqNumber* per spedire il pacchetto di SYN.

- *tpaStatsEnabled* - è un flag che indica se per il TPA sono state abilitate o meno le statistiche.
- *tpaStat* - è il puntatore alla struttura dati che contiene le varie statistiche che devono essere raccolte.

La struttura tpaStatGlobal contiene la dichiarazione di tutte le statistiche definite per il nostro protocollo. Esaminiamo le più importanti:

- *tpas_connects* - indica il numero di connessioni stabilite.
- *tpas_closed* - indica il numero di connessioni chiuse.
- *tpas_sndtotal* - indica il numero totale di pacchetti spediti.
- *tpas_sndpack* - indica il numero di pacchetti dati spediti.
- *tpas_sndctrl* - indica il numero di pacchetti di controllo spediti.
- *tpas_rcvtotal* - indica il numero di pacchetti ricevuti.
- *tpas_rcvctrl* - indica il numero di pacchetti di controllo ricevuti.
- *tpas_rcvpack* - indica il numero di pacchetti dati ricevuti.
- *tpas_rtxPack* - indica il numero di pacchetti ritrasmessi.
- *PckToApp* - indica il numero di pacchetti passati alla applicazione.
- *tpas_pckNonConsiderati* - indica il numero di pacchetti errati che sono stati ricevuti.
- *NumWinTx* - indica il numero di blocchi trasmessi.
- *numPassaggiTxW1vsTxW3* - indica quante volte si è passati da una spedizione con finestra di 1 ad una spedizione con una finestra di 3.
- *numPassaggiTxW3vsTxW1* - indica quante volte si è passati da una spedizione con finestra di 3 ad una spedizione con una finestra di 1.
- *rtoClassicVSrtoMod* - indica quante volte siamo passati dal calcolo classico del RTO al calcolo modificato.
- *rtoModVSrtoClassic* - indica quante volte siamo passati dal calcolo modificato al calcolo classico del RTO.
- *winReceived* - indica il numero di finestre ricevute.
- *numPckAnt* - indica il numero di pacchetti che sono arrivati in anticipo.

- *numPckRit* - indica il numero di pacchetti che sono arrivati in ritardo.

La struttura dati tpacb è la struttura dati del TPA associata ad una connessione. Esaminiamone i campi più importanti:

- *t_state* - descrive lo stato della connessione. Una connessione TPA, in accordo con quanto detto in 3.3.3, si può trovare in uno dei seguenti stati: TPA_CLOSED, TPA_LISTEN, TPA_SYN_SENT, TPA_SYN_RCVD, TPA_ESTABLISHED, TPA_FIN_WAIT_1, TPA_FIN_WAIT_2, TPA_CLOSING, TPA_TIME_WAIT, TPA_CLOSE_WAIT, e TPA_LAST_ACK. Il campo *t_state* indica in quale di questi stati si trova la connessione.
- *tx_state* - descrive lo stato di trasmissione della connessione. Può assumere i valori TxW3_STATE e TxW1_STATE a seconda che si stia trasmettendo con una finestra di 1 o di 3 pacchetti.
- *timer_state* - indica se siamo in una condizione di *route-changes* (sotto o sovra stima del RTO).
- *t_flags* - specifica che tipo di operazione deve effettuare la funzione *tpa_output( )*. In particolare: TF_ACK indica che deve essere spedito un pacchetto di ACK; TF_SYN indica che deve essere spedito un pacchetto di SYN; TF_SYNACK indica che deve essere spedito un pacchetto di SYNACK; TF_ACKDELSYN indica che deve essere acknoleggiato un SYN; TF_DATA indica che deve essere spedito un pacchetto dati; TF_ACKDATA indica che deve essere spedito un ACK per i dati; TF_SENDFIN indica che deve essere spedito un pacchetto di FIN; TF_ACKFIN indica che deve essere acknoleggiato il FIN e che va spedito un FIN; TF_ACKDELFIN significa che dobbiamo acknoleggiare un FIN.
- *t_inpcb* - è il puntatore alla struttura dati della connessione.
- *synTx* - è il *txSequenceNumber* del SYN che si deve spedire per aprire la connessione. Su rispedizione del SYN tale campo non viene incrementato.
- *istanteTxSyn* - specifica, in clocktype, l'istante di spedizione del SYN. Serve per il calcolo del RTT.

- *txSeqNumSynRc* e *winSeqNumSynRc* - sono rispettivamente il *txSeqNumber* e il *winSeqNumber* del SYN ricevuto dall'altro capo della connessione.
- *txSeqNumFinRc* e *winSeqNumFinRc* - sono rispettivamente il *txSeqNumber* e il *winSeqNumber* del pacchetto di FIN ricevuto.
- *numFinTx* - conta il numero di FIN che non hanno ricevuto l'ack. Se il TPA si trova in TPA_CLOSING state e scatta per tre volte il timer dell'ACK, la connessione viene considerata chiusa. Una condizione di questo genere si ha quando un capo della connessione è già chiusa mentre l'altro capo non ha ancora ricevuto l'ack del FIN.
- *txSeqNumber* - è il *txSeqNumber* con cui verrà spedito il prossimo pacchetto. Ad ogni spedizione di un pacchetto di un blocco tale numero viene incrementato. Non appena si passa a spedire il blocco successivo, tale campo viene azzerato.
- *winSeqNumber* - indica il blocco attualmente in uso per spedire i pacchetti e rappresenta la *winSeqNumber* con cui verranno spediti i pacchetti. Non appena si passa alla spedizione del blocco successivo, tale campo viene azzerato.
- *ackWinSeqNumber* - indica la finestra dove devono arrivare i dati. Tale campo, non appena viene stabilita la connessione viene settato al valore del *winSeqNumber* del SYN ricevuto incrementato di uno. Tutte le volte che un blocco risulta completo, e i dati vengono passati alla applicazione, tale campo viene incrementato di uno, indicando che il Sender dovrebbe passare alla spedizione del nuovo blocco. Se così non è, perché il Receiver ha collezionato tutti i pacchetti, ma il Sender non ha ricevuto tutti gli ACK, su ricezione di un pacchetto della vecchia finestra si spedisce un ACK con una bitmap con tutti i bit settati ad uno.
- *ERTT* e *DEV* - rappresentano rispettivamente il valore medio e la deviazione standard della stima del *RTT*, mentre *RTO* rappresenta l'ultimo retransmission timeout calcolato.

- x, y, e z - sono i parametri in uso per il calcolo del *RTO*.
- *<numPckAnticipati,numPckAntCons>,<numPckRitardati,numPckRitCons>, <numPckGiusti, numPckGiustiCons>* - tali coppie servono per gestire le *routechanges*. Esaminiamo il caso dei pacchetti anticipati. Dopo la ricezione di PCKANTICIPATI pacchetti anticipati consecutivi, con al massimo un pacchetto non anticipato, noi vogliamo dare per avvenuta una *routechanges*. Su ricezione di un pacchetto anticipato si incrementano entrambi i campi *numPckAnticipati* e *numPckAntCons*. Se arriva un pacchetto non anticipato, si decrementa il campo *numPckAntCons*. Se il numero di pacchetti giusti o ritardati, ricevuti durante il calcolo dei pacchetti anticipati, arriva a due, il calcolo di questi viene azzerato. Non appena *numPckAnticipati* arriva a PCKANTICIPATI si controlla *numPckAntCons*. Se ha un valore maggiore uguale di PCKANTICIPATI – 1, significa che sono arrivati PCKANTICIPATI pacchetti anticipati con al massimo un pacchetto non anticipato.
- *ackRcvdInTxW1State* - conta il numero di ack ricevuti nel TxW1-State. Serve per il passaggio in TxW3 state.
- *timerSyn*, *timerSynAck*, *timerFin*, e *timerFinAck* – sono i timer usati per la spedizione dei pacchetti di SYN e di FIN. *timerUscita* è il timer usato per passare dal TPA_TIME_WAIT state al CLOSED state, per evita il fenomeno dei pacchetti duplicati dopo la chiusura della connessione (cap. 3.3.10).

Esaminiamo adesso le funzioni principali del TPA (file QUALNET_HOME/transport/tpa.c).

La TransportTpaInit( ) è la funzione d'inizializzazione del protocollo ed è chiamata all'inizio della simulazione. Ha i seguenti parametri:

- *node* - puntatore alla struttura dati *Node* del nodo che ha chiamato la funzione di inizializzazione del protocollo.

- *nodeInput* - struttura dati che contiene il contenuto del file di configurazione della simulazione.

I suoi compiti principali sono i seguenti:

- Alloca la memoria necessaria per contenere la struttura dati transportDataTpa e assegna alla variabile node->transportData.tpaVariable il valore del puntatore alla memoria allocata. Questo serve per accedere alla struttura dati transportDataTpa durante la simulazione.
- Viene inizializzata la coda che dovrà contenere le strutture dati relative alle varie connessioni aperte dal TPA.
- Sono assegnati i valori ai campi *tpaIss* e *tpaIWss* della struttura transportDataTpa.
- Viene esaminato il parametro che abilita le statistiche per il TPA, leggendolo dal parametro nodeInput della funzione TransportTpaInit(), e viene aggiornata in modo opportuno la struttura dati transportDataTpa. Più precisamente viene allocata la memoria per contenere la struttura dati tpaStatGlobal e vengono gestiti i campi TpaStatsEnabled e TpaStat della struttura dati transportDataTpa.

La funzione TransportTpaLayer( ) è l'event dispatcher del TPA. Ha i seguenti parametri di ingresso:

- *node* - puntatore alla struttura dati *Node* del nodo che riceve il messaggio.
- *msg* - puntatore alla struttura dati del messaggio ricevuto.

Tale funzione recupera il puntatore alla struttura dati del TPA (node->transportData.tpaVariable) e implementa uno statement switch sul tipo d'evento ricevuto. Tale statement include tutti gli eventi gestiti dal TPA e in funzione dell'evento ricevuto chiama la funzione che lo deve gestire. Elenchiamo gli eventi gestiti dal TPA e le funzioni chiamate per gestirli:

- MSG_TRANSPORT_FromNetwork - tale evento è gestito dalla funzione tpa_input().
- MSG_TRANSPORT_FromAppListen - tale evento è gestito dalla funzione tpa_listen().

- MSG_TRANSPORT_FromAppOpen - tale evento è gestito dalla funzione tpa_connect().
- MSG_TRANSPORT_FromAppSend - tale evento è gestito dalla funzione tpa_FillBuffer().
- MSG_TRANSPORT_FromAppClose - tale evento è gestito dalla funzione tpa_disconnect().
- MSG_TRANSPORT_TPA_Timer - per gestire tale evento, prima viene ricavata la connessione a cui è diretto (tramite il campo *connectionId* del campo *info* del messaggio) e poi viene analizzato il campo timerType del campo *info* del messaggio. In funzione del valore assunto da questo, vengono intraprese le seguenti azioni: i) il timer di tipo TM_SYNPERSO scatta quando il pacchetto di SYN spedito viene dato per perso. Appena viene ricevuto l'evento relativo a tale timer, viene chiamata la funzione tpa_output( ) per rispedire il SYN; ii) il timer di tipo TM_SYNACKPERSO scatta quando viene perso il pacchetto di SYNACK. Appena viene ricevuto l'evento relativo a tale timer, viene chiamata la funzione tpa_output( ) per rispedire il SYNACK; iii) il timer TM_FINPERSO scatta quando il pacchetto di FIN si perde. Appena viene ricevuto l'evento relativo a tale timer, viene chiamata la tpa_output( ) per rispedire il FIN. Inoltre, se tale timer è scattato gia tre volte e ci troviamo in TPA_CLOSING state, viene schedulato il timer per la chiusura della connessione; iv) il timer TM_CLOSE è il timer che serve per la chiusura della connessione e scatta quando si deve passare dal TIME_WAIT state al CLOSED state. Appena viene ricevuto l'evento relativo a tale timer, viene chiamata la funzione tpa_close( ); v) il timer TM_FINACKPERSO scatta quando il pacchetto di FINACK viene dato per perso. Appena viene ricevuto l'evento relativo a tale timer, viene chiamata la funzione tpa_output( ) per rispedire il FINACK; vi) il timer TM_DATOPERSO scatta quando un pacchetto dati viene dato per perso. Appena viene ricevuto l'evento relativo a tale timer, tramite il campo timerId del campo info del messaggio viene ricavato

l'indice del pacchetto in timeout, viene aggiustato il sendbuffer e viene chiamata la chiamata la fuzione tpa_send( ) per continuare con la spedizione dei pacchetti. Inoltre si controlla il numero di pacchetti in timeout per vedere se si deve passare in TxW1_STATE.

La tpa_connect( ) è la funzione che gestisce la richiesta d'apertura di connessione proveniente dall'applicazione. Ha i seguenti parametri:

- *node* - puntatore alla struttura dati Node del nodo che sta aprendo la connessione.
- *phead* - puntatore alla struttura dati inpcbtpa che punta alla testa della lista delle connessioni.
- *app_type* - indica il tipo di applicazione che ha richiesto l'apertura della connessione.
- *local_addr*, *local_port* - socket del lato locale della connessione.
- *remote_addr*, *remote_port* - socket della parte remota della connessione.
- *tpa_iss* - *txSeqNumber* del pacchetto di SYN che va spedito.
- *tpa_wiss* - winSeqNnumber del pacchetto SYN che va spedito.
- *tpa_stat* - puntatore alla struttura dati che contiene le statistiche.
- *unique_id* - identificatore che determina la coppia client/server.
- *priority* - indica la priorità della connessione.

Tale funzione svolge i seguenti compiti:

- Si accerta che non esista già una connessione tra la stessa coppia di socket. Per fare questo utilizza la funzione in_pcbtpalookup( ) con l'ultimo parametro settato ad INPCBTPA_NO_WILDCAR.
- Se la connessione non esiste, crea una nuova istanza della struttura dati inpcbtpa e della struttura dati tpacb usando la funzione tpa_attach( ) e inizializza in modo opportuno i loro campi. In particolare, entra in TPA_SYN_SENT state in modalità di trasmissione RTOCLASSIC, setta i parametri per il calcolo del RTO e chiama la funzione tpa_output( ) per spedire il SYN.

- Se la connessione era già presente nella lista, oppure non è stato possibile allocare la memoria necessaria per contenere le strutture dati della nuova connessione, viene informata l'applicazione del fallimento dell'apertura.
- Viene settato il campo usrreq della struttura dati inpcbtpa al valore IN-PCBTPA_USRREQ_OPEN, per indicare che l'utente ha effettuato una richiesta di apertura di connessione.

La funzione tpa_listen( ) gestisce le richieste di listen provenienti dal server. Accetta i seguenti parametri in ingresso:

- *node* - puntatore alla struttura dati Node del nodo che sta aprendo la connessione.
- *phead* - puntatore alla struttura dati inpcbtpa che punta alla testa della lista delle connessioni.
- *app_type* - tipo di applicazione che ha effettuato la listen.
- *local_addr*, *local_port* - socket su cui è stata fatta la listen.
- *priority* - indica la priorità della connessione.

Svolge i seguenti compiti:

- Guarda se è gia stata fatta una listen sullo stesso socket, usando la funzione in_pcbtpalookup( ) con l'ultimo parametro settato ad IN-PCBTPA_WILDCARD.
- Se non era stata fatta nessuna listen sul socket d'ingresso, viene chiamata la funzione tpa_attach( ) per creare una nuova istanza delle strutture dati inpcbtpa e tpacb. Tale funzione viene chiamata con i parametri *remote_addr*, *remote_port* e *unique_id* settati a -1. Si passa in TPA_LISTEN state.
- Viene riportato lo stato della listen alla applicazione, con i seguenti valori per il campo connectionId del campo info del messaggio: i) -2 significa che la porta era già in uso; ii) -1 significa che ci sono stati problemi con l'allocazione della memoria; iii) un numero >= 0 indica che la listen è stata eseguita con successo.

La funzione tpa_attach( ) crea una nuova istanza per le strutture dati inpcbtpa e tpacb. Accetta i seguenti parametri:

- *node* - puntatore alla struttura dati *Node* del nodo che sta aprendo la connessione.
- *head* - puntatore alla testa della lista delle connessioni.
- *app_type* - tipo di applicazione che ha richiesto l'apertura della connessione.
- *local_addr*, *local_port* - socket del lato locale della connessione.
- *remote_addr*, *remote_port* - socket della parte remota della connessione.
- *unique_id* - identificatore che determina la coppia client/server della connessione.
- *priority* - indica la priorità della connessione.

Svolge le seguenti azioni:

- recupera la struttura dati del TPA utilizzando la variabile node->transportData.tpaVariable.
- crea una istanza della struttura dati inpcbtpa (con la funzione in_pcbtpaalloc( )) e riempie i suoi campi con le informazioni passate come parametri. Se è stata fatta una listen (e quindi i parametri *remote_addr* e *remote_port* valgono -1), viene chiamata la in_pcbtpaalloc( ) con il secondo parametro a zero. Altrimenti il secondo parametro viene settato ad 1. Il campo con_id di inpcbtpa viene assegnato tramite la funzione get_conidtpa( ). Tale funzione incrementa una variabile ogni qual volta viene aperta una connessione e torna il suo valore.
- Viene creata una istanza della struttura dati tpacb per la nuova connessione usando la funzione tpa_newtpacb( ). Tale funzione alloca la memoria per contenere la stuttura tpacb ed inizializza i campi *t_inpcb* di tpacb e *inp_ppcb* di inpcbtpa.
- Viene settato lo stato della connessione a TPA_CLOSED.

La funzione in_pcbtpalookup( ) fa una ricerca all'interno della lista delle connessioni. Ha i seguenti parametri d'ingresso:

- *head* - puntatore alla testa della lista delle connessioni.
- *local_addr, local_port* - socket del lato locale della connessione.
- *remote_addr, remote_port* - socket della parte remota della connessione.
- *flag* – serve per decidere la modalità di ricerca utilizzata dalla funzione.

Se il parametro *flag* vale INPCBTPA_NO_WILDCARD, la funzione fa una ricerca per vedere se la connessione specificata dalla coppia di socket passata come parametri è presente. Altrimenti, se il *flag* vale INPCBTPA_WILDCARD, la funzione fa una ricerca per vedere se è già stata fatta una listen sul socket <*local_addr*, *local_port*>. In entrambi i casi, la funzione torna, se presente, il puntatore alla struttura dati corrispondente ai criteri di ricerca, altrimenti torna il puntatore alla testa della lista.

La funzione in_pcbtpaalloc( ) crea un'istanza della struttura dati inpcbtpa e la inserisce nella coda delle connessioni. Ha i seguenti parametri d'ingresso:

- *head* - puntatore alla testa della lista dove inserire la struttura dati inpcbtpa.
- *i* - intero che indica se bisogna allocare una struttura dati per una connessione o per una listen.

Svolge i seguenti compiti:

- Alloca la memoria per la struttura inpcbtpa.
- Se *i* vale 1 significa che la struttura dati da allocare si riferisce ad una connessione. Quindi vengono allocate le strutture dati circularBuffer e ReceiveBuffer e vengono inizializzate.
- Inserisce l'istanza creata nella lista delle connessioni, usando la funzione insque( ).

La funzione tpa_output( ) serve per spedire i pacchetti verso il livello IP. Ha i seguenti parametri di ingresso:

- *node* - puntatore alla struttura *Node* del nodo che sta effettuando la spedizione.
- *tp* - puntatore alla struttura dati tpacb relativa alla connessione che sta spedendo il pacchetto.

- *tpaStat* - puntatore alla struttura dati tpaStatGlobal relativa alla connessione che sta spedendo il pacchetto.
- *hdr* - puntatore alla struttura dati tpaHdr che rappresenta l'header del pacchetto TPA.

Tale funzione recupera le strutture dati transportDataTpa e inpcbtpa della connessione a cui appartengono i pacchetti da spedire e implementa uno statement switch sul campo *t_flags* della struttura dati tpacb. Tale campo specifica che tipo di pacchetto si sta spedendo (SYN, FIN, DATA, …). Guardiamo i vari casi che si possono avere:

- TF_SYN - viene creato il messaggio contenente il pacchetto di SYN e viene spedito al livello IP. Inoltre viene settato il timer relativo al SYN ad un valore pari a 1 secondo (usando la funzione settaTimer( )). Per le rispedizioni del SYN si usa sempre lo stesso *txSeqNumber* (campo *synTx* di tpacb). I campi per riempire l'header del pacchetto vengono prelevati dalla struttura dati tpacb puntata da *tp*.
- TF_SYNACK - viene creato e spedito il pacchetto di SYNACK di risposta al SYN ricevuto. Inoltre viene settato il timer per la rispedizione del SYNACK ad un valore pari a 1 secondo (usando la funzione settaTimer( )). Per le rispedizioni del SYN si usa sempre lo stesso *txSeqNumber* (campo *synTx* di tpacb). I campi per riempire l'header del pacchetto vengono prelevati dalla struttura tpacb puntata da *tp*.
- TF_ACKDELSYN - viene creato e spedito il pacchetto di ACK di risposta al pacchetto di SYNACK ricevuto. Su tale pacchetto non si effettua la spedizione di dati. I campi per riempire l'header del pacchetto vengono prelevati dalla struttura tpacb puntata da *tp*.
- TF_DATA - viene creato e spedito un messaggio contenente un pacchetto dati prelevato dal buffer di spedizione. I campi per riempire l'header del pacchetto vengono prelevati dalla struttura tpacb puntata da *tp*. Il campo *bitmapData* dell'header viene ricavato con l'operazione seguente:

2^(indice pacchetto spedito)

Inoltre vengono aggiornati i campi della struttura dati circularBuffer e viene settato il timer relativo al pacchetto che si sta spedendo (funzione settaTimer( )). Per settare il timer si usa il valore del RTO contenuto nella struttura dati tpacb puntata da *tp*. L'estremo inferiore (superiore) dell'intervallo entro cui deve arrivare il pacchetto (in assenza di *route-changes*), è calcolato sottraendo (sommando) al valore stimato per il RTT (ERTT), il valore della deviazione standard moltiplicata per DIMINTERVALLO.

- TF_ACKDATA - viene creato e spedito un pacchetto di ACK per acknoleggiare un pacchetto dati ricevuto. Il parametro *hdr* punta all'header del pacchetto dati che si sta acknoleggiando.
- TF_SENDFIN - Viene creato e spedito un pacchetto di FIN. Inoltre viene settato il timer per una eventuale ritrasmissione. Spedendo i pacchetti di FIN non si incrementato il numero di sequenza.
- TF_ACKFIN - Viene creato e spedito un pacchetto di FINACK. Tale pacchetto acknoleggia il pacchetto di FIN ricevuto e spedisce il pacchetto di FIN per chiudere il capo della connessione che è ancora aperto. Viene settato il timer per una eventuale ritrasmissione del FIN.
- TF_ACKDELFIN - Viene creato e spedito il pacchetto di ACK per ackoleggiare il pacchetto di FIN ricevuto. Il parametro *hdr* punta all'header del pacchetto dati che si sta acknoleggiando.

La funzione settaTimer( ) ha i seguenti parametri d'ingresso:

- *node* - puntatore alla struttura dati Node del nodo che sta settando il timer.
- *inp* - puntatore alla struttura dati inpcbtpa che rappresenta l'istanza della connessione a cui si riferisce il timer.
- *delay* - valore con cui settare il timer.
- *timerType* - tipo del timer.

Il compito di questa funzione è quello di creare e spedire un messaggio di timer. Come campo *info* per tale messaggio viene usata la struttura dati TpaTimerPacket. Tale funzione torna il puntatore al messaggio spedito.
La funzione tpa_input( ) riceve in ingresso i seguenti parametri:

- *node* - puntatore alla struttura dati Node del nodo che ha ricevuto il pacchetto.
- *msgRicevuto* - puntatore al messaggio che contiene il pacchetto ricevuto.
- *p_head* - puntatore alla testa della lista delle connessioni.
- *tpa_stat* - puntatore alla struttura dati tpaStatGlobal contenente le statistiche del TPA.

Tale funzione recupera la struttura dati del TPA e tramite la funzione in_pcbtpalookup( ) guarda se esiste la connessione per il pacchetto appena ricevuto. L'ultimo parametro di tale funzione viene settato a INPCBTPA_NO_WILDCARD. Se la connesione per tale pacchetto non esiste, viene usata ancora la funzione in_pcbtpalookup( ), con l'ultimo parametro settato a INPCBTPA_WILDCARD, per vedere se è stata fatta una listen sul socket <*destinationAddr*, *destinationPort*> del pacchetto. Se è stata fatta una listen ci si prepara ad eseguire il 3-way handshake per aprire la connessione. In particolare si guarda se il pacchetto ricevuto è un pacchetto di SYN. Se non lo è si scarta il pacchetto ricevuto, altrimenti si crea la struttura dati per la connessione che si sta aprendo, usando la funzione tpa_attach( ). A questo punto si prende la struttura dati tpacb relativa alla nuova connessione, si inizializzano i suoi campi, si va in TPA_SYN_RCVD state e si spedisce il pacchetto di SYNACK. Al campo *ackWinSeqNumber* della struttura dati tpacb viene assegnato il valore *winSeqNumber* + 1, in quanto i pacchetti dati verranno spediti sulla finestra successiva a quella di spedizione del SYN.
Se non è stata fatta nessuna listen sul sochet <*destinationAddr*, *destinationPort*> il pacchetto viene scartato.

Se viene ricevuto un pacchetto appartenente ad una connessione già esistente (per la quale le strutture dati sono gia state create), questo viene gestito in maniera differente a seconda dello stato della connessione. Guardiamo come:

- TPA_CLOSED - la connessione è chiusa e quindi si scarta il pacchetto.
- TPA_SYN_SENT - in questo stato si deve ricevere il pacchetto di SYNACK per aprire la connessione. Quindi se il pacchetto è di SYNACK, si controlla che l'ACK si riferisca al pacchetto di SYN spedito e se è così, si spedisce l'ACK per il SYN ricevuto e si entra in TPA_ESTABLISHED state. Inoltre viene informata la applicazione dell'apertura della connessione (con una active open se la applicazione aveva fatto la richiesta di apertura della connessione, con una passive open se la applicazione aveva fatto una listen). La connessione viene infine marcata come attiva. Per finire viene calcolato l'ERTT del pacchetto di SYN, viene stimata la deviazione standard e viene chiamata la funzione calcolaRTO( ) per stimare il RTO del prossimo pacchetto da spedire.
- TPA_SYN_RCVD - se la connessione si trova in questo stato, significa che è stato ricevuto un pacchetto di SYN ed è stato spedito il SYNACK di risposta. Quindi il TPA si aspetta di ricevere o il pacchetto di ACK per il SYN spedito o un pacchetto di SYN duplicato. Se viene ricevuto un ACK, si guarda se effettivamente si riferisce al SYN spedito. Se è cosi si informa la applicazione dell'apertura della connessione (con una passive open se era stata fatta una listen o con una active open altrimenti). A questo punto si marca la connessione come attiva, si passa in TPA_ESTABLISHED state, si calcola il RTO per il prossimo pacchetto da spedire e si aggiornano i campi della struttura tpacb. Se invece viene ricevuto un SYN duplicato si suppone che la risposta al SYN ricevuto in precedenza sia andata persa e si rispedisce il SYNACK. In tale stato può capitare che l'ACK del SYN spedito non arrivi e che arrivino dei pacchetti dati. Questo può succedere, ad esempio, se il pacchetto di ACK relativo al pacchetto di SYNACK spedito viene perso. In tale situazione, il capo della connessione che ha ricevuto il

SYNACK considera la connessione aperta e comincia la spedizione dei dati. L'altro capo della connessione, non riceve nessun ACK per il SYN spedito ma si vede arrivare dei dati. In seguito alla ricezione dei pacchetti dati viene dato per perso l'ACK del SYN e viene apertala connessione, informando di questo la applicazione. Su ricezione del pacchetto dati si chiama la funzione tpa_receivePck( ) per processarlo.

- TPA_ESTABLISHED - in tale stato la connessione è aperta e vengono gestiti i seguenti pacchetti: i) Se arriva un SYN-ACK che si riferisce alla connessione già aperta (perche è scattato il timer presso il Sender), viene dato per perso l'ACK di risposta spedito in precedenza e viene rispedito; ii) Se arriva un pacchetto di SYN, si considera questo come un SYN duplicato (è scattato il timer presso il Sender del pacchetto) e si scarta tale pacchetto, dato che la connessione è già aperta da entrambi i lati (se sono in established vuol dire che ho gia ricevuto il SYN, spedito il SYNACK e ricevuto l'ACK del SYN); iii) se arriva un pacchetto dati o ACK (con l'ACK che si riferisce ad un pacchetto dati), viene chiamata la funzione tpa_receivePck( ) per gestirlo; iv) se arriva un pacchetto di FIN, viene chiuso un capo della connessione, viene spedito l'ACK del FIN, e la connessione entra in TPA_CLOSE_WAIT state. Inoltre viene informata di questo l'applicazione, attraverso una passive close.
- TPA_FIN_WAIT_1 - in questo stato non si possono più spedire dati (la applicazione ha chiuso il suo capo della connessione) ma si possono ancora ricevere pacchetti dati dall'altro capo della connessione. Di conseguenza dobbiamo continuare a spedire gli ACK per tali pacchetti. Guardiamo che tipo di pacchetti sono gestiti in tale stato: i) Su ricezione di un SYNACK duplicato viene considerato perso il pacchetto di ACK del SYN e viene rispedito; ii) su ricezione di un pacchetto dati o ACK (con l'ACK che si riferisce ad un pacchetto dati), viene chiamata la funzione tpa_receivePck( ) per gestirlo; iii) Su ricezione di un ACK che si riferisce al FIN spedito si passa in TPA_FIN_WAIT_2 state; iv) Su ricezione di un FIN si passa in

TPA_CLOSING state e viene spedito l'ACK per il FIN ricevuto; v) Su ricezione di un pacchetto di FINACK si guarda se l'ACK si riferisce al FIN spedito. Se è così si passa in TPA_TIME_WAIT state e viene spedito l'ACK per il FIN ricevuto. Inoltre viene settato il timer di chiusura ad un valore di 5 secondi.

- TPA_FIN_WAIT_2 - in questo stato non si possono più spedire dati (la applicazione ha chiuso il suo capo della connessione) ma si possono ancora ricevere pacchetti dati dall'altro capo della connessione. Di conseguenza dobbiamo continuare a spedire gli ACK per tali pacchetti. Guardiamo che tipo di pacchetti sono gestiti in tale stato: i) su ricezione di un pacchetto dati o ACK (con l'ACK che si riferisce ad un pacchetto dati), viene chiamata la funzione tpa_receivePck( ) per gestirlo; ii) Su ricezione di un ACK che si riferisce al FIN spedito non si fa niente; iii) Su ricezione di un FIN si passa in TPA_TIME_WAIT state, viene spedito l'ACK per il FIN ricevuto e viene settato il timer di chiusura ad un valore di 5 secondi.
- TPA_LAST_ACK - in questo stato il TPA è in attesa di ricevere l'ACK del FIN spedito e di chiudere la connessione. Su ricezione di tale ACK il TPA passa in TPA_CLOSED state e viene chiamata la funzione tpa_close( ) per deallocare le strutture dati relative alla connessione appena chiusa.
- TPA_CLOSING - in questo stato il TPA è in attesa di ricevere l'ACK del FIN spedito e di chiudere la connessione. Su ricezione di tale ACK il TPA passa in TPA_TIME_WAIT state e viene settato il timer di chiusura al valore di 5 secondi.
- TPA_TIME_WAIT - in questo stato il TPA è in attesa dello scattare del timer di chiusura. Su ricezione di un FIN duplicato viene rispedito l'ACK per tale FIN.
- TPA_CLOSE_WAIT - in questo stato vengono presi in considerazione i seguenti pacchetti: i) Su ricezione di un pacchetto di FIN duplicato viene rispedito il relativo ACK; ii) Su ricezione di un pacchetto di ACK per i dati spediti, viene chiamata la funzione tpa_receivePck( ) per gestirlo.

La funzione tpa_FillBuffer( ) serve per riempire il buffer di spedizione con il blocco di dati proveniente dall'applicazione. Ha i seguenti parametri d'ingresso:

- *node* - puntatore alla struttura dati Node del nodo che deve spedire i pacchetti.
- *phead* - puntatore alla testa della lista delle connessioni.
- *conn_i* - identificatore della connessione da cui provengono i pacchetti.
- *msgFromApp* - messaggio contenente il pacchetto da inserire nel buffer.
- *tpa_stat* - puntatore alla struttura dati relativa alle statistiche.

Tale funzione ha il seguente comportamento. Inizialmente, tramite la funzione in_pcbtpasearch( ), cerca la struttura dati relativa alla connessione a cui si riferisce il pacchetto da inserire nel buffer. Se questa è presente, il pacchetto dati ricevuto viene diviso in pacchetti di dimensioni pari a MSS byte e tali pacchetti vengono inseriti nel buffer. Inoltre vengono aggiustati i campi della struttura dati circularBuffer. Per finire il TPA passa nello stato di trasmissione TxW3_STATE e viene chiamata la funzione tpa_send( ) per dare inizio alla trasmissione del blocco dati appena ricevuto.

La funzione tpa_send( ) serve per spedire i pacchetti contenuti nel send buffer. Ha i seguenti parametri d'ingresso:

- *node* - puntatore alla struttura dati Node del nodo che deve spedire i pacchetti.
- *tp* - puntatore alla struttura dati tpacb della connessione a cui appartengono i pacchetti da spedire.
- *tpa_stat* - puntatore alla struttura dati relativa alle statistiche.

Tale funzione è implementata tramite uno statement switch che opera sul campo *tx_state* della struttura tpacb relativa alla connessione a cui appartengono i pacchetti da spedire. Se il TPA si trova in TxW3_STATE, i pacchetti vengono trasmessi in modo da avere al massimo CWNDmax pacchetti in volo. Se il TPA si trova in TxW1_STATE i pacchetti vengono trasmessi con un procedimento di tipo stop-and-wait. In altre parole, la trasmissione di un pacchetto viene schedulata dalla ricezione dell'ACK per un pacchetto precedente o dallo scattare del timeout

del pacchetto spedito in precedenza. Per trovare il pacchetto da spedire viene scandito il buffer dalla posizione dell'ultimo pacchetto spedito fino a trovare un pacchetto non acknoleggiato con il timer non attivo. A questo punto viene chiamata la funzione tpa_output( ) per spedire tale pacchetto.

La funzione tpa_receivePck( ) viene chiamata su ricezione di un pacchetto proveniente dal livello IP e gestisce il sendBuffer ed il receiveBuffer. Ha i seguenti parametri d'ingresso:

- *node* - puntatore alla struttura dati Node del nodo che deve spedire i pacchetti.
- *tp* - puntatore alla struttura dati tpacb della connessione a cui appartengono i pacchetti da spedire.
- *tpaStat* - puntatore alla struttura dati relativa alle statistiche.
- *msg* - puntatore al messaggio contenente il pacchetto ricevuto.

Tale funzione si comporta nella seguente maniera. Guarda se il pacchetto ricevuto contiene dei dati: se il pacchetto dati si riferisce ad una finestra vecchia significa che il Sender non ha ancora collezionato tutti gli ack ma il Receiver ha ricevuto tutti i dati ed ha gia incrementato la finestra di ricezione. Viene inviato al Sender un ACK con la bitmap con tutti i bit settati ad uno. Se la finestra del pacchetto ricevuto è quella corrente, viene aggiustato il receiveBuffer e viene spedito il pacchetto di ACK relativo al pacchetto dati ricevuto. Non appena il Receiver ha collezionato tutti i pacchetti del blocco di spedizione, questi vengono passati alla applicazione e viene aggiustato il receive buffer per permettere la ricezione di un nuovo blocco. Se il pacchetto ricevuto è un pacchetto di ACK per i dati spediti, viene calcolato il SRTT di tale pacchetto e vengono svolte le seguenti azioni: i) se il TPA si trova in TxW1_state e il numero di ACK ricevuti è pari a CONTACK il TPA passa in TxW3_STATE; ii) viene gestito il calcolo del RTO, guardando se il pacchetto è stato ricevuto dentro o fuori l'intervallo determinato dalla varianza, passando da RTOCLASSIC a RTOMOD (e viceversa) quando è necessario; iii) viene calcolato il RTO con la funzione calcolaRTO( ); iv) viene aggiornato il send buffer; v) se il blocco è tutto acknoleggiato, viene informa la applicazione della

spedizione del blocco e il TPA si prepara alla spedizione di un nuovo blocco; vi) se sono ancora presenti dei pacchetti non acknoleggiati viene chiamata la tpa_send( ) per spedirli.

La funzione tpa_disconnect( ) viene chiamata su richiesta dell'utente per chiudere la connessione. Ha i seguenti parametri d'ingresso:

- *node* - puntatore alla struttura dati Node del nodo che deve spedire i pacchetti.
- *phead* - puntatore alla lista delle connessioni.
- *conn_id* - identificatore della connessione da chiudere.
- *tpaStat* - puntatore alla struttura dati relativa alle statistiche.

Tale funzione si comporta nella seguente maniera. Cerca la struttura dati che si riferisce alla connessione *conn_id* tramite la funzione in_pcbtpasearch( ) e marca la connessione come chiusa (INPCBTPA_USRREQ_CLOSE). Chiama la funzione tpa_usrclosed( ) per decidere le operazioni da compiere (tale funzione decide le operazioni da compiere per la chiusura in funzione dello stato della connessione), comincia un nuovo blocco per la spedizione del FIN e chiama la tpa_output( ).

La funzione tpa_close ( ) viene chiamata per deallocare le strutture dati relative ad una connessione. Ha i seguenti parametri di ingresso:

- *node* - puntatore alla struttura dati Node del nodo che deve deallocare le strutture.
- *tp* - puntatore alla struttura dati tpacb della connessione che si deve deallocare.
- *tpaStat* - puntatore alla struttura dati relativa alle statistiche.

Tale funzione dealloca la struttura dati tpacb, informa l'applicazione della chiusura della connessione e chiama la funzione in_pcbtpadetach( ) per rimuove la struttura dati *inpcbtpa* dalla coda e deallocare la memoria relativa a questa. Ritorna un puntatore nullo.

La funzione calcolaRTO( ) calcola il RTO del prossimo pacchetto da spedire. Ha i seguenti parametri:

- *SRTT* - è il SRTT dell'ultimo pacchetto ricevuto dal TPA.

- *tp* - puntatore alla struttura dati relativa alla connessione di cui stiamo calcolando il RTO.

La funzione TransportTpaFinalize( ) è chiamata alla fine della simulazione per stampare le statistiche. Dealloca la struttura dati relativa al TPA.

Per finire vengono implementate le seguenti funzioni da utilizzare al livello applicazione per la gestione di una connesione TPA (contenute nel file QUALNET_HOME/application/app_util.c).

La funzione APP_TpaOpenConnectionWithPriority( ) serve per aprire una connessione TPA avente una priorità specifica. Ha i seguenti parametri:

- *node* - puntatore alla struttura dati Node del nodo che sta aprendo la connessione.
- *appType* - indica il tipo di applicazione che sta aprendo la connessione.
- *localAddr, localPort* - indirizzo IP e numero di porta della applicazione che vuole aprire la connesione.
- *remoteAddr*, *remotePort* - indirizzo IP e numero di porta del server.
- *uniqueId* - intero usato per identificare il cliente che sta effettuando la richiesta di apertura.
- *waitTime* - intervallo dopo cui deve cominciare la sessione.
- *priority* - priorità dei dati della connessione.

La funzione APP_TpaServerListen ( ) viene utilizzata dal server per mettersi in attesa di ricevere richieste di apertura di connessione. Ha i seguenti parametri:

- *node* - puntatore alla struttura dati Node del nodo che sta effettuando la listen.
- *appType* - indica il tipo di applicazione che ha effettuato la listen.
- *serverAddr* - indirizzo IP del server.
- *serverPort* - numero di porta del server.

La funzione APP_TpaCloseConnection( ) viene chiamata per chiudere una connessione TPA. Ha i seguenti parametri:

- *node* - puntatore alla struttura dati Node del nodo che sta effettuando la close.

- *connId* - identificatore della connessione che si deve chiudere.

La funzione APP_TpaSendData( ) serve per spedire un dato lungo la connessione TPA. Ha i seguenti parametri:

- *node* - puntatore alla struttura Node del nodo che sta effettuando la spedizione.
- *connId* - identificatore della connessione lungo cui vanno spediti i dati.
- *Payload* - dato da spedire.
- *length* - lunghezza in byte del dato da spedire.
- *traceProtocol* - specifica il tipo di applicazione usata per il packet tracing.

5.2 Realizzazione dell'applicazione FTP/TPA

L'applicazione FTP/TPA (file ftpTpa.c, ftpTpa.h), è un'applicazione che utilizza, per la spedizione dei pacchetti, il protocollo di trasporto TPA. E' stata implementata seguendo lo schema dell'applicazione GEN/FTP presente in QualNet. Con quest'applicazione, il cliente spedisce i dati al server senza ricevere nessun'informazione di controllo. Per usare il FTPTPA è necessario inserire nel file di configurazione (.app) dell'applicazione una stringa con il seguente formato:

```
FTPTPA <src> <dest> <items to send> <item size> <start time> <end time> [tos]
```

I vari campi della stringa di configurazione hanno il seguente significato:

- <src> rappresenta il nodo dove gira il lato client dell'applicazione.
- <dest> rappresenta il nodo dove gira il lato server dell'applicazione.
- <items to send> rappresenta il numero di blocchi da spedire.
- <item size> rappresenta la dimensione di ciascun blocco. Deve avere un valore pari ad 12 * MSS.
- <start time> rappresenta l'istante d'inizio dell'attività dell'applicazione FTPTPA durante la simulazione.
- <end time> rappresenta l'istante di terminazione di FTPTPA durante la simulazione.

- [tos] questo campo è opzionale e rappresenta il contenuto che deve avere il campo TOS dell'header dei pacchetti IP.

Se <items to send> è settato a zero, FTPTPA continuerà a spedire pacchetti fino all'istante di <end time> o fino alla fine della simulazione, a seconda di quale evento si verifica per primo. Se <end time> è settato a zero, FTPTPA continuerà la sua attività fino a quando non saranno stati trasmessi <items to send> pacchetti o fino alla fine della simulazione, a seconda di quale evento si verifica per primo. Se <items to send> e <end time> sono entrambi settati a un valore maggiore di zero, FTPTPA continuerà la sua attività fino alla spedizione di tutti gli <items to send> pacchetti, fino al raggiungimento del <end time> o fino alla fine della simulazione, a seconda di quale evento scatta si verifica per primo. Il funzionamento del FTPTPA è il seguente. Dopo start time secondi dall'inizio della simulazione, la applicazione parte con la fase di apertura della connessione. Una volta terminata, comincia la spedizione dei blocchi dati verso il TPA. Il client spedisce un blocco di dimensione pari a 12 * MSS al Server e si mette in attesa dal TPA della notifica di spedizione avvenuta. Su ricezione di questa, rispedisce un altro blocco di dati.

Esaminiamo le principali strutture dati e funzioni che modellano il comportamento dell'applicazione FTPTPA.

La struttura AppDataFtpTpaClient rappresenta un'istanza del client della applicazione. Ha i seguenti campi:

- *connectionId* - identificatore della connessione della istanza del client.
- *localAddr* - indirizzo IP locale.
- *remoteAddr* - indirizzo IP del server.
- *sessionStart* - istante di inizio della sessione.
- *sessionFinish* - istante di terminazione della sessione.
- *sessionIsClosed* - se vale TRUE significa che sono stati spediti il numero di blocchi specificato nella stringa di configurazione.
- *itemSize* - rappresenta la dimensione in byte del blocco di spedizione.
- *itemsToSend* - indica il numero di blocchi che si devono ancora spedire.
- *itemsSent* - indica il numero di blocchi spediti. E' una statistica.

- *numBytesSent* - indica il numero di Byte spediti. E' una statistica.
- *lastItemSent* - indica l'istante in cui la applicazione viene a sapere dell'avvenuta spedizione da parte del TPA dell'ultimo blocco.
- *uniqueId* - identifica una particolare istanza del client.
- *endTime* - indica l'istante di terminazione dell'attività della applicazione.
- *timeSentLastItem* - indica l'istante di spedizione dell'ultimo blocco.
- *tempoSpedizione* - indica il tempo impiegato per la spedizione di un blocco di dati.
- *tempoMedioSpedizione* - il valore di tale campo, diviso per il numero di items spediti, rappresenta il tempo medio impiegato per la spedizione dei blocchi.

La struttura AppDataFtpTpaServer rappresenta un'istanza del server della applicazione. Ha i seguenti parametri:

- *connectionId* - identifica la connessione a cui si riferisce questa particolare istanza del server.
- *localAddr* - indirizzo IP locale della connessione.
- *remoteAddr* - indirizzo IP del client.
- *sessionStart* - istante di inizio della sessione.
- *SessionFinish* - istante di terminazione della sessione.
- *sessionIsClosed* - indica se la sessione e' chiusa.
- *lastItemRecvd* - indica l'istante di ricezione dell'ultimo pacchetto.
- *itemsReceived* - indica il numero di blocchi ricevuti. E' una statistica.
- *numBytesRecvd* - indica il numero di byte ricevuti. E' una statistica.

La funzione AppFtpTpaClientInit( ) inizializza una sessione dell'applicazione. Ha i seguenti parametri:

- *node* - puntatore alla struttura dati del nodo.
- *clientAddr* e *serverAddr* - indirizzi IP del client e del server.
- *itemsToSend* - numero di blocchi da spedire.
- *itemSize* - dimensione del blocco dati da spedire.

- *startTime* - ritardo di inizio della sessione.
- *endTime* - istante di fine sessione.
- *tos* - contenuto per il campo type of service del pacchetto IP.

Si comporta nella seguente maniera. Appena viene chiamata, invoca la funzione AppFtpTpaClientNewFtpTpaClient( ) per creare una nuova istanza della struttura dati AppDataFtpTpaClient. Successivamente, se la memoria è stata allocata correttamente, invoca la funzione APP_TpaOpenConnectionWithPriority( ) per richiedere l'apertura della connessione. Tale funzione, definita nel file app_util.c, richiede in ingresso il socket del client e il socket del server. Per assegnare il numero di porta al client viene presa ed incrementata la variabile node->appData.nextPortNum. Come numero di porta del server si usa APP_FTPTPA_SERVER.

La funzione AppFtpTpaClientNewFtpTpaClient( ) crea una nuova istanza della struttura dati AppDataFtpTpaClient e la inserisce all'inizio della lista contenete le strutture dati delle varie applicazioni. Ha i seguenti parametri:

- *node* - puntatore alla struttura dati del nodo.
- *clientAddr* - indirizzo IP del client.
- *serverAddr* - indirizzo IP del server.
- *itemsToSend* - numero di blocchi che devono essere spediti durante la simulazione.
- *itemSize* - dimensione in byte di ciascun blocco.
- *endTime* – istante di terminazione della applicazione FTPTPA.

Ritorna il puntatore alla struttura dati creata. Una volta allocata la memoria per la struttura dati AppDataFtpTpaClient, vengono inizializzati i suoi campi. Il campo *uniqueId* viene inizializzato con il valore della variabile node->appData.uniqueId incrementato di uno. Tale variabile, tutte le volte che viene creata una istanza di una applicazione deve essere incrementata di uno. Per ultimo, viene chiamata la funzione APP_RegisterNewApp( ). Tale funzione serve per inserire la struttura dati creata nella lista delle applicazioni del nodo dove risiede l'applicazione.

La funzione AppLayerFtpTpaClient( ) modella il comportamento dell'applicazione su ricezione di un evento. Ha i seguenti parametri:

- *node* - puntatore alla struttura dati del nodo che riceve il messaggio.
- *msg* - puntatore alla struttura dati del messaggio ricevuto.

E' implementata tramite uno statement switch che lavora sul campo *eventType* del messaggio ricevuto. Esaminiamo il comportamento della funzione su ricezione dei vari tipi d'evento:

- MSG_APP_FromTransOpenResult - se il messaggio ricevuto è di active open, viene esaminato il campo *connectionId* del campo *info* del messaggio ricevuto, per vedere se la connessione è stata aperta oppure no. Se la connessione non è stata aperta, viene richiamata la funzione APP_TpaOpenConnectionWithPriority( ), altrimenti, tramite la funzione AppProvaTpaClientUpdateProvaTpaClient( ), viene ritrovata ed aggiornata la struttura dati relativa al client specificato dal campo *uniqueId* del campo *info* del messaggio. Successivamente viene chiamata la funzione AppFtpTpaClientSendNextItem( ) per spedire un blocco.
- MSG_APP_FromTransCloseResult - su ricezione di tale messaggio viene chiamata la funzione AppFtpTpaClientGetFtpTpaClient( ) per ricercare la struttura dati della applicazione identificata dal campo *connectioId* del campo *info* del messaggio ricevuto e per settare il campo *sessionIsClosed* di tale struttura a TRUE (per indicare che la spedizione dei blocchi è terminata).
- MSG_APP_FromTransDataSent - su ricezione di tale messaggio, viene chiamata la funzione AppFtpTpaClientGetFtpTpaClient( ) per cercare la struttura dati della applicazione identificata dal campo *connectioId* del campo *info* del messaggio ricevuto e aggiornarne i campi. Successivamente viene esaminata la variabile clientPtr->sessionIsClosed appartenente al client a cui è indirizzato l'evento per vedere se la spedizione dei dati è terminata oppure no. Se ci sono ancora dei blocchi da spedire viene chiamata la funzione AppFtpTpaClientSendNextItem( ), altrimenti viene chia-

mata la funzione APP_TpaCloseConnection( ) per chiudere la connessione. Tale funzione invia un messaggio di close al TPA.

La funzione AppFtpTpaServerInit( ) esegue l'operazione di listen tramite la funzione APP_TpaServerListen(*)*. Il server si viene così a trovare in ascolto sulla porta APP_FTPTPA_SERVER.

La funzione AppFtpTpaServerNewFtpTpaServer( ) serve per creare una nuova istanza della struttura dati AppDataFtpTpaServer e per inserirla all'inizio della lista contenente le strutture dati delle varie applicazioni che si trovano sul nodo in questione. Ha i seguenti parametri:

- node - puntatore alla struttura dati del nodo dove risiede il server.
- openResult - struttura dati contenente il risultato della richiesta di apertura della connessione.

Tale funzione torna il puntatore alla struttura dati creata o un puntatore NULL, nel caso in cui non è stato possibile allocare la memoria. Una volta allocata la memoria per la struttura dati AppDataFtpTpaServer, vengono inizializzati i suoi campi e viene chiamata la funzione APP_RegisterNewApp( ). Tale funzione serve per inserire la struttura dati creata nella lista delle applicazioni del nodo dove risiede l'applicazione.

La funzione AppLayerFtpTpaServer( ) modella il comportamento del server su ricezione di un messaggio. Ha i seguenti parametri:

- *node* - puntatore alla struttura dati del nodo che riceve il messaggio.
- *msg* - puntatore alla struttura dati del messaggio ricevuto.

E' implementata tramite uno statement switch che lavora sul campo *eventType* del messaggio ricevuto. Esaminiamo il comportamento della funzione su ricezione dei vari tipi d'evento:

- MSG_APP_FromTransListenResult - controlla che la listen sia stata eseguita correttamente, tramite il campo *connectionId* del campo *info* del messaggio ricevuto. Se tale campo vale -1 viene richiamata la funzione APP_TpaServerListen( ).

- MSG_APP_FromTransOpenResult - tale evento riporta il risultato della apertura della connessione. Se la connessione è stata aperta con successo (il campo *connectionId* del campo *info* del messaggio ricevuto ha un valore >= 0), viene invocata la funzione AppFtpTpaServerNewFtpTpaServer( ).
- MSG_APP_FromTransDataReceived - tale evento indica la ricezione di un pacchetto proveniente dal protocollo TPA. Su ricezione di tale evento, viene invocata la funzione AppFtpTpaServerGetFtpTpaServer( ) per ricavare la struttura dati dell'istanza del server a cui è destinato il pacchetto (tramite il campo *connectionId* del campo *info* del messaggio ricevuto) e vengono aggiornate in modo opportuno le statistiche.
- MSG_APP_FromTransCloseResult - tale evento viene ricevuto o quando il client ha finito di spedire i blocchi ed invoca la close per chiudere la connessione o quando la connessione risulta effettivamente chiusa. Su ricezione di tale messaggio viene invocata la funzione AppProvaTpaServerGetProvaTpaServer( ) per ricavare la struttura dati dell'istanza del server a cui è destinato il messaggio. Se la sessione non è ancora chiusa viene invocata la funzione APP_TpaCloseConnection( ) per chiudere anche il lato server della applicazione.

Capitolo VI.

Valutazione del protocollo TPA

Il protocollo TPA è stato valutato nell'ambiente di sviluppo e simulazione Qual-Net Developer. Nelle varie simulazioni effettuate, sono stati utilizzati i seguenti protocolli:

- come protocollo di routing è stato utilizzato il protocollo AODV.
- come protocollo di livello MAC è stato utilizzato quello definito nello standard 802.11b. Lo standard 802.11 implementato in QualNet ha i seguenti valori per i raggi di trasmissione (calcolati sperimentalmente):

$$TX\_range = 378 metri$$
$$CS\_range \cong 860 metri$$
$$IF\_range \max \cong 750 metri$$

- Il protocollo TCP utilizzato è il TCP RENO con un valore per il MSS pari a 1024 bytes.
- Per quanto riguarda il TPA, abbiamo utilizzato una finestra di controllo della congestione con dimensione massima pari a 3 pacchetti. Tale scelta è stata fatta basandosi sugli studi effettuati in [9] [7] [16]. Da questi studi, risulta che la dimensione ottima per la finestra di controllo del TCP dipende dalla lunghezza in hop della connessione, dalla topologia della rete e dalla presenza di altre connessioni attive. Per connessioni non troppo lunghe (come nella realtà sono) con topologia a stringa, viene mostrato che la finestra ottimale si mantiene attorno ad un numero basso di pacchetti. In [16], viene mostrato come una finestra di 4 pacchetti si comporti in modo ottimo per connessioni fino a 7 hop (una unica connessione di tipo a stringa). L'unica anomalia viene trovata per connessioni attorno ai 3-4 hops. In

questo caso, le performance sono un poco inferiori rispetto a quelle che si otterrebbero con una finestra di 1 pacchetto. In funzione di questi risultati, abbiamo deciso di utilizzare una finestra pari a 3 in quanto, la differenza tra 3 e 4 pacchetti è piccola ed il nostro meccanismo di trasmissione funziona bene se utilizza una finestra di piccole dimensioni. La dimensione del blocco l'abbiamo settata al valore di 12 pacchetti. Anche per il TPA abbiamo usato un valore per il MSS pari a 1024 bytes.

- Per quanto riguarda le applicazioni utilizzate, per il TCP abbiamo scelto la GEN_FTP, mentre per il TPA abbiamo scelto la FTP/TPA. Entrambe queste applicazioni sono state settate per spedire un totale di circa 6Mbytes di dati.

6.1 Indici di prestazione

Come indici di prestazione per la valutazione della bontà del protocollo TPA, abbiamo deciso di utilizzare il throughput e l'indice di ritrasmissione. Il throughput, viene calcolato a livello applicazione come il rapporto tra il numero di bit inviati ed il tempo impiegato per la spedizione. Tale parametro fornisce un'indicazione sulla bontà della connessione.
L'indice di ritrasmissione, calcolato al livello trasporto, viene definito come segue:

$$indiceRitrasmissione = 1 - \left(\frac{totPckSent}{pckToSend} \right)$$

Dove *totPckSent* sono il numero totale di pacchetti dati spediti (compresi i pacchetti ritrasmessi) e dove *pckToSend* sono il numero di pacchetti che devono essere spediti (numero totale di pacchetti spediti meno il numero di pacchetti ritrasmessi). Tale indice ci da informazioni sul consumo energetico (più è elevato maggiore è il numero di pacchetti spediti e maggiore è il consumo energetico) e sul numero di pacchetti persi per contesa.

6.2 Risultati

Per testare la bontà del protocollo TPA sono stati eseguiti gli esperimenti riportati di seguito.

Nel primo esperimento è stata considerata una rete statica con topologia a stringa del tipo riportato in figura 36:

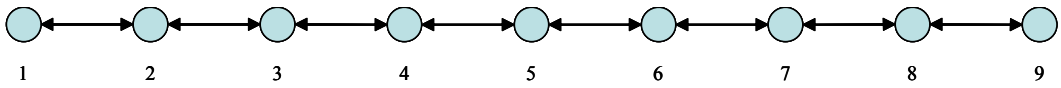

Figura 36 – rete con topologia a stringa.

I nodi sono stati posti ad una distanza di 300 metri l'uno dall'altro, in maniera che ciascun nodo fosse in grado di comunicare solo con i propri vicini (ad esempio, il nodo 4 è in grado di comunicare solo con i nodi 3 e 5). Con tale configurazione, i nodi distanti 2 hop da un nodo che sta spedendo, ricadono nel suo raggio di interferenza e nel suo CS_range. Su tale rete è stata settata una singola connessione TCP tra una specifica coppia di nodi, ed è stato misurato il throughput e l'indice di ritrasmissione al variare del numero di hop della connessione. La lunghezza della connessione TCP è stata fatta variare tra un minimo di 1 hop (connessione tra 1 e 2) ed un massimo di 8 hop (connessione tra 1 e 9). Tali misurazioni sono state ripetute utilizzando il protocollo di trasporto TPA.

Le misurazioni sul throughput dei due protocolli sono state messe a confronto in figura 37. Da tale figura risulta evidente che, non appena i problemi dell'hidden e dell'exposed node cominciano a farsi sentire, e cioè per lunghezze della connessione superiori a 3 hop, il throughput del TCP comincia a calare rispetto a quello del TPA. Ad esempio, per una connessione di 5 hop, il TCP mostra avere un throughput inferiore del 28% circa rispetto a quello del TPA. Le misurazioni effettuate sull'indice di ritrasmissione confermano quanto detto in precedenza (figura 38). Non appena la lunghezza della connessione supera i 3 hop, i fenomeni dell'exposed e dell'hidden node cominciano a farsi sentire, ed il TCP comincia a perdere vari pacchetti a causa della contesa. Questo porta ad un degrado del

throughput. Il TPA invece, spedendo i dati con una finestra di dimensione pari a 3 pacchetti, non sovraccarica la rete, riducendo al minimo il numero di pacchetti persi per contesa.

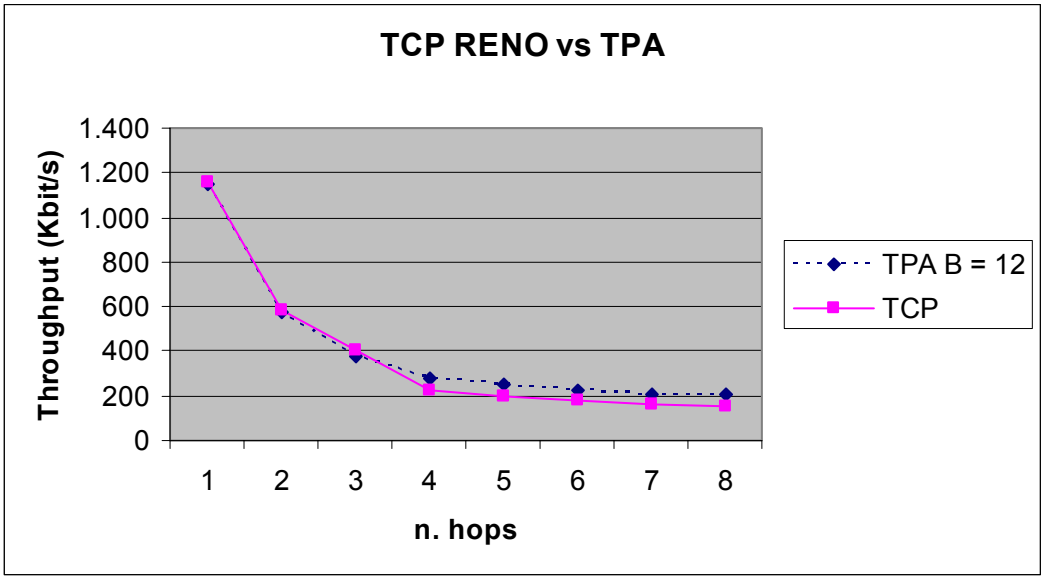

Figura 37 – Throughput TCP vs Throughput TPA.

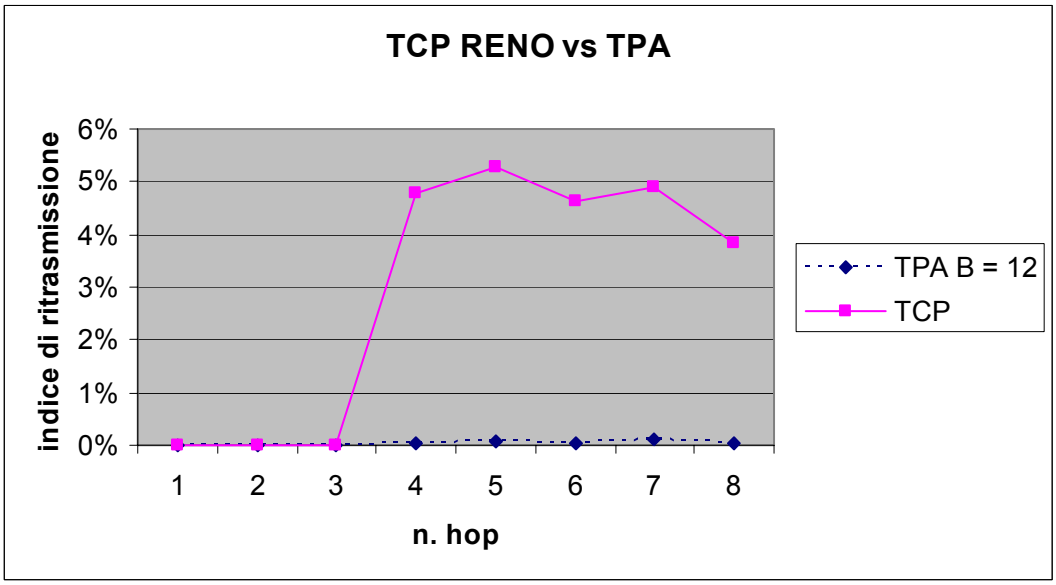

Figura 38 – indice di ritrasmissione TPA vs indice di ritrasmissione TCP.

Nel secondo esperimento è stata presa una rete statica con topologia a stringa del tipo riportato in figura 39.

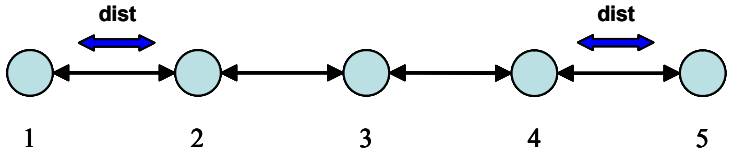

Figura 39 – rete con topologia a stringa e distanza tra i nodi variabile.

Su tale rete è stata settata una singola connessione TCP tra il nodo 1 ed il nodo 5 (connessione di 4-hop) ed è stato misurato il throughput e l'indice di ritrasmissione al variare della distanza tra i vari nodi. Più precisamente, sempre mantenendo l'equidistanza tra i nodi, la distanza *dist* tra le varie coppie di nodi è stata variata da un minimo di 50 ad un massimo di 400 metri. Tale esperimento è stato ripetuto utilizzando il protocollo TPA. Le misurazioni sul throughput dei due protocolli sono state messe a confronto in figura 40. Dall'analisi di tale figura possiamo giungere alle seguenti conclusioni:

- Per valori di *dist* inferiori a 300 metri i problemi di contesa non si fanno sentire ed il protocollo TPA mostra avere delle prestazioni equivalenti a quelle del TCP.
- Per valori di *dist* che vanno da 300 a 350 metri, i problemi dell'hidden e dell'exposed node cominciano a farsi sentire, ed il throughput del TCP risulta essere inferiore di circa il 28% rispetto al throughput del TPA.
- Non appena i nodi escono dal raggio di trasmissione dei propri vicini, il throughput arriva a zero.

Le misurazioni effettuate sull'indice di ritrasmissione confermano quanto detto in precedenza (figura 41). Non appena la distanza *dist* cresce sopra i 250 metri, i problemi dell'hidden e dell'exposed node cominciano a farsi sentire e il TCP comincia a perdere vari pacchetti per contesa (con conseguente degrado del throu-

ghput). Il TPA invece, avendo una finestra di dimensione massima pari a 3, non sovraccarica la rete, riducendo al minimo il numero di pacchetti persi per contesa.

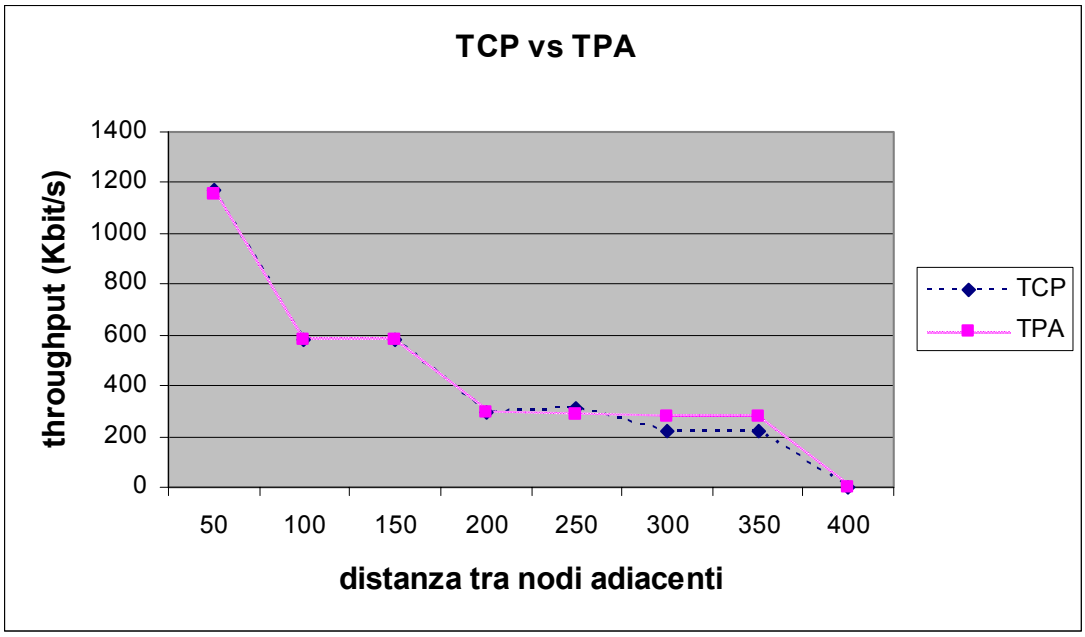

Figura 40 – Andamento del throughput al variare della distanza tra i nodi della connessione.

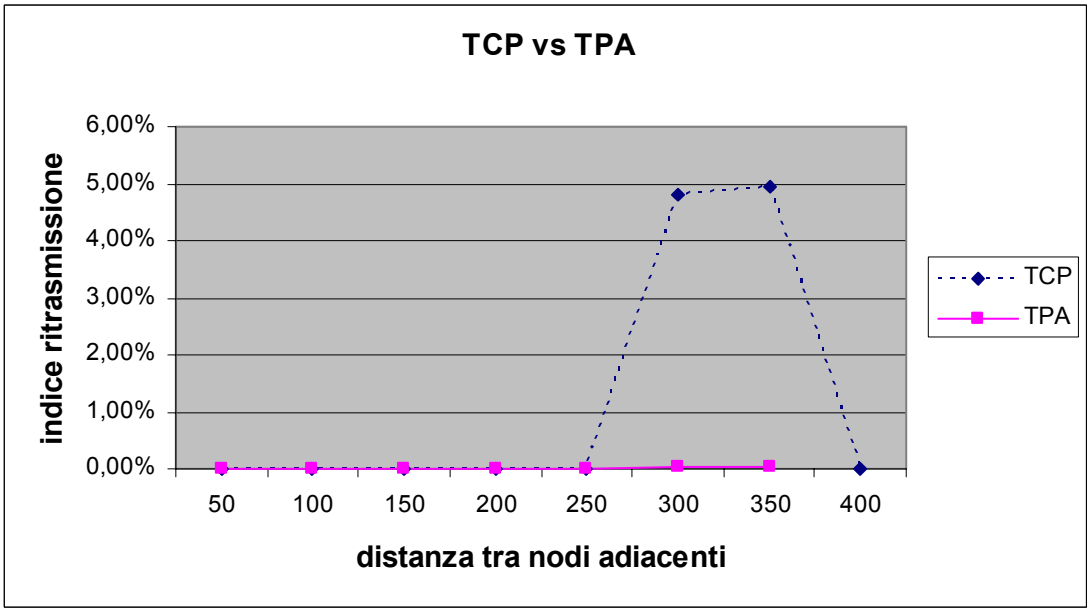

Figura 41 – Andamento dell'indice di ritrasmissione al variare della distanza tra i nodi.

Capitolo VII.

Conclusioni

Numerosi studi nel campo delle MANET hanno evidenziato come le prestazioni del protocollo TCP in tale ambiente risultino scadenti. Questo è dovuto alle forti differenze che esistono tra le MANET e le reti wired tradizionali (come Internet) per le quali il TCP è stato studiato. In particolare, nelle MANET la mobilità dei nodi può portare frequentemente a fenomeni di route change e route failure con conseguenti perdite di pacchetti. Questo può provocare, presso il sender TCP, lo scattare di diversi timeout per i pacchetti spediti. Inoltre, il fenomeno della congestione assume un aspetto diverso rispetto a quello che si verifica nelle reti wired tradizionali: la condivisione del mezzo trasmissivo da parte di molti nodi e la mancanza d'informazioni scambiate tra questi per ottimizzare tale condivisione, fa sì che numerosi pacchetti collidano e vadano persi. Entrambi questi fenomeni, non sono gestiti esplicitamente dal TCP e tipicamente causano in esso l'attivazione del meccanismo di controllo della congestione. L'invocazione di tale meccanismo, al presentarsi di eventi che nulla hanno a che fare con la congestione della rete, provocano un degrado delle prestazioni del TCP. In particolare si ha un basso throughput e numerose ritrasmissioni inutili. Tali fenomeni si verificano sia in situazioni dinamiche (nodi che si muovono) sia in situazioni statiche (assenza di mobilità dei nodi).

In questa tesi viene proposto un nuovo protocollo di trasporto (Transport Protocol for Ad Hoc o TPA) studiato appositamente per essere utilizzato in ambient MANET. Il TPA è in grado di gestire adeguatamente gli eventi di route change e route failure ed è dotato di un meccanismo di controllo della congestione completamente ridisegnato rispetto a l TCP. Oltre a questo, il TPA adotta una politica di trasmissione dei pacchetti in grado di ridurre il più possibile il numero di ritrasmis-

sioni non necessarie, al fine di di risparmiare energia e di ridurre il carico di lavoro offerto alla rete.

Il protocollo TPA è stato realizzato nell'ambiente di sviluppo e simulazione QualNet Developer. Durante questa tesi è stata svolta una analisi preliminare delle prestazioni del TPA nel caso statico. I risultati ottenuti hanno evidenziato che il TPA ha delle prestazioni migliori rispetto al TCP, in termini di throughput e di consumo energetico. Questo miglioramento è dovuto principalmente al nuovo meccanismo di controllo della congestione ed alla politica con cui vengono ritrasmessi i pacchetti considerati persi.

In futuro si prevede di estendere la valutazione delle prestazioni del protocollo TPA al caso dinamico. In particolare, sarà necessario analizzarne il comportamento in presenza di più connessioni attive nella rete ed al variare della topologia di quest'ultima.

Bibliografia

[1] *Kartik Chandran, Sudarshan Raghunathan, Subbarayan Venkatesan*, *Ravi Prakash*, "A feedback-based scheme for improving TCP performance in ad hoc wireless networks", *IEEE Personal Communications,* no. 1, February 2001 pp. 34-39

[2] *Dong Sun, Hong Man*, "ENIC--An improved reliable transport scheme for mobile ad hoc networks", *GLOBECOM 2001 - IEEE Global Telecommunications Conference,* no. 1, Nov 2001 pp. 2852-2856

[3] *Dongkyun Kim, C.-K. Toh, Yanghee Choi*, "TCP-BuS: Improving TCP performance in wireless ad-hoc networks", *ICC 2000 - IEEE International Conference on Communications,* no. 1, June 2000 pp. 1707-1713

[4] *Gavin Holland, Nitin Vaidya*, "analisis of TCP performance over Mobile ad hoc network", *Proceedings of IEEE/ACM MOBICOM '99*

[5] *T. D. Dyer R. V. Boppana*, "A comparison of TCP Performance over Three Routing Protocols for Mobile Ad Hoc Networks", *ACM Symposium on Mobile Ad Hoc Networking & Computing-Mobihoc*, October 2001.

[6] *F. Wang, Y. Zhang*, "Improving TCP Performance over Mobile Ad-Hoc Networks with Out-of-Order Detection and Response", *In Proceeding of Mobihoc'02*, June 2002.

[7] *Zhenghua Fu, Ben Greenstein, Xiaoqiao Meng, Songwu Lu,* "Design and Implementation of a TCP-Friendly Transport Protocol for Ad Hoc Wireless Networks," *to appear in IEEE ICNP'02*, 2002.

[8] *K. Xu, M. Gerla*,"TCP Over an IEEE 802.11 Ad Hoc Network: Unfairness Problem and Solutions", *Mobihoc'02*.

[9] *Zhenghua Fu, Petros Zerfos, Haiyun Luo, Songwu Lu, Lixia Zhang and Mario Gerla*, "On TCP performance in multihop wireless networks," *UCLA WiNG Technical Report*, 2002.

[10] *Mesut Günes, Donald Vlahovic,* "The performance of the TCP/RCWE Enhancement for Ad-Hoc Networks", *Seventh International Symposium on Computers and Communications (ISCC'02*) , July 01 - 04, 2002.

[11] *Jian Liu, Suresh Singh*, "ATCP: TCP for mobile ad hoc networks", *IEEE Journal on Selected Areas in Communications,* no. 7, July 2001 pp. 1300-1315.

[12] *Shugong Xu, Tarek Saadawi*, "Revealing the problem with 802.11 medium access control protocol in multi-hop wireless ad hoc networks", *Computer Networks 38* (2002), pp. 531-548.

[13] IEEE standard for Wireless LAN- Medium Access Control and Phisical Layer Specification, P802.11, November 1997.

[14] Supplement to IEEE std 802.11-1999

[15] Supplement to IEEE std 802.11-1999

[16] *Shugong Xu, Tarek Saadawi*, "Performance Evaluation of TCP Algorithms in Multi-hop Wireless Packet Networks",

[17] "Limitations of TCP-ELFN for Ad Hoc Networks"

[18] *W. Richard Stevens*. TCP/IP Illustrated Volume 1. Addison Wesley, 1994.

[19] *M. Allman,V. Paxson and W. Stevens.* TCP Congestion Control. *IETF RFC 2681*, April 1999.

[20] *V.D. Park* and *M.S. Corson*, "A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks", Proceedings *of IEEE INFOCOM '97*, Kobe, Japan, 1997.

[21] K. Xu, S. Bae, S. Lee and M.Gerla, \TCP Behavior accross Multihop WirelessNetworks and the Wired Internet", The Fifth International Workshop on Wireless Mobile Multimedia (WoWMoM 2002), Atlanta (GA), September 28, 2002.

[22] A.Bakre, B.R.Badrinath, \Implementation and Performance Evaluation of Indirect TCP", IEEE Transactions on Computers, Vol.46, No.3, March 1997.