UNIVERSITÀ DI PISA

FACOLTÀ DI SCIENZE MATEMATICHE,
FISICHE E NATURALI
DIPARTIMENTO DI INFORMATICA

CORSO DI LAUREA TRIENNALE IN INFORMATICA
(classe L-31)

PROVA FINALE

TCP BUFFERING IN BSD

Overview riguardante l'implementazione dei buffer di invio e ricezione con i relativi controlli di flusso e congestione

Candidato
Jacopo Soldani

Relatore
Antonio Brogi

Anno Accademico 2010 / 2011

Sommario

I meccanismi di buffering, controllo di flusso e controllo di congestione del protocollo TCP vengono comunemente introdotti astraendo da dettagli implementativi. L'obiettivo di questo documento è cercare di illustrare sinteticamente il modo in cui tali meccanismi sono effettivamente realizzati in una delle più note implementazioni di TCP (BSD), evidenziando alcune delle differenze rispetto alla descrizione di TCP fornita in uno dei testi [1] di introduzione alle reti attualmente più diffusi.

Indice

Parte I – Introduzione

1. Introduzione

1.1 Introduzione

Nel testo di Kurose e Ross [1] la trattazione dei buffer di spedizione e ricezione avviene dandone una concezione molto astratta: in pratica vengono considerate due porzioni di memoria indicizzate solamente per mezzo dei valori di *sendBase* (numero di sequenza del byte più vecchio inviato ma non ancora riscontrato), *nextSeqNum* (numero di sequenza assegnabile al prossimo byte da spedire), *lastByteRead* (numero di sequenza dell'ultimo byte nel flusso di dati letto a partire dal buffer da parte del processo applicativo) e *highestByteRcvd* (numero di sequenza dell'ultimo byte nel flusso di dati che proviene dalla rete e che è stato copiato nel buffer di ricezione). Ma come potrebbero essere realmente implementati questi buffer? Lo scopo di questo documento è appunto quello di mostrare una delle loro tante implementazioni. Inoltre sappiamo che, nella gestione dei suddetti buffer, sono presenti i controlli di flusso e congestione.
Questo capitolo provvede, perciò, ad introdurre il *networking code* della Berkeley. Cominceremo con una descrizione del metodo di presentazione del codice e delle varie convenzioni usate durante il testo.
Proseguiremo dando una breve collocazione storica alle varie release del sistema analizzato, per poi passare ad introdurre l'implementazione del sistema.

1.2 Convenzioni per la presentazione

Presentare migliaia di righe di codice è già di per sé un problema complesso. Nel testo perciò useremo il formato seguente per esplicitare tutto il codice sorgente:

```
------------------------------------------------------------tcp_subr.c
 381 void
 382 tcp_quench(inp,errno)
 383 struct inpcb *inp;
 384 int errno;
 385 {
 386    struct tcpcb *tp = intotcpcb(inp);

 387    if(tp)
 388       tp->snd_cwnd = tp->maxseg;
 389 }
------------------------------------------------------------tcp_subr.c
```

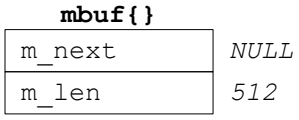
La prima e l'ultima riga della sezione contenente il codice riferiscono al file[1] in cui è definito (quella presentata è la funzione `tcp_quench` contenuta nel file

1 I nomi dei file sorgente riportati nel testo riferiscono a quelli contenuti nella distribuzione 4.4BSD-Lite

`tcp_subr.c`). Ogni riga contenente codice viene numerata.
Inoltre ogni capitolo che descrive codice sorgente solitamente inizia con un elenco dei file sorgente relativi, seguito dalle variabili globali e da tutte le altre informazioni necessarie ad una corretta comprensione del codice. Le variabili globali sono spesso definite attraverso più file sorgenti e header, perciò andremo a racchiuderle in una tabella unica per una consultazione più agevole.
Nelle tabelle utilizzeremo un font a dimensione costante per nomi di variabile e campi interni alle strutture, e lo stesso font sarà messo in corsivo per i nomi delle costanti definite. Ad esempio:

`m_flags`	Descrizione
`M_BCAST`	Spedito o ricevuto come broadcast di livello link

Solitamente mostreremo tutte le `#define` attraverso questa metodologia (riportando il valore del simbolo solamente se necessario) e le ordineremo alfabeticamente.
Per quanto riguarda le figure riportate in questo documento utilizzeremo un font a dimensione costante per nomi di variabili e campi delle strutture (`m_next`), un font corsivo a dimensione costante per rappresentare costanti (*`NULL`*) o valori (*`510`*), e un font grassetto a dimensione costante con parentesi graffe per nomi di strutture (**`mbuf{}`**).
Ecco un esempio:



1.3 Storia di BSD

Questo documento descrive l'implementazione dei meccanismi di spedizione e ricezione di TCP/IP implementati dal Computer System Research Group (CSRG) dell'Università di Berkeley (California). Stiamo parlando dei sistemi 4.x BSD (Berkeley Software Distribution) e delle "BSD Networking Releases", il cui codice sorgente è stato utilizzato come punto di partenza per molte altre implementazioni, sia per sistemi Unix sia per sistemi non-Unix.
Ma andiamo con ordine: la prima versione del sistema operativo BSD (1BSD) venne rilasciata dall'Università di Berkeley nel Marzo 1978. Si trattava di una serie di patch di uno studente del campus, Bill Joy. Il sistema fu distribuito, come si usava ai tempi, su di un nastro sotto forma di codice sorgente. Tale codice includeva, tra le altre cose, un editor di testi (ex) ed un compilatore Pascal.
L'anno successivo, sempre sotto la supervisione di Joy, venne rilasciata 2BSD che introduceva due innovazioni software sopravvissute fino ad oggi: vi (versione visuale di ex) e la C shell. Alla 2BSD seguì una terza versione (3BSD) il

cui successo portò la "Defense Advanced Research Project Agency" (DARPA) a finanziare il CSRG per la produzione di una piattaforma Unix standard. Nel Novembre del 1980, quindi, venne rilasciata 4BSD. Seguirono varie release fino alla 4.3BSD del 1986 (quando ormai Bill Joy aveva lasciato la leadership del progetto a McKusick e Karels per fondare la Sun Microsystems). L'implementazione della suite TCP/IP di 4.3BSD risultò talmente superiore a quella degli altri sistemi in commercio da spingere la DARPA a sceglierla come standard.
Nel Giugno 1994 venne poi rilasciata la definitiva 4.4BSD in due forme:

- 4.4BSD-Lite → gratuita (poiché priva di codice sorgente prodotto da AT&T);
- 4.4BSD-Encumbered → ottenibile solo con licenza AT&T.

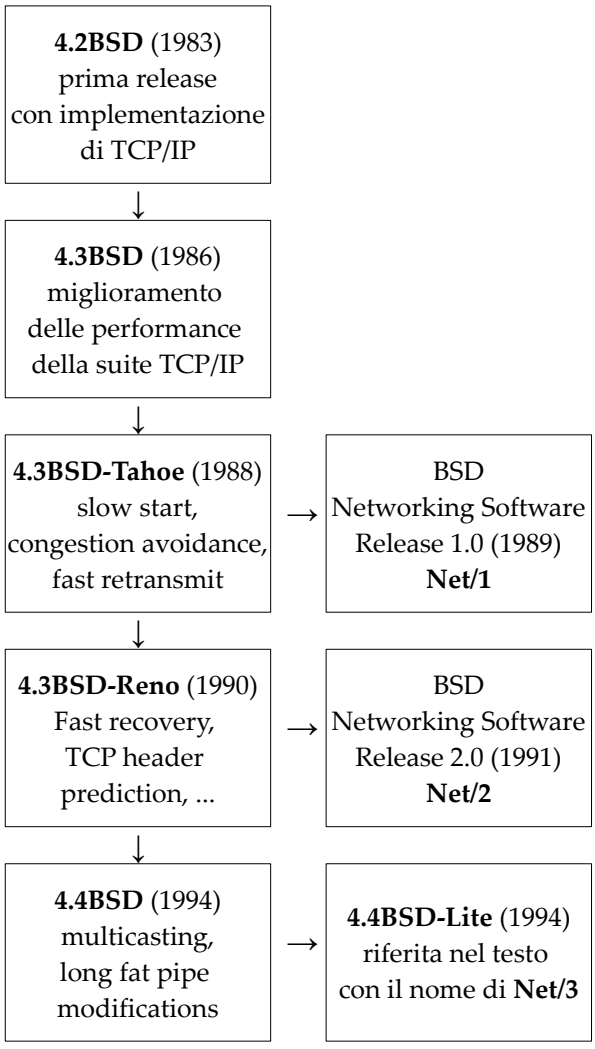


Figura 1.1
Varie release BSD con importanti feature TCP/IP

La natura permissiva della licenza BSD concesse poi a molti altri sistemi

operativi, liberi e proprietari, di incorporare codice BSD al loro interno. Per esempio, in Microsoft Windows 2000 del codice derivato da BSD è utilizzato nell'implementazione della suite di protocolli TCP/IP.
Nello sviluppo di questo testo utilizzeremo il codice di rete della 4.4BSD-Lite [9] (cui ci riferiremo con il nome di Net/3).

1.4 Implementazione del sistema di rete

Net/3 offre un'infrastruttura general-purpose capace di supportare simultaneamente svariati protocolli di comunicazione. 4.4BSD, infatti, supporta quattro diversi protocolli di rete:

- TCP/IP (suite di protocolli Internet), argomento della nostra discussione;
- XNS (Xerox Network System), una suite di protocolli simile a TCP/IP popolare negli anni '80 per interconnettere periferiche Xerox su di una rete locale;
- protocolli OSI;
- protocolli interni al dominio Unix, utilizzati per lo scambio di informazioni tra sistemi diversi attraverso l'IPC (Inter Process Communication).

Il codice di rete all'interno del kernel è organizzato su tre livelli, come mostrato in Figura 1.2:

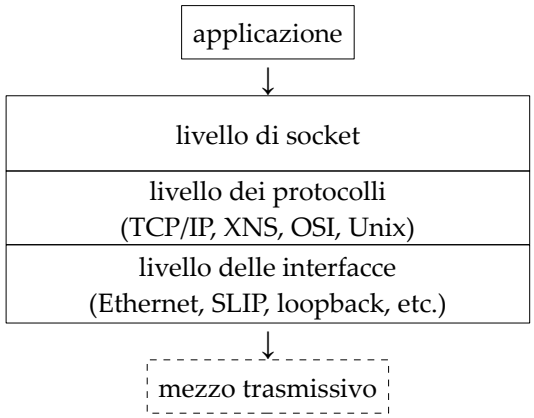


Figura 1.2
Suddivisione del codice di rete in Net/3

1. Il *livello di socket* è un'interfaccia (indipendente dal protocollo usato) per il livello inferiore (dipendente dal protocollo usato). Tutte le chiamate di sistema sono effettuate al livello socket.
2. Il *livello dei protocolli* contiene un'implementazione dei quattro protocolli di comunicazione precedentemente menzionati.
3. Il *livello delle interfacce* contiene i driver di gestione delle interfacce con cui il dispositivo si connette al mezzo trasmissivo.

Nel nostro documento, poiché interessati all'implementazione dei meccanismi

di invio/ricezione di TCP/IP, ci concentreremo sui livelli di *socket* e *protocolli*.
In particolare, dando un'occhiata all'organizzazione del codice, noteremo che la nostra trattazione sarà prevalentemente incentrata sulla directory `netinet`.

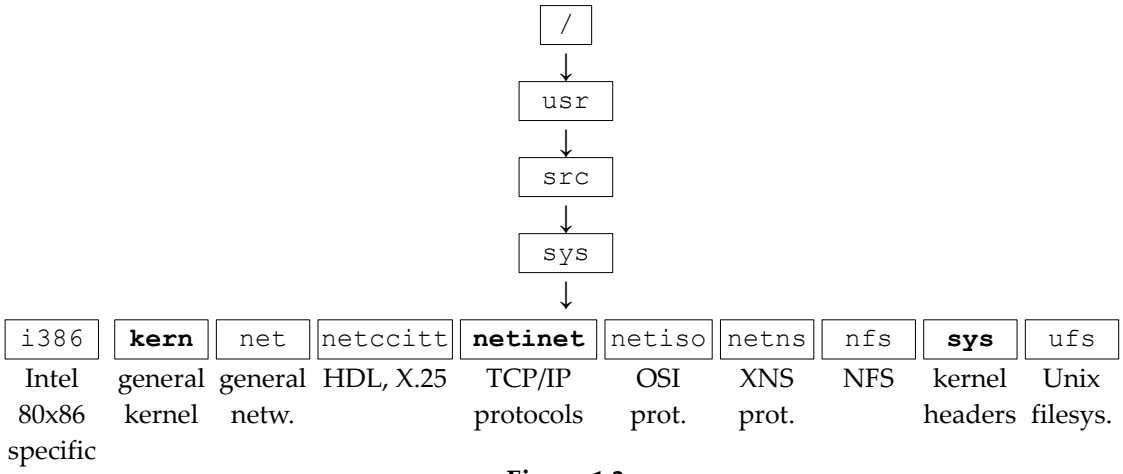


Figura 1.3
Organizzazione del codice sorgente di Net/3

1.5 Contenuto del documento

Questo documento si articola in quattro parti. Ad eccezione della prima, di carattere puramente introduttivo, le altre parti si preoccuperanno di mostrare analogie e differenze tra una presentazione astratta dei concetti (come quella in [1]) e la loro realizzazione effettiva.
Nella *Parte II* richiameremo i concetti teorici riguardanti i buffer di spedizione e ricezione e i relativi controlli di flusso e congestione (in tale parte seguiremo le convenzioni di presentazione introdotte dall'opera di Kurose e Ross).
Lo sviluppo del documento proseguirà, nella *Parte III*[2], seguendo lo schema della trattazione riportata in [2]. Nel *Capitolo 2*, infatti, ci concentreremo sulla gestione della memoria introducendo il concetto di *memory buffer*, utilizzato in tutto il sistema BSD per la memorizzazione di svariate informazioni. Con il *Capitolo 3*, invece, comincerà la trattazione riguardante l'implementazione del *networking*. In questo capitolo, infatti, andremo a parlare dell'invio e della ricezione di dati su di una connessione per mezzo di una sua astrazione (*socket*). Attraversata l'astrazione della socket, potremmo finalmente passare ad analizzare come Net/3 implementa effettivamente il Transmission Control Protocol. Tale protocollo sarà oggetto dei *Capitoli 4, 5, 6 e 7*.
Terminata la trattazione riguardante TCP, con la *Parte IV* concluderemo il nostro documento mostrando analogie e differenze presenti tra la presentazione teorica e l'effettiva implementazione di Net/3.

2 La *Parte III* consiste in una presentazione semplificata della porzione di [7] riguardante l'implementazione della suite TCP. Le convenzioni e il codice usati in questo documento, infatti, si rifanno a quelle proposte in tale testo.

Parte II – Richiami di teoria

2. Richiami di teoria

2.1 Introduzione

In questo capitolo ci occupiamo di richiamare i concetti presentati nel testo [1] di Kurose e Ross.
Inizialmente presenteremo l'astrazione dei buffer di ricezione e di spedizione e poi proseguiremo con i controlli di flusso e di congestione. Da notare che, per semplificare la presentazione degli algoritmi, nella teoria si assume di inviare sempre segmenti aventi come dimensione quella massima consentita (1 MSS).

2.2 Buffer di spedizione e ricezione

In una connessione TCP, ciascuna delle due entità comunicanti operano sia da mittente sia da destinatario. Per svolgere il proprio compito di invio e ricezione di segmenti, ciascun capo della comunicazione necessita di due porzioni di memoria. La prima porzione, chiamata *buffer di spedizione*, è adibita alla memorizzazione dei dati in attesa di essere inviati.

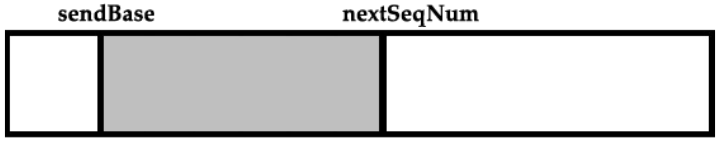


Figura 2.1
Send buffer

Notiamo come il buffer sia rappresentato come una finestra di memoria a dimensione limitata. Tale finestra è indicizzabile solamente per mezzo di *sendBase*, il più vecchio byte inviato e non ancora riscontrato, e *nextSeqNum*, il prossimo numero di sequenza utilizzabile.
La seconda porzione di memoria, chiamata *buffer di ricezione*, è invece utilizzata per la memorizzazione dei dati ricevuti dall'altra entità comunicante.

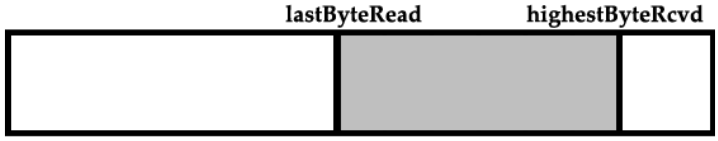


Figura 2.2
Receive buffer

Analogamente al caso della spedizione, il buffer di ricezione è composto da una finestra di memoria di dimensione limitata (non necessariamente uguale a quella del buffer di spedizione) e indicizzabile solamente per mezzo di due indici: *lastByteRead*, numero di sequenza dell'ultimo byte letto dal buffer da

parte dell'applicativo, e *highestByteRcvd*, numero di sequenza del più recente byte copiato all'interno del buffer di ricezione.
In entrambi i casi, quindi, al nostro livello di astrazione, utilizziamo i numeri di sequenza come indici della memoria: questo non ci suggerisce niente riguardo l'effettiva implementazione.

2.3 Controllo di flusso

La limitazione della dimensione del buffer di ricezione introduce un primo problema: se l'applicazione che consuma i dati presenti al suo interno non è sufficientemente rapida, può accadere che il mittente mandi in overflow il buffer di ricezione inviando dati troppo rapidamente.
TCP offre un meccanismo per scongiurare l'eventualità sopra riportata: il *controllo di flusso*. Intuitivamente, il controllo di flusso è un meccanismo di confronto sulle velocità (confronta, infatti, la frequenza di invio del mittente con quella di lettura del ricevente).
Ma come viene realmente realizzato il controllo di flusso? TCP offre tale funzionalità imponendo al mittente di mantenere una variabile chiamata *finestra di ricezione* (RcvWin in [1]) che, in sostanza, fornisce un'indicazione dello spazio disponibile nel buffer di ricezione del destinatario.
Dato che il buffer di ricezione del destinatario non può andare in overflow, deve valere:

$$highestByteRcvd - lastByteRead \leq RcvBuffer$$

dove, con *RcvBuffer*, intendiamo la dimensione del buffer di ricezione.
Il destinatario, quindi, durante la comunicazione fornisce al mittente il valore della finestra di ricezione secondo la formula:

$$RcvWin = RcvBuffer - [highestByteRcvd - lastByteRead]$$

Una volta venuto a conoscenza dello spazio disponibile all'interno del buffer del destinatario, il mittente può inviare una quantità di byte al massimo uguale a *RcvWin*.
Notiamo come, in questo modo, la finestra di ricezione potrebbe assumere il valore di 0 byte. In questo caso, il mittente si bloccherebbe e non invierebbe alcun segmento. Di conseguenza, il ricevente non avrebbe dati da riscontrare e quindi non comunicherebbe alcuna variazione della finestra di ricezione e la comunicazione finirebbe in una situazione di stallo. Per evitare questa situazione, TCP prevede che, nel caso sia comunicata una finestra di ricezione di 0 byte, il mittente entri in *zero-window probing*: il mittente provvederà a inviare segmenti contenenti un solo byte di dati in modo che il ricevente sia costretto a riscontrarli e a inviare di conseguenza un aggiornamento della finestra di ricezione.

2.4 Controllo di congestione

Il segmento TCP viene inviato al destinatario attraverso il protocollo IP. Tale protocollo non offre alcun controllo per evitare il congestionamento della rete: TCP dovrà perciò prevedere un controllo di congestione end-to-end.
Ciascun capo della comunicazione, perciò, mantiene una variabile, chiamata *CongWin*, per imporre un vincolo alla frequenza di immissione di traffico sulla rete da parte dei mittenti TCP.
Ma come viene percepita da TCP la congestione della rete? Definiamo "evento di perdita" per il mittente TCP l'occorrenza di un timeout o la ricezione di tre ACK duplicati da parte del destinatario. In presenza di una congestione eccessiva uno o più buffer dei router lungo il percorso si trovano in condizioni di overflow (ed eliminano, non potendoli memorizzare, i datagrammi in eccesso). L'eliminazione di un datagramma viene percepita dal mittente come un evento di perdita. Quest'ultimo, quindi, dovrà interpretare gli eventi di perdita come indicativi di congestionamento della rete.
Allo stesso tempo, però, la ricezione di un ACK positivo è sintomatica del fatto che la rete è libera e che quindi possiamo aumentare il numero di datagrammi in transito.
Queste due idee sono alla base dell'algoritmo di controllo della congestione [5]. Di seguito riportiamo una tabella che riassume i punti principali dell'algoritmo:

Stato	*Evento*	*Azione del controllo di congestione TCP*	*Commenti*
Slow Start (SS)	Ricezione ACK per dati precedentemente non riscontrati	CongWin = CongWin + MSS; if (CongWin ≥ Threshold) then *Stato* = CA;	CongWin raddoppia ad ogni RTT
Congestion Avoidance (CA)	Ricezione ACK per dati precedentemente non riscontrati	CongWin += MSS * MSS / CongWin;	Incremento additivo[3]
SS o CA	Ricezione del terzo ACK duplicato	Threshold = CongWin / 2; CongWin = Threshold; Stato = CA;	Ripristino rapido con il decremento moltiplicativo
SS o CA	Timeout	Threshold = CongWin / 2; CongWin = 1 MSS; stato = SS;	Entra nello stato di Slow Start

A questo punto siamo in grado di determinare il valore di CongWin in ogni momento della nostra comunicazione. Questo ci permette, ogni volta che

3 CongWin aumenta di al massimo 1 MSS ad ogni RTT

vogliamo spedire nuovi dati, di avere un limite superiore tale da ridurre i problemi relativi al congestionamento.

2.5 Combinazione dei controlli di flusso e di congestione

Abbiamo visto come sia il controllo di flusso sia quello di congestione impongano un limite superiore alla quantità di "nuovi" dati inviabili. Quando, perciò, il mittente TCP deve andare a effettuare la spedizione di dati non ancora inviati, deve combinare i due limiti. Se la finestra disponibile per la spedizione (calcolabile come il minimo tra *RcvWin* e *CongWin*) non è totalmente occupata da dati già inviati, possiamo procedere all'invio di altri dati. In formula:

$$nextSeqNum - sendBase \leq min(CongWin, RcvWin)$$

Notiamo che tale vincolo deve essere verificato solamente quando TCP invia "nuovi" dati: non si tratta, perciò, di una proprietà invariante della connessione.

Parte III – Implementazione di BSD

3. Mbufs: Memory Buffers

3.1 Introduzione

I protocolli di rete eseguono molte richieste di gestione della memoria la cui implementazione risiede nel kernel. Queste richieste includono manipolare facilmente buffer di dimensione variabile:

- aggiunta di dati al buffer (quando i livelli inferiori procedono ad incapsulare i dati passati dai livelli superiori);
- rimozione di dati dal buffer (quando vengono rimossi gli header mentre i dati viaggiano verso l'alto nello stack dei protocolli);
- minimizzazione della quantità di dati copiati per le precedenti operazioni.

In particolare, nella nostra trattazione, siamo interessati ad avere una struttura dati dinamica in grado di memorizzare una coda di segmenti (con dimensione variabile). A tal fine uno dei concetti fondamentali nella progettazione del codice di rete della Berkeley è il *memory buffer*, chiamato `mbuf`, usato all'interno del sorgente per gestire svariate porzioni di informazione.

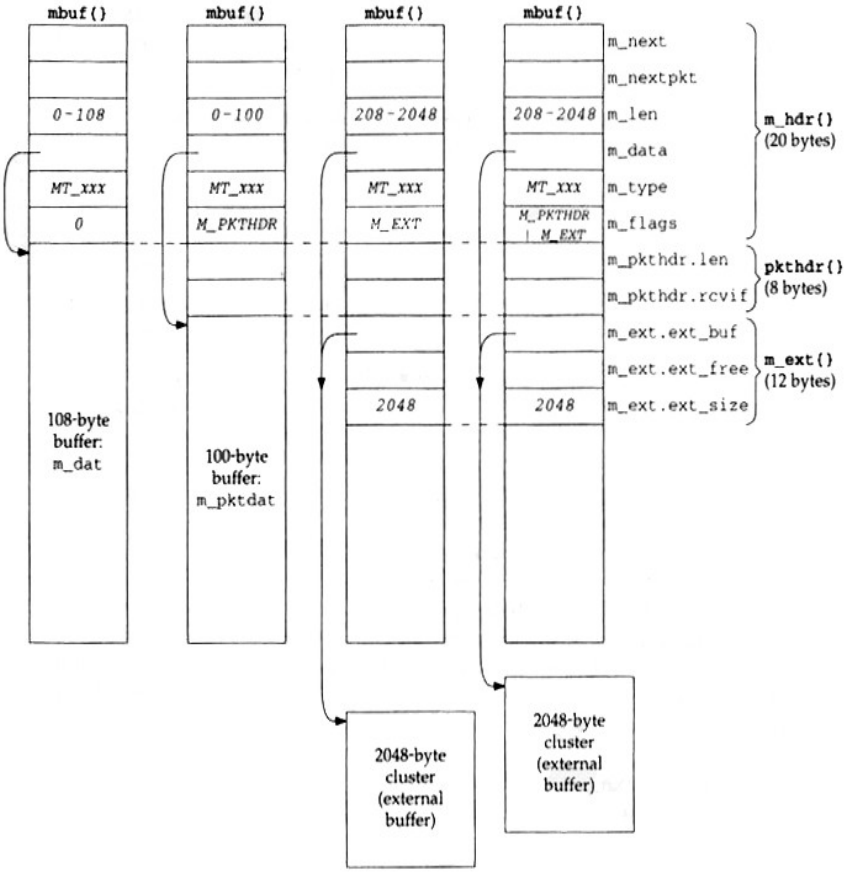


Figura 3.1
Quattro differenti tipi di mbufs, dipendenti dal valore di `m_flags`

In Figura 2.1, da sinistra a destra, sono presentati i quattro possibili utilizzi degli mbuf:

- Se `m_flags` è uguale a 0, l'mbuf contiene solamente dati. All'interno del mbuf c'è spazio per un massimo di 108 byte di dati (array `m_dat`). Il puntatore `m_data` riferisce a qualche posizione interna al buffer (in Figura è posizionato all'inizio ma può riferire uno qualsiasi dei 108 byte). Il campo `m_len` specifica la lunghezza, in byte, dei dati (a partire da `m_data`).
- E se dovessimo memorizzare più di 108 byte? Potremmo concatenare più mbuf in modo da costituire una catena dinamica. Tale catena necessiterà di un inizio: il secondo tipo di mbuf ha `m_flags` settato a `M_PKTHDR` per specificare che si tratta di un *packet header*, ovvero del primo mbuf che descrive un pacchetto di dati. Il puntatore `m_next` collega, quindi, vari mbuf per formare un singolo pacchetto all'interno di una *catena di mbuf*.
  Anche in questo caso i dati sono contenuti all'interno del mbuf, ma poiché 8 byte sono utilizzati per l'header di pacchetto, solamente 100 byte di dati possono essere memorizzati al suo interno (array `m_pktdat`).
  Il valore di `m_pkthdr.len` rappresenta la lunghezza totale di tutti i dati nella catena di mbuf per questo pacchetto (la somma di tutti gli `m_len` incontrati attraversando i puntatori `m_next`).
- La tipologia successiva di mbuf non contiene un header di pacchetto (`M_PKTHDR` non è settato) ma contiene più di 208[4] byte di dati. A tal fine è disposto un buffer esterno, chiamato *cluster* (`M_EXT` settato)[5]. Invece di utilizzare mbuf multipli per contenere i dati (il primo con 100 byte e i restanti con 108 byte ciascuno), Net/3 alloca un cluster di 1024 o 2048 byte[6]. Il puntatore `m_data` presente in mbuf riferisce una qualunque posizione all'interno del cluster. Gli mbuf con cluster contengono sempre l'indirizzo iniziale del buffer (`m_ext.ext_buf`) e la sua dimensione (`m_ext.ext_size`).
- Il quarto ed ultimo mbuf rappresentabile contiene un packet header (e quindi è il primo di una catena di mbufs) e, contemporaneamente, contiene più di 208 byte di dati. Sia `M_PKTHDR` sia `M_EXT` sono settati.

Abbiamo visto di poter memorizzare pacchetti di dati. Facendo puntare `m_nextpkt` ad un altro packet header, possiamo organizzare i pacchetti in una *coda di mbuf*. Ogni pacchetto nella coda può essere un mbuf singolo o una catena di mbuf. Condizione imprescindibile, però, è che ogni pacchetto contenga il

4 Net/3 considera 208 come la quantità minima di dati memorizzabili all'interno di un mbuf clusterizzato.

5 Lo spazio per l'header viene comunque riservato anche se non viene utilizzato.

6 Nel proseguimento della nostra trattazione assumeremo che gli mbuf abbiamo sempre la dimensione di 2048 byte.

relativo packet header. Se più mbuf definiscono un pacchetto, il puntatore `m_nextpkt` del primo di questi è l'unico utilizzato (i puntatori `m_nextpkt` presenti negli mbuf rimanenti nella catena saranno tutti riferimenti a `NULL`).

3.2 File di riferimento

Il codice riguardante mbuf è interamente contenuto in un file C e in un header:

File	Descrizione
`sys/mbuf.h`	Struttura `mbuf`, macro e definizioni
`kern/uipc_mbuf.c`	Funzioni relative a mbuf

3.3 Definizioni relative a mbuf

Ci sono alcune costanti che incontreremo ripetutamente mentre lavoriamo con gli mbuf. I loro valori sono esplicati nella tabella seguente:

Costante	Valore	Descrizione
MCLBYTES	2048	Dimensione di un cluster esterno
MHLEN	100	Quantità massima di dati per un mbuf con header di pacchetto
MINCLSIZE	208	Quantità minima di dati da inserire in un cluster
MLEN	108	Quantità massima di dati contenibile in un mbuf classico
MSIZE	128	Dimensione di ciascun mbuf

3.4 Struttura `mbuf`

```
----------------------------------------------------------------------mbuf.h
 60 /*header at beginning of each mbuf*/
 61 struct m_hdr {
 62    struct mbuf *mh_next;          /*next buffer in chain*/
 63    struct mbuf *mh_nextpkt;       /*next chain in queue*/
 64    int      mh_len;               /*amount of data in this mbuf*/
 65    caddr_t  mh_data;              /*pointer to data*/
 66    short mh_type;                 /*type of data*/
 67    short    mh_flag;              /*flags*/
 68 };

 69 /*packet header in first mbuf of chain; valid if M_PKTHDR set*/
 70 struct pkthdr {
 71    int      len;                  /*total packet length*/
 72    struct ifnet rcvif;            /*receive interface*/
 73 };

 74 /*external storage mapped into mbuf; valid if M_EXT set*/
 75 struct m_ext {
 76    caddr_t  ext_buf;              /*start of buffer*/
 77    void     (*ext_free)();        /*free routine if not the
                                        usual*/
 78    u_int ext_size;                /*size of buffer, for ext_free*/
 79 };

 80 struct mbuf {
 81    struct m_hdr m_hdr;
 82    union {
 83       struct {
```

```
 84          struct pkthdr MH_pkthdr;  /*M_PKTHDR set*/
 87          union {
 86             struct m_ext MH_ext;   /*M_EXT set*/
 87             char MH_databuf[MHLEN];
 88          } MH_dat;
 89       } MH;
 90       char  M_databuf[MLEN];       /*!M_PKTHDR, !M_EXT*/
 91    } M_dat;
 92 };

 93 #define  m_next      m_hdr.mh_next
 94 #define  m_len       m_hdr.mh_len
 95 #define  m_data      m_hdr.mh_data
 96 #define  m_type      m_hdr.mh_type
 97 #define  m_flags     m_hdr.mh_flags
 98 #define  m_nextpkt   m_hdr.mh_nextpkt
 99 #define  m_act       m_nextpkt
100 #define  m_pkthdr    M_dat.MH.MH_pkthdr
101 #define  m_ext       M_dat.MH.MH_dat.MH_ext
102 #define  m_pktdat    M_dat.MH.MH_dat.MH_databuf
103 #define  m_dat       M_dat.M_databuf
-------------------------------------------------------------------mbuf.h
```

La struttura `mbuf` è definita come una struttura `m_hdr`, seguita da una `union`. Come indicato dai commenti, i contenuti della `union` dipendono dai valori dei flags `M_PKTHDR` e `M_EXT` (righe 80 – 92).
Le undici `#define` (righe 93-103) semplificano l'accesso ai membri delle strutture e delle `union` interni alla struttura `mbuf`. Questa tecnica è stata utilizzata nello sviluppo del codice sorgente di Net/3 ogniqualvolta sia stata definita una struttura contenente altre strutture o unioni.

3.5 Funzioni e macro relative a mbuf

Le funzioni e le macro usate per gestire mbuf sono più di venti. Allo scopo della nostra trattazione risulta interessante analizzare solamente quelle riguardanti l'allocazione dello spazio di memoria riguardante mbuf.
Partiamo con `m_get`, la funzione da invocare ogni volta che si voglia creare un mbuf:

```
-------------------------------------------------------------uipc_mbuf.c
134 struct mbuf *
135 m_get(nowait,type)
136 int nowait, type;
137 {
138    struct mbuf *m;
139    MGET(m, nowait, type);
140    return (m);
141 }
-------------------------------------------------------------uipc_mbuf.c
```

`m_get`, dunque, si limita solamente ad espandere la macro `MGET`:

```
-------------------------------------------------------------------mbuf.h
154 #define  MGET(m, how,type) { \
155          MALLOC((m),   struct   mbuf   *,   MSIZE,   mbtypes[type],
(how)); \
156          if (m) { \
```

```
157             (m)->m_type = (type); \
159             (m)->m_next = (struct mbuf *) NULL; \
160             (m)->m_nextpkt = (struct mbuf *) NULL; \
161             (m)->m_data = (m)->m_dat; \
162             (m)->m_flags = 0; \
163          } else \
164             (m) = m_retry((how),(type)); \
165 }
-------------------------------------------------------------------mbuf.h
```

`MGET` inizialmente (riga 155) invoca la macro `MALLOC`, che costituisce un allocatore di memoria generico per il kernel. Se la memoria può essere allocata, allora procede ad inizializzare tutti i valori degli argomenti di mbuf (righe 156 – 162). Se la chiamata all'allocatore di memoria del kernel fallisce, `m_retry` viene invocata con lo scopo di tentare nuovamente:

```
-----------------------------------------------------------uipc_mbuf.c
 92 struct mbuf *
 93 m_retry(i, t)
 94 int i, t;
 95 {
 96    struct m_buf *m;
 97    m_reclaim();
 98 #define  m_retry(i, t)  (struct mbuf *)0;
 99    MGET(m, i, t);
100 #undef   m_retry
101    return (m);
102 }
-----------------------------------------------------------uipc_mbuf.c
```

Appena c'è la possibilità che ci sia più memoria disponibile (dopo l'invocazione di `m_reclaim` – riga 97), la macro `MGET` viene nuovamente invocata per provare a ottenere un mbuf. Prima di espandere `MGET`, viene comunque ridefinita `m_retry` in modo da evitare un loop infinito dovuto alla mancanza di memoria disponibile (righe 98 – 99). Al termine dell'espansione di `MGET`, procediamo a togliere la definizione di `m_retry` affinché non influisca su altre eventuali invocazioni di `MGET` all'interno del codice (riga 100).
Per quanto riguarda le restanti macro e funzioni definite al riguardo di mbuf riportiamo due tabelle riassuntive[7]:

Macro	Descrizione
`MCLGET`	Riserva un cluster esterno e setta il puntatore ai dati del buffer corrente al buffer riservato `void` **`MCLGET`**`(struct mbuf *m, int nowait)`
`MFREE`	Libera la memoria riservata al mbuf puntato da `m` e memorizza il suo successore (`m->m_next`) in `n`. `void` **`MFREE`**`(struct mbuf *m, struct mbuf *n)`
`MGETHDR`	Alloca un mbuf e lo inizializza in modo da renderlo un packet header. `void` **`MGETHDR`**`(struct mbuf *m, int nowait, int type)`

7 Per quanto riguarda le macro utilizzeremo una notazione in stile "prototipo di funzione C".

Macro	Descrizione
`MH_ALIGN`	Setta il puntatore ai dati del mbuf in modo da riservare spazio per un oggetto di dimensione len in fondo all'area dati `void` **`MH_ALIGN`**`(struct mbuf *m, int len)`
`M_PREPEND`	Aggiorna, se possibile, il puntatore (altrimenti alloca un nuovo mbuf) in modo da riservare spazio per `len` bytes in cima all'area dati `void` **`MCLGET`**`(struct mbuf *m, int len, int nowait)`
`dtom`	Converte il puntatore `x`, che deve puntare qualche posizione interna alla area dati del buffer, in un puntatore all'inizio del mbuf `struct mbuf*` **`dtom`**`(void *x)`
`mtod`	Esegue il cast a *`type`* del puntatore all'area dati del mbuf (`m`) *`type`* **`mtod`**`(struct mbuf *m,` *`type`*`)`

Funzione	Descrizione
`m_adj`	Rimuove `len` bytes di dati dal mbuf riferito dal puntatore passato come parametro `void` **`m_adj`**`(struct mbuf *m,int len)`
`m_cat`	Concatena gli mbuf puntati da `m` e da `n` `void` **`m_cat`**`(struct mbuf *m,struct mbuf *n)`
`m_copy`	Versione a tre parametri della funzione `m_copym` che rende implicito il quarto parametro (al valore `M_DONTWAIT`) `struct mbuf *`**`m_copy`**`(struct mbuf *m,int offset,int len)`
`m_copydata`	Copia `len` bytes dalla catena di mbuf puntata da `m` nel buffer puntato da `p`. La copia comincia a partire dal byte `offst` specificato. `void` **`m_copydata`**`(struct mbuf *m,int offst,int len,caddr_t p)`
`m_copyback`	Esegue la stessa operazione di `m_copydata` invertendo sorgente e destinazione dei dati `void` **`m_copyback`**`(struct mbuf *m,int offst,int len,caddr_t p)`
`m_copym`	Crea una nuova catena di mbuf e vi copia `len` bytes di `m` a partire dal byte di `offst` `void` **`m_copym`**`(struct mbuf *m,int offst,int len,int nowait)`
`m_devget`	Crea una nuova catena di mbuf e ritorna il puntatore al packet header `struct mbuf *`**`m_devget`**`(char *buf,int len,int off, struct inet *ifp,void (*copy)(const void *,void*,u_int))`
`m_free`	Funzione che incapsula la macro `MFREE` `struct mbuf *`**`m_free`**`(struct mbuf *m)`
`m_freem`	Libera tutti gli mbuf nella catena puntata da `m` `void` **`m_freem`**`(struct mbuf *m)`
`m_getclr`	La funzione invoca la macro `MGET` per ottenere un mbuf e poi azzera la parte dati `struct mbuf *`**`m_getclr`**`(int nowait,int type)`

Funzione	Descrizione
`m_gethdr`	Funzione che incapsula la macro `MGETHDR` `struct mbuf *`**`m_gethdr`**`(int nowait,int type)`
`m_pullup`	Risistema i dati esistenti in modo da far sì che i primi `len` bytes presenti nel buffer siano memorizzati nel primo mbuf della catena `void` **`m_pullup`**`(struct mbuf *m,int len)`

4. Socket I/O

4.1 Introduzione

Questo capitolo tratta il codice di livello socket in Net/3. L'astrazione socket venne introdotta nella release 4.2BSD del 1983 per fornire un'interfaccia uniforme ai protocolli di comunicazione di rete.
Come discusso nella sezione 1.4, il livello di socket mappa le richieste dei processi (indipendenti dal protocollo di comunicazione utilizzato) sull'implementazione specifica del protocollo selezionata al momento della creazione della socket.
Per permettere alle system call tipiche dello Unix I/O (come `read` e `write`) di operare con le connessioni di rete, il filesystem e le infrastrutture di rete nelle release di BSD sono state integrate al livello delle system call. L'accesso alle connessioni di rete, rappresentate dai socket, avviene attraverso il descrittore così come avviene per l'accesso ai file. Questo permette alle *filesystem call* standard di lavorare con il descrittore associato alla socket.

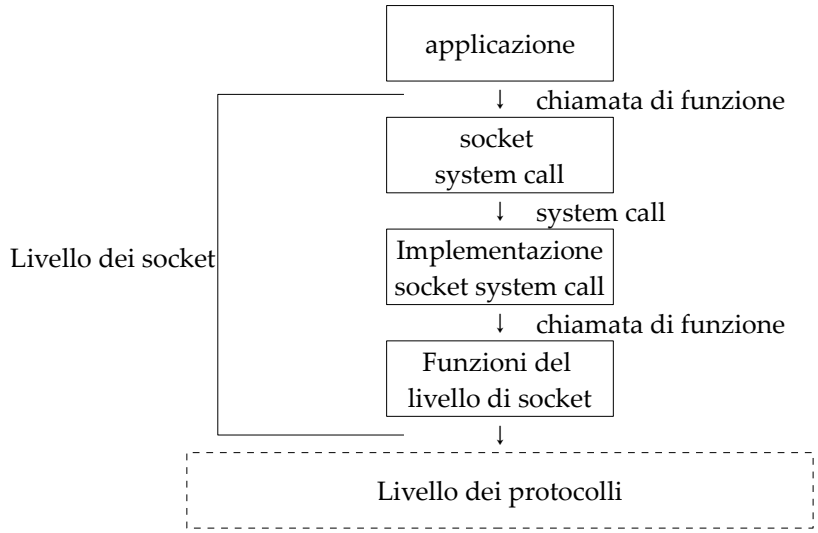


Figura 4.1
Il livello di socket converte richieste generiche in operazioni specifiche del protocollo di comunicazione

Questo documento si concentrerà sull'implementazione delle socket (in particolare dei buffer di invio e ricezione) e delle system call di lettura e scrittura.

4.2 File di riferimento

La trattazione affrontata in questo capitolo riguarda i seguenti file:

File	Descrizione
`sys/socket.h`	Strutture e macro per la socket API
`sys/socketvar.h`	Struttura `socket` e macro
`sys/uio.h`	Definizione della struttura `uio`
`kern/uipc_syscalls.c`	system call su `socket`
`kern/uipc_socket.c`	Processing del livello di socket

All'interno dei quali è definita la seguente variabile globale:

Variabile	Tipo di dato	Descrizione
`sb_max`	`u_long`	Dimensione massima (in byte) per i buffer di invio/ricezione

4.3 Socket Buffer

Una socket rappresenta uno dei capi della comunicazione e mantiene tutte le informazioni associate ad essa. Queste informazioni includono il protocollo da usare, lo stato del protocollo (tra cui figurano gli indirizzi di sorgente e destinazione), ed altre svariate informazioni. In particolare, ai fini della nostra trattazione, siamo interessati ad analizzare i buffer di invio (`so_snd`) e ricezione (`so_rcv`).

```
------------------------------------------------------------socketvar.h
  41 struct socket {

  69 /*
  70  * Variables for socket buffering
  71  */
  72    struct sockbuf {
  73       u_long   sb_cc;              /*actual chars in buffer*/
  74       u_long   sb_hiwat;           /*max actual octet char count*/
  75       u_long   sb_mbcnt;           /*chars of mbuf used*/
  76       u_long   sb_mbmax;           /*max chars of mbufs to use*/
  77       long     sb_lowat;           /*low water mark*/
  78       struct mbuf sb_mb;           /*the mbuf chain*/
  79       struct selinfo sb_sel;       /*process selecting read/write*/
  80       short sb_flags;              /*flags*/
  81       short sb_timeo;              /*timeout for read/write*/
  82    } so_rcv, so_snd;
  83    caddr_t  so_tpcb;               /*Wisc, protocol control block
                                         *XXX*/

  86 };
------------------------------------------------------------socketvar.h
```

Ciascun buffer contiene informazioni di controllo e puntatori ai dati memorizzati in catene di mbuf (righe 72 – 78):

- `sb_mb` punta al primo degli mbuf nella catena;
- `sb_cc` è il numero totale di byte di dati contenuti all'interno degli mbufs;
- `sb_hiwat` (*high water*) e `sb_lowat` (*low water*) sono utilizzati nell'algoritmo di controllo di flusso della socket;
- `sb_mbcnt` è il totale della memoria allocata per gli mbuf nel buffer.

Ricordiamo che ogni mbuf può memorizzare da 0 a 2048 byte di dati (se viene utilizzato un cluster esterno) e che `sb_mbmax` rappresenta un limite superiore

per la quantità di memoria allocabile per ogni socket buffer. I limiti di default sono specificati da ogni protocollo quando viene eseguita la system call `socket`. Nella nostra trattazione gli algoritmi di buffering saranno descritti successivamente, quando parleremo delle funzioni di invio e ricezione.
Gli altri campi (righe 79 – 81) presenti nella struttura `sockbuf` non risultano interessanti ai fini di questo documento[8].

4.4 Macro e funzioni per la gestione dei socket buffer

Ci sono molte macro e funzioni che manipolano i buffer di spedizione e ricezione associati ad ogni socket. Quelle riportate nella tabella seguente sono quelle utilizzate per "bloccare" i buffer ed effettuare la sincronizzazione:

Nome	Descrizione
`sblock`	Acquisice la *lock* per `sb`. In base a `wf`, il processo decide se "dormire" in attesa del buffer o comportarsi diversamente `int` **`sblock`**`(struct sockbuf *sb,int wf)`
`sbunlock`	Rilascia la *lock* per `sb`. Se ci sono processi in attesa della lock, vengono risvegliati `void` **`sbunlock`**`(struct sockbuf *sb)`

mentre di seguito esponiamo quelle per l'allocazione e la manipolazione del buffer:

Nome	Descrizione

`sbspace`	Numero di bytes aggiungibili a `sb` prima che venga considerato pieno
	`min((sb_hiwat - sb_cc),(sb_max - sb_mbcnt))`
	`long` **`sblock`**`(struct sockbuf *sb)`

`sballoc`	`m` viene aggiunto a `sb`. Aggiorna `sb_cc` e `sb_mbcnt` di conseguenza. `void` **`sballoc`**`(struct sockbuf *sb, struct mbuf *m)`
`sbfree`	`m` viene rimosso da `sb`. Aggiorna `sb_cc` e `sb_mbcnt` di conseguenza. `int` **`sbfree`**`(struct sockbuf *sb,struct mbuf *m)`
`sbappend`	Esegue la append di `m` su `sb`. (chiama `sbcompress`) `int` **`sbappend`**`(struct sockbuf *sb,struct mbuf *m)`
`sbappendaddr`	Inserisce l'indirizzo `asa` in un mbuf. Concatena indirizzo, `control`, e `m0` ed esegue la append del risultato in `sb`. `int` **`sbappendaddr`**`(struct sockbuf *sb,struct sockaddr *asa, struct mbuf *m0,struct mbuf *control)`
`sbinsertoob`	Inserisce `m0` prima del primo record in `sb` senza dati out-of-band. (chiama `sbcompress`) `int` **`sbinsertoob`**`(struct sockbuf *sb,struct mbuf *m0)`

8 `sb_sel` è utilizzata nell'implementazione della system call `select`; `sb_flags` è utilizzata come variabile di controllo per selezionare il comportamento voluto; `sb_timeo`, misurata in tick del clock, limita il tempo di attesa del processo durante una system call di `read`/`write`.

Nome	Descrizione
`sbcompress`	Esegue la append di `m` su `n` comprimendo lo spazio inutilizzato. `void` **`sbcompress`**`(struct sockbuf *sb,struct mbuf *m, struct mbuf *n)`
`sbdrop`	Scarta i primi `len` bytes in sb. `void` **`sbdrop`**`(struct sockbuf *sb,int len)`
`sbdroprecord`	Scarta il primo record in `sb`. `void` **`sbdroprecord`**`(struct sockbuf *sb)`
`sbrelease`	Invoca la `sbflush` per rilasciare tutti i buffer in `sb`. Resetta i valori di `sb_hiwat` e `sb_mbmax` a 0. `void` **`sbrelease`**`(struct sockbuf *sb)`
`sbflush`	Rilascia tutti i buffer in `sb`. `void` **`sbflush`**`(struct sockbuf *sb)`
`soreserve`	Setta *high-water* e *low-water*. Per il send buffer invoca la `sbreserve` con `sndcc`. Per il receive buffer invoca la `sbreserve` con `rcvcc`. `int` **`soreserve`**`(struct socket *so,int sndcc,int rcvcc)`
`sbreserve`	Setta i limiti di high-water e low-water in relazione a `cc`. (In questa funzione non viene allocata memoria) `int` **`sbreserve`**`(struct socket *sb,int cc)`

4.5 System call di scrittura

`write`, `writev`, `sendto` e `sendmsg`, cui ci riferiamo collettivamente come system call di scrittura, sono utilizzate per inviare dati su di una connessione di rete.
Tutte le system call di scrittura, direttamente o indirettamente, invocano `sosend`, che svolge il compito di copiare i dati dal processo al kernel e di passarli al protocollo associato alla socket.

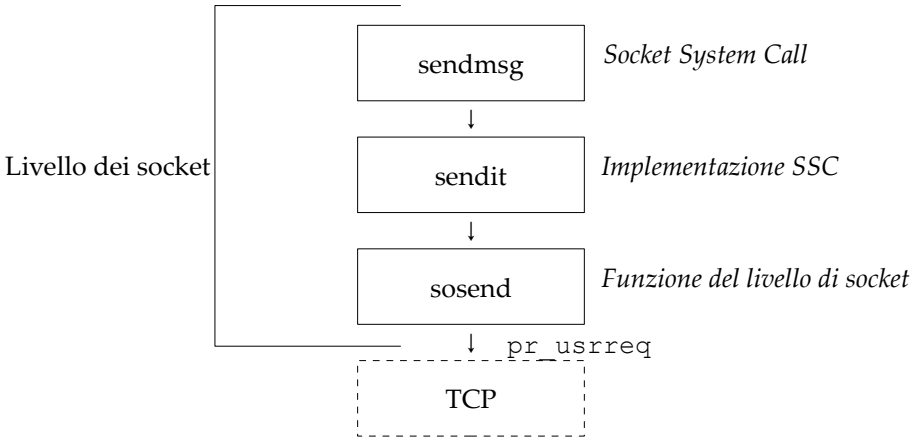


Figura 4.2
Passaggio delle informazioni da sendmsg al protocollo TCP

Nell'esempio riportato, `sendmsg` e `sendit` preparano le strutture dati[9] necessarie a `sosend`.

4.6 Funzione `sosend`

`sosend` è una delle funzioni più complicate del livello di socket. Il suo compito è quello di passare i dati e le informazioni di controllo alla funzione `pr_usrreq` relativa al protocollo associato alla socket (rispettando i limiti di buffer in esso specificati). Ai fini della nostra trattazione è importante notare che `sosend` non inserisce niente all'interno del send buffer; immagazzinare e rimuovere i dati è, infatti, compito dell'implementazione del protocollo.
Con l'utilizzo di `sosend` viene introdotto un primo meccanismo di controllo di flusso. Per protocolli affidabili (come TCP), il buffer di invio mantiene sia i dati non ancora trasmessi, sia quelli già trasmessi ma non ancora riscontrati. `sb_cc` è la quantità, in byte, di dati che risiedono all'interno del send buffer. La responsabilità di `sosend` è quella di assicurare che ci sia spazio sufficiente nel send buffer prima di passare qualsiasi dato al livello dei protocolli. Prima di invocare `pr_usrreq` deve[10], infatti, valere:

$$0 \leq \texttt{sb\_cc} \leq \texttt{sb\_hiwat}$$

Introduciamo lo schema generico del codice della funzione `sosend`:

```
-------------------------------------------------------------uipc_socket.c
271 sosend(so, addr, uio, top, control, flags)
272 struct socket *so;
273 struct mbuf *addr;
274 struct uio *uio;
275 struct mbuf *top;
276 struct mbuf *control;
277 int flags;
278 {

            /*inizializzazione delle variabili*/

305  restart:
306    if(error = sblock(&so->so_snd,SBLOCKWAIT(flags)))
307       goto out;
308    do {                          /*main loop, until resid == 0*/

            /*attesa spazio sufficiente nel send buffer*/

            /*riempimento di un singolo mbuf o di una catena di
              mbuf*/

            /*passaggio della catena di mbuf al protocollo*/

413    } while (resid);
414  release:
415    sbunlock(&so->so_snd);
```

9 In particolare `struct io {}`. Si rimanda al lettore una trattazione più approfondita dell'argomento in quanto esula dagli obiettivi di questo documento.

10 Il valore di `sb_cc` può eccedere temporaneamente quello di `sb_hiwat` uando vengono inviati dati out-of-band.

```
            /*cleanup*/

 422 }
-------------------------------------------------------uipc_socket.c
```

Osserviamo che, per evitare problemi di race condition tra processi che utilizzano la stessa socket, vengono utilizzati i meccanismi di *lock* e *unlock* riguardanti il buffer di spedizione (righe 306 e 415). Inoltre, riguardo alla nostra trattazione, le fasi più interessanti sono quella di *attesa dello spazio*, in cui viene implementato il controllo di flusso della socket precedentemente accennato, e quella di *passaggio delle informazioni al protocollo TCP*.
Cominciamo con la parte di controllo. Per mezzo della funzione `sbspace` viene calcolata la quantità di spazio rimanente nel send buffer (quello trovato è un limite amministrativo basato sul *high water* del send buffer e sul valore di `sb_mbmax`).
Se c'è abbastanza spazio all'interno del send buffer, i dati vengono processati e infine possono essere rimandati all'elaborazione da parte del protocollo relativo alla nostra socket[11](righe 400 – 402):

```
-------------------------------------------------------uipc_socket.c
 400      error = (*so->so_proto->pr->usrreq) (so,
 401                  (flags & MSG_OOB) ? PRU_SENDOOB : PRU_SEND,
 402                                    top, addr, control);
-------------------------------------------------------uipc_socket.c
```

3.7 System call di lettura

Quando parliamo di system call di lettura (`read`, `readv`, `recvfrom` e `recvmsg`) ci riferiamo a system call in grado di ricevere dati da una connessione di rete. Così come accade per le system call di scrittura, anche quelle di lettura invocano, direttamente o no, una funzione comune: `soreceive`.
Analogamente al caso della spedizione di dati, la chiamata di `rcvmsg` avvia una catena di invocazioni, nella quale `recvmsg` e `rcvit` si occupano di preparare le strutture dati necessarie a `soreceive`.

11 Il controllo, interno all'invocazione della funzione (riga 400), serve a verificare se i dati inviati sono stati marcati dal processo come out-of-band oppure no. Nel caso lo siano, eseguiamo una richiesta di tipo `PRU_SENDOOB`, altrimenti si tratta di una semplice richiesta `PRU_SEND`.

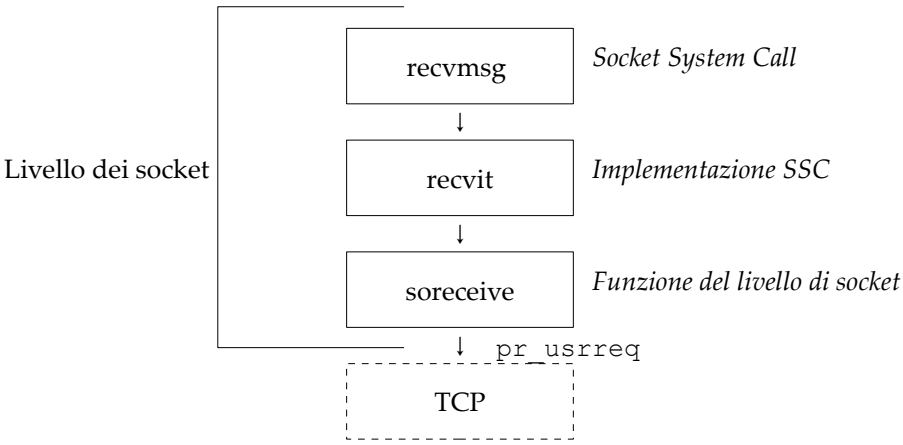


Figura 4.3
Esecuzione della system call recvmsg

3.8 Funzione `soreceive`

La funzione in questione si occupa di trasferire dati dal receive buffer della socket nel buffer specificato dal processo.
Prima di poter trattare la funzione dobbiamo però dare un sguardo all'organizzazione del buffer di ricezione. Per protocolli che non mantengono i limiti dei messaggi (p.e. i protocolli di tipo `SOCK_STREAM` come TCP), i dati in arrivo vengono accodati dopo l'ultima catena di mbuf con `sbappend`.

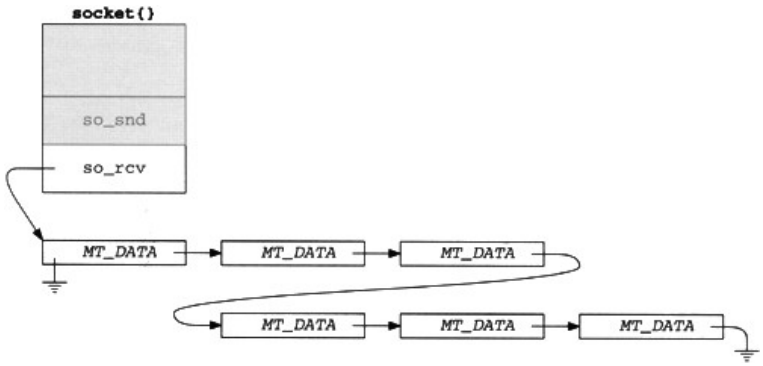


Figura 4.4
`so_rcv` buffer in TCP

Ci basta dunque sapere che la nostra funzione si limita a consumare le informazioni presenti nel buffer e a spostarle nel buffer specificato dal processo[12].

```
------------------------------------------------------------uipc_socket.c
 439 soreceive(so, paddr, uio, mp0, controlp, flagsp)
 440 struct socket *so;
 441 struct mbuf **paddr;
 442 struct uio *uio;
```

12 Utilizzerà la funzione

```
int uiomove(caddr_t cp, int n, struct uio *uio)
```

che *sposta* `n` byte dal buffer riferito da `cp` in quello riferito da `uio`.

```
443 struct mbuf **mp0;
444 struct mbuf **controlp;
445 int *flagsp;
446 {
447    struct mbuf *m, **mp;
448    int   flags, len, error, s, offset;
449    struct protosw *pr = so->so_proto;
450    struct mbuf *nextrecord;

            /*inizializzazione, MSG_OOB processing*/

            /*conferma implicita della connessione*/

484    if (error = sblock(&so->so_rcv,SBLOCKWAIT(flags)))
485       return error;
486    s = splnet();
487    m = so->so_rcv.sb_mb;

            /*attesa (eventuale) ricezione dati*/

545    nextrecord = m->m_nextpkt;

            /*elaborazione indirizzi,informazioni di controllo,etc*/

600    while(m && uio->uio_resid > 0 && error == 0) {

            /*copia e rimozione del primo mbuf della coda*/

            /*aggiornamento error e uio->uio_resid*/

690         if(m = so->so_rcv.sb_mb)
691            nextrecord = m->nextpkt;

693    }               /*while more data and more space to fill*/

            /*cleanup*/

703         (*pr->pr_usrreq) (so, PRU_RCVD, (struct mbuf *) 0,
                        (struct mbuf *) flags, (struct mbuf *) 0,
                        (struct mbuf *) 0);
            /*cleanup (continue)*/

716    sbunlock(&so->so_rcv);
717    splx(s);
718    return(error);
719 }
-------------------------------------------------------------uipc_socket.c
```

Analogamente al caso della spedizione, durante l'elaborazione del socket buffer vengono evitati problemi di race condition tramite i meccanismi di *lock* e *unlock*[13] (righe 484 e 716). Notiamo che il ciclo di consumazione del buffer di ricezione continua fino a che non si verifica una delle tre seguenti condizioni:

- il buffer di ricezione si svuota (`m == NULL`);
- la quantità di dati richiesta dal processo (`uio->uio_resid`) è stata

13 Inoltre, per evitare di essere interrotto mentre esamina i dati, l'elaborazione del protocollo è sospesa

raggiunta;
- si verifica un errore.

Al termine del ciclo, nell'eventualità che la richiesta sia stata soddisfatta, sappiamo che il buffer di ricezione ha variato la propria dimensione (in termini di byte di dati contenuti). `soreceive` prevede, perciò, di comunicare al protocollo sottostante una richiesta di tipo `PRU_RCVD`. In TCP, questa feature viene sfruttata per aggiornare il valore della finestra di ricezione relativa alla connessione.

5. TCP: Transmission Control Protocol

5.1 Introduzione

Il Transmission Control Protocol (TCP) fornisce un servizio affidabile per la trasmissione di stream di byte tra due *end-point* di un'applicazione.
Le sezioni di questo capitolo non sono, però, un'introduzione a TCP [3]. Siamo interessati a mostrare le strutture dati principali utilizzate da TCP e il metodo con cui vengono servite le richieste `pr_usrreq`.

5.2 Introduzione al codice

Il codice riguardante la porzione dell'implementazione di TCP trattata in tutto il documento appare nei seguenti file:

File	Descrizione
`netinet/tcp.h`	Definizione della struttura `tcphdr`
`netinet/tcp_seq.h`	Macro per il confronto dei numeri di sequenza TCP
`netinet/tcp_timer.h`	Definizioni riguardanti i timer di TCP
`netinet/tcp_var.h`	Struttura `tcpcb` (control block)
`netinet/tcp_input.c`	Funzione `tcp_input`
`netinet/tcp_output.c`	Funzione `tcp_output`
`netinet/tcp_usrreq.c`	Gestione richieste di tipo `PRU_xxx`

All'interno dei file presentati sono definite le seguenti variabili globali:

Variabile	Tipo dato	Descrizione
`tcp_recvspace`	`u_long`	Dimensione di default per il socket buffer di ricezione
`tcp_sendspace`	`u_long`	Dimensione di default per il socket buffer di spedizione
`tcp_iss`	`tcp_seq`	Initial Send Sequence number (ISS)
`tcprexmtthresh`	`int`	Numero di ACK duplicati da attendere per Fast Retransmit (3)
`tcp_mssdflt`	`int`	MSS di default

5.3 Intestazione di un segmento TCP

L'*header TCP* è definito con la struttura `tcphdr` (nel codice sottostante sono state riportate solamente le informazioni interessanti ai nostri scopi):

```
-------------------------------------------------------------tcp.h
  40 struct tcphdr {
  41    u_short  th_sport;              /*source port*/
  42    u_short  th_dport;              /*destination port*/
  43    tcp_seq  th_seq;                /*sequence number*/
  44    tcp_seq  th_ack;                /*acknowledgement number*/

  54    u_short  th_win;                /*advertised window*/

  57 };
-------------------------------------------------------------tcp.h
```

Net/3 prevede, poi, di inglobare l'header TCP all'interno di una struttura dati più complessa contenente anche le informazioni riguardanti IP:

```
------------------------------------------------------------tcpip.h
  38 struct tcpiphdr {
  39    struct ipovly ti_i;            /*overlaid ip structure*/
  40    struct tcphdr ti_t;            /*tcp header*/
  41 };

  49 #define  ti_sport ti_t.th_sport
  50 #define  ti_dport ti_t.th_dport
  51 #define  ti_seq   ti_t.th_seq
  52 #define  ti_ack   ti_t.th_ack

  56 #define  ti_win   ti_t.th_win
------------------------------------------------------------tcpip.h
```

5.4 Numeri di sequenza TCP

A ciascun byte di dati scambiato all'interno di una connessione TCP viene assegnato un numero di sequenza di 32 bit. Il campo *numero di sequenza* interno all'header TCP contiene il numero di sequenza del primo byte di dati del segmento. Viceversa, il campo *numero di riscontro*, presente anch'esso nell'header TCP, contiene il prossimo numero di sequenza che il mittente dell'ACK si aspetta di ricevere (e in tal modo riscontra la ricezione di tutti i dati aventi numero di sequenza strettamente inferiore ad esso).
All'interno del nostro header TCP, però, vediamo che è usato direttamente il tipo di dato `tcp_seq`. Questo chiaramente non è un tipo primitivo del linguaggio C ma è definito all'interno del file `tcp.h` come:

```
-------------------------------------------------------------tcp.h
  40 typedef u_long tcp_seq;
-------------------------------------------------------------tcp.h
```

Ma come facciamo a confrontare dei numeri di sequenza? In nostro aiuto vengono le quattro macro definite di seguito:

```
-----------------------------------------------------------tcp_seq.h
  40 #define  SEQ_LT(a,b)    ((int)((a)-(b)) < 0)
  41 #define  SEQ_LEQ(a,b)   ((int)((a)-(b)) <= 0)
  42 #define  SEQ_GT(a,b)    ((int)((a)-(b)) > 0)
  43 #define  SEQ_GEQ(a,b)   ((int)((a)-(b)) >= 0)
-----------------------------------------------------------tcp_seq.h
```

5.5 Blocco di controllo TCP

TCP mantiene un proprio blocco di controllo (definito come una struttura `tcpcb`):

```
------------------------------------------------------------tcp_var.h
  41 struct tcpcb {
  42    struct tcpiphdr *seg_next;   /*reassembly queue of received
                                       segments*/
  43    struct tcpiphdr *seg_prev;   /*reassembly queue of received
                                       segments*/
  44    short    t_state;            /*connection state*/
  45    short    t_timer[TCPT_NTIMERS]; /*tcp timers*/
  46    short    t_rxtshift;         /*log(2) of rexmt exp. backoff*/
  47    short    t_rxtcur;           /*current retransmission timeout*/
  48    short    t_dupacks;          /*#consecutive duplicate ACKs
```

```
                                        received*/
 49    u_short  t_maxseg;           /*maximum segment size to send*/
 50    char     t_force;            /*1 if forcing out a byte*/
 51    u_short  t_flags;            /*flags*/
 52    struct tcpiphdr *t_template; /*skeletal packet for transmit*/
 53    struct inpcb *t_inpcb;    /*back pointer to internet pcb*/
 54 /*
 55  * The following fields are used as in the protocol specification
 56  * See RFC783, Dec. 1981, page 21
 57  */
 58 /*send sequence variables*/
 59    tcp_seq  snd_una;         /*send unacknowledged*/
 60    tcp_seq  snd_nxt;         /*send next*/
 61    tcp_seq  snd_up;          /*send urgent pointer*/
 62    tcp_seq  snd_wl1;         /*window update seg seq number*/
 63    tcp_seq  snd_wl2;         /*window update seg ack number*/
 64    tcp_seq  iss;             /*initial send sequence number*/
 65    u_long   snd_wnd;         /*send window*/
 66 /*receive sequence variables*/
 67    u_long   rcv_wnd;         /*receive window*/
 68    tcp_seq  rcv_nxt;         /*receive next*/
 69    tcp_seq  rcv_up;          /*receive urgent pointer*/
 70    tcp_seq  irs;             /*initial receive sequence number*/
 71 /*
 72  * Additional variables for this implementation
 73  */
 74 /*receive variables*/
 75    tcp_seq  rcv_adv;         /*advertised window by other end*/
 76 /*retransmit variables*/
 77    tcp_seq  snd_max;         /*highest sequence number sent;
 78                               *used to recognize retransmit*/
 79 /*congestion control (slow start, source quench, rexmit after
      loss*/
 80    u_long   snd_cwnd;        /*congestion controlled window*/
 81    u_long   snd_ssthresh;    /*snd_cwnd size threshhold for slow
 82                               *start exponential to linear
                                   switch*/
 83 /*
 84  * Transmit timing stuff. See below for scale of srtt and rttvar.
 85  * "Variance" is actually smoothed difference.
 86  */
 87    short t_idle;             /*inactivity time*/
 88    short t_rtt;              /*round-trip time*/
 89    tcp_seq  t_rtseq;         /*sequence number being timed*/
 90    short t_srtt;             /*smoothed round-trip time*/
 91    short t_rttvar;           /*variance in round-trip time*/
 92    u_short  t_rttmin;        /*minimum rtt allowed*/
 93    u_long   max_sndwnd;      /*largest window peer has offered*/

106 };

107 #define  intotcpcb(ip)    ((struct tcpcb *)(ip)->inp_ppcb)
108 #define  sototcpcb(so)    (intotcpcb(sotoinpcb(so)))
-------------------------------------------------------------tcp_var.h
```

La discussione delle variabili presenti all'interno del blocco di controllo è rimandata al momento in cui verranno utilizzate all'interno della nostra trattazione. Per il momento è sufficiente sapere che, al momento della creazione

della socket viene invocata la funzione `tcp_newtcpcb` allo scopo di creare e inizializzare un nuovo blocco di controllo.

5.6 Funzione `tcp_usrreq`

Adesso che abbiamo introdotto le strutture dati alla base dell'implementazione di TCP di Net/3 possiamo andare ad analizzare come vengono servite le richieste utente (in particolare `PRU_SEND`, `PRU_SENDOOB`, `PRU_RCVD`). Nel capitolo 3 abbiamo visto come le richieste di lettura e scrittura all'interno del socket buffer implicassero l'utilizzo di quella che genericamente è identificata come funzione `pr_usrreq`[14]:

```
------------------------------------------------------------tcp_usrreq.c
  45 int
  46 tcp_usrreq(so, req, m, nam, control)
  47 struct socket *so;
  48 int   req;
  49 struct mbuf *m, *nam, *control;
  50 {

  65    s = splnet();

  82    switch (req) {

 201       /*
 202        * After a receive, possibly send window update to peer
 203        */
 204       case PRU_RCVD:
 205          (void) tcp_output(tp);
 206          break;

 207       /*
 208        * Do a send by putting data in output queue and updating
 209        * urgent marker if URG set. Possibly send more data.
 210        */
 211       case PRU_SEND:
 212          sbappend(&so->so_snd, m);
 213          error = tcp_output(tp);
 214          break;

 242       case PRU_SENDOOB:
 243          if (sbspace(&so.so_snd) < -512) {
 244             m_freem(m);
 245             error = ENOBUFS;
 246             break;
 247          }
 248          /*
 249           * According to RFC961 (Assigned Protocols).
 250           * the urgent pointer points to the last octet
 251           * of urgent data. We continute, however,
 252           * to consider it to indicate the first octet
 253           * of data past the urgent section.
 254           * Otherwise, snd_up should be one lower.
 255           */
```

14 Notiamo come a livello di socket venga utilizzata una interfaccia *protocol indipendent* e poi sia la socket stessa a mappare `pr_usrreq` sul protocollo specifico (`tcp_usrreq`).

```
256          sbappend(&so->so_snd, m);
257          tp->snd_up = tp->snd_una + so->so_snd.sb_cc;

258          tp->t_force = 1;
259          error = tcp_output(tp);
260          tp->t_force = 0;

261          break;

276     default:
277        panic("tcp_usrreq");
278     }

283 }
-------------------------------------------------------------tcp_usrreq.c
```

Intuitivamente possiamo dire che, in relazione agli argomenti da noi trattati, la funzione `tcp_usrreq` opera da interfaccia tra il livello dei socket e quello dei protocolli. È, infatti, abbastanza evidente che, dopo aver eventualmente aggiunto dati al buffer di spedizione[15], `tcp_usrreq` si limita ad agire da dispatcher rimandando il problema a `tcp_output` (righe 205, 213 e 259).
Nel prossimo capitolo andremo, perciò, a vedere come Net/3 prevede di risolvere le problematiche relative agli argomenti al centro della nostra discussione quando ci troviamo nella fase di output.

15 È interessante notare l'utilizzo del meccanismo di forzatura (`tp->t_force`) durante la gestione della richiesta di invio di dati fuori banda (`PRU_SENDOOB`).

6. TCP Output

6.1 Introduzione

La funzione `tcp_output` viene invocata ogniqualvolta si renda necessario l'invio di un segmento lungo la connessione.
`tcp_output` inizialmente si occupa di determinare se il segmento possa essere spedito oppure no. TCP, infatti, controlla numerosi fattori prima di procedere alla spedizione del segmento. In particolare ci riferiamo ai meccanismi di controllo di flusso e di congestione.
Per contro, va detto che alcune funzioni settano determinati flag in modo da forzare `tcp_output` a spedire il segmento, come ad esempio il flag `TF_ACKNOW` che significa che il riscontro deve essere inviato immediatamente.

6.2 Struttura generale di `tcp_output`

`tcp_output` è una funzione molto estesa, perciò andremo ad analizzarla a sezioni. Dopo aver dichiarato le variabili necessarie, la funzione si occupa di verificare se TCP è in attesa di un ACK da parte dell'altro capo della comunicazione (riga 61). `idle` è "vero" se il massimo numero di sequenza spedito (`snd_max`) è uguale al più vecchio numero di sequenza non riscontrato. Se non c'è attesa di riscontri da parte dell'altro capo della comunicazione e non è stato ricevuto alcun segmento nell'ultimo RTT, la finestra di congestione viene forzata a un segmento (`t_maxseg` byte). Questo comportamento impone l'utilizzo dell'algoritmo di *Slow Start* quando sarà effettuato il calcolo del nuovo valore della finestra di congestione. La scelta di forzare l'algoritmo di controllo a Slow Start è dovuta al fatto che, quando abbiamo una pausa significativa nella trasmissione di dati (dove con "significativo" intendiamo almeno un round-trip time), le condizioni della rete potrebbero essere cambiate da quanto stimato in precedenza. Net/3 assume, quindi, di trovarsi nella situazione peggiore possible e perciò prevede di tornare ad operare in slow start.

```
------------------------------------------------------------tcp_output.c
 43 int
 44 tcp_output(tp)
 45 struct tcpcb *tp
 46 {
 47    struct socket *so = tp->inpcb->inp_socket;
 48    long     len,win;
 49    int      off,flags,error;
 50    struct mbuf *m;
 51    struct tcpiphdr *ti;
 52    u_char   op[MAX_TCPOPTLEN];
 53    unsigned optlen, hdrlen;
 54    int      idle,sendalot;

 55    /*
 56     * Determine length of data should be transmitted
```

```
 57     * and flags that will be used.
 58     * If there are some data critical controls (SYN,RST)
 59     * to send, then transmit; otherwise, investigate further.
 60     */
 61    idle = (tp->snd_max == tp->snd_una);
 62    if (idle && tp->t_idle >= tp->t_rxtcur)
 63       /*
 64        * We have been idle for "a while" and no acks are
 65        * expected to clock out any data we send --
 66        * slow start to get ack "clock" running again
 67        */
 68       tp->snd_cwnd = tp->t_maxseg;

 69  again:
 70    sendalot = 0; /*set nonzero if more than one segment to
                       output*/

             /*determinazione della necessità di spedire un segmento*/
             /*ed esecuzione di "goto send" in caso affermativo*/

218    /*
219     * No reason to send a segment, just return
220     */
221    return (0);

222  send:

             /*costruzione segmento e invocazione ip_output()*/

489    if (sendalot)
490       goto again;
491    return (0);
492 }
-------------------------------------------------------------tcp_output.c
```

Da notare è l'utilizzo di `sendalot`: se `tcp_output` determina la necessità di inviare più di un segmento, `sendalot` viene settato a 1 e la funzione prova a spedire un altro segmento. Questo implica che una singola a `tcp_output` può comportare l'invio di più di un segmento.

6.3 Determinazione della necessità di spedire un segmento

In alcune situazioni `tcp_output` viene invocata ma non è generato alcun segmento. Si pensi ad esempio al caso della richiesta `PRU_RCVD` generata quando il livello socket rimuove dati dal buffer di ricezione e li passa al processo. È possibile che il processo abbia rimosso dati a sufficienza da far sì che TCP debba mandare alla controparte un segmento con l'aggiornamento della finestra. Ma questa è solamente una possibilità, non una certezza. La prima metà di `tcp_output` determina se sia necessario o meno inviare un segmento.

```
-------------------------------------------------------------tcp_output.c
 71    off = tp->snd_nxt - tp->snd_una;
 72    win = min(tp->snd_wnd, tp->cwnd);

106    len = min(so->so_snd.sb_cc, win) - off
```

```
124    if (len > tp->t_maxseg) {
125       len = tp->t_maxseg;
126       sendalot = 1;
127    }

130    win = sbspace(&so->so_rcv);
-------------------------------------------------------tcp_output.c
```

Dopo aver determinato (righe 71 – 72) l'offset (`off`) in byte dall'inizio del send buffer al primo byte da spedire e la finestra di invio (`win`)[16], la funzione determina la quantità di dati da spedire (riga 106). `len` è calcolato come il minimo tra i byte contenuti all'interno del send buffer e `win`. Alla quantità trovata viene sottratto l'offset in modo da tenere in considerazione i byte presenti nel send buffer che sono già stati spediti e attendono un riscontro[17].
Se la quantità di dati eccede il singolo segmento, `len` viene settato a 1 MSS ed il flag `sendalot` viene settato a 1 (righe 124 – 127). In questo modo la procedura già svolta sarà ripetuta per i byte rimanenti.
A questo punto possiamo cambiare il valore di `win` (riga 130): per mezzo della funzione `sbspace` viene settato alla quantità di spazio disponibile nel receive buffer. In questo modo, da essere utilizzato come massima quantità spedibile, `win` diviene il valore della finestra da comunicare all'altra entità TCP presente nella comunicazione.
I sistemi di controllo di flusso basati su finestra (come quello presente in TCP) possono essere vittime di quello che viene definito *silly window syndrome* (SWS) [4]. Quando occorre, invece di scambiare segmenti di dimensione massima, vengono scambiate piccole quantità di dati lungo la connessione.
Questo fenomeno può essere causato da entrambi i capi: il ricevitore può comunicare piccole finestre (invece di attendere finché non sia possibile comunicarne una più grande) e il trasmettitore può spedire piccole quantità di dati (invece di attendere ulteriori dati, per spedire un segmento più grande). Le operazioni necessarie a evitare questo fenomeno devono essere eseguite da entrambi i capi della comunicazione.
Il mittente non deve trasmettere a meno che non sia verificata una delle seguenti condizioni:

- può essere spedito un segmento di dimensione massima (righe 142 – 143);
- possiamo spedire almeno la metà della massima dimensione della finestra comunicata dal ricevente durante la comunicazione (righe 149 –

16 Così come specificato dalle RFC, la finestra di invio è il minimo tra la *receive window* comunicata dall'altro capo della comunicazione e la *congestion window*.

17 Osserviamo che in questo modo len potrebbe scendere sotto zero. Si tratta di alcuni casi particolari e Net/3 ne prevede la gestione. Ai fini della nostra trattazione però tali casi risultano non interessanti dunque rimandiamo al lettore l'approfondimento di questa problematica.

150)[18];

- possiamo trasmettere tutto ciò che abbiamo e non stiamo attendendo un ACK (righe 144 – 146)[19].

Notiamo che oltre alle condizioni utilizzate per evitare la SWS, ne sono presenti altre due:

- controllo di forzatura della spedizione (righe 147 – 148);
- controllo di ritrasmissione (righe 151 – 152).

```
-------------------------------------------------------------tcp_output.c
131    /*
132     * Sender silly window avoidance. If connection is idle
133     * and can send all data, a maximum segment,
134     * at least a maximum default-sized segment do it,
135     * or are forced, do it; otherwise don't bother.
136     * If peer's buffer is tiny, then send
137     * when window is at least half open.
138     * If retransmitting (possibly after persist timer forced us
139     * to send into a small window), then must resend.
140     */
141    if (len) {
142       if (len == tp->t_maxseg)
143          goto send;
144       if ((idle || tp->flags & TF_NODELAY) &&
145          len + off >= so->so_snd.sb_cc)
146          goto send;
147       if (tp->t_force)
148          goto send;
149       if (len >= tp->max_sndwnd / 2)
150          goto send;
151       if (SEQ_LT(tp->snd_nxt,tp->snd_max))
152          goto send;
153    }
-------------------------------------------------------------tcp_output.c
```

Allo stesso modo dobbiamo implementare il controllo dal lato ricevente durante la *window update* (evitando di comunicare finestre troppo piccole). L'algoritmo classico per il ricevente prevede di non comunicare una finestra maggiore di quella già comunicata (che potrebbe essere zero) a meno che non possa essere incrementata di 1 MSS o di almeno la metà dello spazio disponibile nel buffer di ricezione.

```
-------------------------------------------------------------tcp_output.c
154    /*
155     * Compare available window to amount of window
156     * known to peer (as advertised window less
157     * next expected input). If the difference is at least two
158     * max size segments, or at least 50% of the maximum possible
159     * window, then want to send a window update to peer.
160     */
161    if (win > 0) {
162       /*
163        * “adv” is the amount we can increase the window,
```

18 Permette di gestire la comunicazione con host che comunicano sempre finestre molto piccole, spesso inferiori alla dimensione di un segmento.

19 Evita che il mittente spedisca piccoli segmenti quando abbiamo dati non ancora riscontrati.

```
164         * taking into account that we are limited by
165         * TCP_MAXWIN << tcp->rcv_scale
166         */
167        long  adv = min(win,(long) TCP_MAXWIN << tcp->rcv_scale) -
168        (tcp->rcv_adv - tcp->rcv_nxt);

169        if (adv >= (long) (2 * tp->t_maxseg))
170           goto send;
171        if (2 * adv >= (long) so->so_rcv.sb_hiwat)
172           goto send;
173     }
-------------------------------------------------------------tcp_output.c
```

La porzione di `tcp_output` sopra riportata, mostra come Net/3 implementa la fase di *window update.* L'espressione

```
min(win, (long) TCP_MAXWIN << tp->rcv_scale)
```

calcola il minore tra lo spazio disponibile nel buffer (`win`) e la massima dimensione per la finestra concessa per questa connessione. Notiamo l'utilizzo di "`<< tp->rcv_scale`". Ciò avviene poiché all'inizio della comunicazione le due entità concordano una scala di rappresentazione della finestra di ricezione[20]. Questo valore dunque rappresenta la dimensione massima della finestra che TCP può comunicare a questo punto della connessione. L'espressione:

```
(tp->rcv_adv - tp->rcv_nxt)
```

calcola la quantità di byte rimanenti al riguardo dell'ultima comunicazione della finestra da parte di TCP. Sottraendo questo dal massimo valore disponibile per la finestra otteniamo `adv`, ovvero l'aumento, in byte, della dimensione della finestra (righe 167 – 168).
Se la finestra è aumentata di almeno 2 MSS oppure di almeno il 50% del valore massimo possibile per la finestra (*high-water mark* del socket buffer di ricezione), Net/3 prevede che venga inviato un segmento di window update (righe 169 – 172).

6.4 Spedizione di un segmento

L'ultima parte di `tcp_ouput` si occupa di costruire e spedire il segmento:

```
-------------------------------------------------------------tcp_output.c
278    /*
279     * Grab a header mbuf, attaching a copy of data to
280     * be transmitted, and initialize the header from
281     * the template for sends on this connection
282     */
283    if (len) {

293       MGETHDR(m, M_DONTWAIT, MT_HEADER);
294       if (m == NULL) {
295          error = ENOBUFS;
296          goto out;
297       }
```

20 In questo modo è assicurata una rappresentazione uniforme delle finestre di ricezione su entrambi i capi della comunicazione.

```
298       m->data += max_linkhdr;
299       m->m_len = hdrlen;
300       if (len <= MHLEN - hdrlen - max_linkhdr) {
301          m_copydata(so->so_snd.sb_mb, off, (int) len,
302                     mtod(m, caddr_t) + hdrlen);
303          m->m_len +=len;
304       } else {
305          m->m_next = m_copy(so->so_snd.sb_mb, off, (int) len);
306          if (m->m_next == 0)
307             len = 0;
308       }
----------------------------------------------------------tcp_output.c
```

Nel caso ci siano dati da spedire (`len != 0`), viene allocato un header di pacchetto con `MGETHDR`. Questo verrà utilizzato per gli header TCP e IP, e possibilmente per i dati (se c'è spazio a sufficienza). Poiché `tcp_output` viene invocata anche come interruzione di un'altra system call (p.e. `tcp_input`), viene specificato `M_DONTWAIT` per evitare che si perda troppo tempo in attesa di spazio. Se occorre un errore, saltiamo direttamente alla fase di `out` (righe 293 – 297).

A questo punto siamo in grado di procedere alla copia dei dati: se la quantità dei dati è inferiore a 44 byte (100 – 40 – 16, assumendo che non ci siano opzioni TCP), i dati sono copiati direttamente dal socket buffer di spedizione all'interno del mbuf che svolge il ruolo di header di pacchetto (per mezzo della funzione `m_copydata`, righe 300 – 303). Altrimenti utilizziamo la funzione `m_copy`, per creare una nuova catena di mbuf in cui copiare i dati del socket send buffer, e attacchiamo il risultato ritornato al nostro header di pacchetto (righe 304 – 307).

```
----------------------------------------------------------tcp_output.c
317       } else {

326       MGETHDR(m, M_DONTWAIT, MT_HEADER);
327       if (m == NULL) {
328          error = ENOBUFS;
329          goto out;
330       }
331       m->data += max_linkhdr;
332       m->m_len = hdrlen;
333    }
----------------------------------------------------------tcp_output.c
```

Nel caso in cui, invece, non ci siano dati da spedire Net/3 si limita a costruire richiedere un mbuf che utilizzerà solamente per contenere le informazioni di intestazione TCP e IP.

A questo punto, TCP è in grado di riempire l'header del segmento che andrà a spedire[21]:

```
----------------------------------------------------------tcp_output.c
335    ti = mtod(m, struct tcpiphdr *)

347    /*
348     * If we are doing retransmission, then snd_nxt will
349     * not reflect the first unsent octet. For ACK only
```

21 Riportiamo solamente le parti che risultano interessanti ai fini di questo documento.

```
350     * packets, we do not want the sequence number of the
351     * retransmitted packet, we want the sequence number
352     * of the next unsent octet. So, if there is no data
353     * (and no SYN or FIN), use snd_max instead of snd_nxt
354     * when filling ti_seq. But if we are in persist
355     * state, snd_max might reflect one byte beyond the
356     * right edge of the window, so use snd_nxt in that case,
357     * since we know we aren't doing a retransmission.
358     * (retransmit and persist are mutually exclusive...)
359     */
360    if (len || (flags & (TH_SYN | TH_FIN)) ||
            tp->t_timer[TCPT_PERSIST])
361       ti->ti_seq = htonl(tp->snd_nxt);
362    else
363       ti->ti_seq = htonl(tp->snd_max);

364    ti->ti_ack = htonl(tp->rcv_nxt);

370    /*
371     * Calculate receive window. Don't shrink window,
372     * but avoid silly window syndrome.
373     */
374    if (win < (long)(so->so_rcv.sb_hiwat / 4) &&
            win < (long)tp->t_maxseg)
375       win = 0;
376    if (win > (long) TCP_MAXWIN << tp->rcv_scale)
377       win = (long) TCP_MAXWIN << tp->rcv_scale;
378    if (win < (long) (tp->rcv_adv - tp->rcv_nxt))
379       win = (long) (tp->rcv_adv - tp->rcv_nxt);
380    ti->ti_win = htons((u_short) (win >> tp->rcv_scale));
-------------------------------------------------------tcp_output.c
```

Notiamo (righe 360 – 363) che il numero di sequenza normalmente viene settato a `snd_nxt` (a meno che non sia verificata una delle condizioni riportate[22], in tal caso vale `snd_max`). Per quanto riguarda il numero di riscontro, invece, questo vale sempre `rcv_nxt`.

Settati i numeri di sequenza si passa al calcolo della finestra da comunicare alla controparte. Dopo aver eseguito la *silly window avoidance* (righe 370 – 375), per evitare di comunicare finestre troppo piccole, Net/3 si preoccupa di far sì che la finestra comunicata:

- non sia superiore al valore massimo consentito per questa comunicazione (righe 376 – 377);
- non sia ristretta[23] (righe 378 – 379).

Eseguiti gli opportuni controlli, alla riga 380, l'implementazione riportata provvede a inserire il valore della finestra all'interno dell'header del segmento TCP.

22 (a) non ci sono dati da spedire;
(b) né il flag SYN né quello FIN sono settati;
(c) il timer di persist non è settato.

23 Se il valore della finestra è inferiore allo spazio ancora disponibile all'interno del buffer, allora tale valore è settato allo spazio disponibile (per evitare quello che nel paragrafo 5.3 era stato definito *window shrinking*).

Net/3, una volta preparato il segmento, prima di poterlo trasmettere deve andare ad aggiornare tutte le variabili interessate all'interno del blocco di controllo TCP:

```
------------------------------------------------------------tcp_output.c
404    if(tp->t_force == 0 || tp->t_timer[TCPT_PERSIST] == 0) {

417       tp->snd_nxt +=len;
418       if (SEQ_GT(tp->snd_nxt, snd_max)) {
419          tp->snd_max = tp->snd_nxt;

429       }
430       /*
431        * Set retransmit timer if not currently set,
432        * and not doing an ack or a keepalive probe.

437        */
438       if (tp->t_timer[TCPT_REXMT] == 0 &&
439          tp->snd_nxt != tp->snd_una) {
440          tp->t_timer[TCPT_REXMT] = tp->rxtcur;

445       }
446    } else if (SEQ_GT(tp->snd_nxt + len, snd_max))
447       tp->snd_max = tp->snd_nxt + len;
------------------------------------------------------------tcp_output.c
```

Se TCP è "forzato" a spedire (o non si trova in stato di persist), vengono aggiornate le variabili che tengono traccia dei numeri di sequenza di spedizione e viene attivato (nel caso non lo sia già) il timer di ritrasmissione (righe 404 – 445). Se, invece, TCP non è "forzato" a spedire, Net/3 si limita ad aggiornare, se necessario, il valore di `snd_max`.

Il segmento è pronto e le variabili sono aggiornate. Rimane, quindi, da effettuare l'invio dei dati[24]:

```
------------------------------------------------------------tcp_output.c
463    error = ip_output(m, tp->t_inpcb->inp_options, &tp->t_inpcb->
464                      inp_route, so->so_options & SO_DONTROUTE, 0);
465    if (error)
466     out:

476       return (0);
477    }
------------------------------------------------------------tcp_output.c
```

A questo punto (in assenza di errori) siamo certi che i dati siano stati inviati. In precedenza abbiamo notato che, nel calcolo della finestra da comunicare, tenevamo conto di quanto avevamo comunicato in precedenza. Dunque abbiamo bisogno, nel caso il valore della finestra sia superiore a quello precedente, di memorizzare il cambiamento:

```
------------------------------------------------------------tcp_output.c
479    /*
480     * Data sent (as far as we can tell).
481     * If this advertises a larger window than any other segment,
```

24 Ci limitiamo a riportare l'invocazione della funzione `ip_output` in quanto il problema esula dagli scopi che ci siamo predisposti.

```
 482     * then remember the size of the advertised window.

 484     */
 485    if (win > 0 && SEQ_GT(tp->rcv_nxt + win, tp->rcv_adv))
 486      tp->rcv_adv = tp->rcv_nxt + win;
-------------------------------------------------------------tcp_output.c
```

7. TCP Input

7.1 Introduzione

L'elaborazione dell'input in TCP è la porzione di codice più grande tra quelle presenti in `netinet`. La funzione `tcp_input` si estende per circa 1100 linee di codice. Tuttavia, l'elaborazione dei segmenti in arrivo non è complicata, solamente lunga e dettagliata.

La funzione `tcp_input` è invocata da `ipintr` quando viene ricevuto un pacchetto avente il campo protocollo settato a TCP e, perciò, è eseguita al livello di interruzione software.

Diamo uno schema generale dell'esecuzione di `tcp_input`:

```
void
tcp_input()
{
      checksum TCP header and data;
 findpcb:
      locate PCB for segment;
      if (not found)
            goto dropwithreset;
      reset idle time to 0 and keepalive timer to 2 hours;
      process options if not LISTEN state;
      if(packet matched by header prediction) {
            completely process received segment;
            return;
      }
      drop TCP and IP header, including TCP options;
*     calculate receive window;
      switch (tp->state) {
            case TCPS_LISTEN:
                  if SYN flag set, accept new connection
                  request;
                  goto trimthenstep6;
            case TCPS_SYN_SENT:
                  if ACK of our SYN, connection completed;
 trimthenstep6:
                  trim any data not within window;
                  goto step6;
      }
      process RFC 1323 timestamp;
*     check if some data byte are within the receive window;
*     trim data segment to fit within window;
      if (RST flag set) {
            process depending on state;
            goto drop;
      }
*     if (ACK flag set) {
            if (SYN_RCVD state)
                  passive open or simultaneous open complete;
*           if (duplicate ACK)
*                 fast recovery and fast retransmit
algorithms;
            update RTT estimators if segment timed;
```

```
*           open congestion window;
*           remove ACKed data from send buffer;
            change state if in FIN_WAIT_1, CLOSING or
LAST_ACK
            state;
      }
 step 6:
      update window information;
      process URG flag;
 dodata:
*     process data in segment, add to reassembly queue;
      if (FIN flag is set)
            process depending on state;
      if (SO_DEBUG socket option)
            tcp_trace(TA_INPUT);
      if (need output || ACK now)
            tcp_output();
      return;
 dropafterack:
      tcp_output() to generate ACK;
      return;
 dropwithreset:
      tcp_respond() to generate RST;
      return;
 drop:
      if (SO_DEBUG socket option)
            tcp_trace(TA_DROP);
      return;
}
```

Figura 7.1
`tcp_input`: schema generale

Ai fini di questo documento ci limiteremo ad analizzare le fasi riguardanti l'elaborazione del buffer di ricezione, di quello di spedizione e dei controlli di flusso e di congestione (in grassetto in Figura 6.1).

7.2 Calcolo della finestra di ricezione

```
-------------------------------------------------------------tcp_input.c
443   /*
444    * Calculate amount of space in receive window,
445    * and then do TCP input processing.
446    * Receive window is amount of space in rcv queue,
447    * but not less than advertised window.
448    */
449   {
450      int   win;
451      win = sbspace(&so->so_rcv);
452      if (win < 0)
453         win = 0;
454      tp->rcv_wnd = max(win, (int) (tp->rcv_adv - tp->rcv_nxt));
455   }
-------------------------------------------------------------tcp_input.c
```

`win` viene settato al numero di byte disponibili all'interno del socket buffer di ricezione (riga 451). `rcv_adv` meno `rcv_nxt` è il valore attuale della *advertised*

window. Il valore della finestra di ricezione sarà dato dal massimo tra questi due valori (riga 454).
Questo valore è calcolato ora, poiché successivamente il codice della funzione `tcp_input` dovrà determinare quanti dei dati ricevuti sono compresi all'interno della finestra di ricezione. Qualsiasi dato al di fuori di essa sarà scartato: i dati a sinistra della finestra saranno considerati duplicati poiché sono già stati ricevuti e riscontrati; i dati a destra, invece, non dovrebbero essere stati spediti da parte dell'altro capo della comunicazione.

7.3 Potatura del segmento affinché sia contenuto all'interno della finestra di ricezione

Questa sezione si occupa di "spuntare" (*trim*) il segmento in modo che sia completamente contenuto all'interno della finestra di ricezione:

- i dati duplicati all'inizio del segmento ricevuto sono scartati, e
- i dati che risiedono dopo la fine della finestra di ricezione sono scartati (a partire dal limite destro della finestra di ricezione).

Ciò che rimane è composto dai dati (nuovi) interni alla finestra di ricezione. Il codice riportato di seguito verifica se ci sono dati duplicati all'inizio del segmento:

```
------------------------------------------------------------tcp_input.c
635    todrop = tp->rcv_nxt - ti->ti_seq;
636    if (todrop > 0) {

646       if (todrop >= ti->ti_len) {

671                goto dropafterack;

676       }
677       m_adj(m, todrop);
678       ti->ti_seq += todrop;
679       ti->ti_len -= todrop;

686    }
------------------------------------------------------------tcp_input.c
```

Se il numero di sequenza del segmento ricevuto (`ti_seq`) è inferiore a quello atteso (`rcv_nxt`), i dati all'inizio del segmento sono vecchi e `todrop` sarà superiore a 0 (righe 635 – 636). Questi dati sono già stati riscontrati e inviati al processo.
Se la quantità di dati duplicati è maggiore o uguale della dimensione del segmento stesso significa che l'intero pacchetto è duplicato (riga 646). Tale pacchetto deve essere scartato e deve essere inviato un ACK (che in questo caso sarà duplicato) all'altro capo della comunicazione (riga 671). Nel caso in cui, invece, parte dei dati non risultino ancora ricevuti, Net/3 si occupa di rimuovere dal segmento i dati già riscontrati (righe 677 – 679).
E quando i dati terminano a destra della nostra finestra di ricezione?

```
------------------------------------------------------------tcp_input.c
 697    /*
 698     * If segments ends after window, drop tailing data
 699     * (and PUSH and FIN); if nothing left, just ACK.
 700     */
 701    todrop = (ti->ti_seq + ti->ti_len) -
                 (tp->rcv_nxt + tp->rcv_wnd);
 702    if (todrop > 0) {

 704       if(todrop >= ti->ti_len) {

 730            goto dropafterack;
 731       }

 733       m_adj(m, -todrop);
 734       ti->ti_len -= todrop
 736    }
------------------------------------------------------------tcp_input.c
```

`todrop` contiene i dati a destra della finestra di ricezione (riga 701) e che quindi devono essere rimossi.
Se la quantità di dati da scartare è superiore alla lunghezza del segmento stesso, allora dobbiamo scartarlo interamente dopo aver inviato il relativo ACK (riga 704 e riga 730). Altrimenti, per mezzo della funzione `m_adj`, andiamo a togliere `todrop` byte di dati dal fondo della catena di mbuf ricevuta (righe 733 – 734).

7.4 ACK Processing: ACK duplicati

Net/3 gestisce gli ACK duplicati utilizzando gli algoritmi di *Fast Recovery* e *Fast Retransmit*[25] [5].
Osserviamo che un ACK è accettabile se appartiene all'intervallo (`snd_una`,`snd_max`]. Il primo test del campo di riscontro viene fatto solamente con `snd_una`. Il test con `snd_max` sarà fatto successivamente.

```
------------------------------------------------------------tcp_input.c
 831       if (SEQ_LEQ(ti->ti_ack, tp->snd_una)) {
 832          if (ti->ti_len == 0 && tiwin == tp->snd_wnd) {

 834             /*
 835              * If we have outstanding data (other than
 836              * a window probe), this is a completely
 837              * duplicate ack (ie, window info didn't
 838              * change), the ack is the biggest we've
 839              * seen and we've seen exactly our rexmt
 840              * threshold of them, assume a packet
 841              * has been dropped and retransmit it.
 842              * Kludge snd_nxt & the congestion
 843              * window so we send only this one
 844              * packet.
 845              *
 846              * We know we're losing the current
 847              * window size so do congestion avoidance
 848              * (set sstresh to half the current window
 849              * and pull our congestion window back to
 850              * the new sshtresh).
```

25 I due algoritmi sono separati ma normalmente vengono implementati insieme.

```
851             *
852             * Dup acks mean that packets have left the
853             * network (they're now cached at the receiver)
854             * so bump cwnd by the amount in the receiver
855             * to keep a costant cwnd packets in the
856             * network.
857             */
-------------------------------------------------------------tcp_input.c
```

I due test (quello con `snd_una` e `snd_max`) vengono separati in modo da poter effettuare le seguenti cinque verifiche. Se

- il campo di riscontro è inferiore o uguale a `snd_una`, e
- la lunghezza del segmento ricevuto è 0, e
- la finestra comunicata (`tiwin`) non è stata modificata, e
- TCP ha dei dati inviati ma non ancora riscontrati (ovvero il timer di ritrasmissione è diverso da 0), e
- il segmento ricevuto contiene l'ACK più grande ricevuto (il campo di riscontro è uguale a `snd_una`),

allora il segmento ricevuto è completamente un ACK duplicato.
TCP tiene conto degli ACK duplicati ricevuti in sequenza (nella variabile `t_dupacks`), e quando il numero raggiunge la soglia di 3 (`tcprexmtthresh`), il segmento che apparentemente è andato perso viene ritrasmesso[26].
La ricezione di un ACK duplicato comunica, inoltre, a TCP che il segmento ha "lasciato la rete", perché l'altro capo della comunicazione lo ha riscontrato con un ACK duplicato (dopo, ovviamente, averlo ricevuto). L'algoritmo di *Fast Recovery* dice che, dopo aver ricevuto una certa quantità di ACK duplicati, TCP dovrebbe entrare in Congestion Avoidance.
Dunque, quando tutti e cinque i test sono verificati, Net/3 processa i segmenti in arrivo in base al numero di ACK duplicati ricevuti:

- `t_dupack` è uguale a 3 (`tcprexmtthresh`): entra in Congestion Avoidance e il segmento perso viene ritrasmesso;
- `t_dupack` è maggiore di 3: incrementa la congestion window e esegue `tcp_output`;
- `t_dupack` è inferiore a 3: non fa niente.

```
-------------------------------------------------------------tcp_input.c
858             if (tp->timer[TCPT_REXMT] == 0 ||
859                ti->ti_ack != tp->snd_una)
860                tp->t_dupacks = 0;
861             else if (++tp->t_dupacks == tcprexmtthresh) {
862                tcp_seq onxt = tp->snd_nxt;
863                u_int win =
864                   min(tp->snd_wnd, tp->snd_cwnd) / 2 /
865                      tp->t_maxseg;

866                if (win < 2)
867                   win = 2;
868                tp->snd_ssthresh = win * tp->t_maxseg;
```

26 Algoritmo di Fast Retransmission

```
869                tp->t_timer[TCPT_REXMT] = 0;

871                tp->snd_nxt = ti->ti_ack;
872                tp->snd_cwnd = tp->t_maxseg;
873                (void) tcp_output(tp);
874                tp->snd_cwnd = tp->snd_ssthresh +
875                   tp->t_maxseg * tp->t_dupacks;
876                if(SEQ_GT(onxt, tp->snd_nxt)) {
877                   tp->snd_nxt = onxt;
878                goto drop;
879             } else if (tp->t_dupacks > tcprexmtthresh) {
880                tp->snd_cwnd += tp->t_maxseg;
881                (void) tcp_output(tp);
882                goto drop;
883             }
884          } else
885             tp->t_dupacks = 0;
886          break;           /*beyond ACK processing (to step 6)*/
887       }
-------------------------------------------------------------tcp_input.c
```

Quando il numero di ACK duplicati raggiunge `tcprexmtthresh`, il valore di `snd_nxt` viene salvato in `onxt` e la soglia di slow start (`ssthresh`) è settata alla metà della congestion window corrente (con un valore minimo di 2 MSS). Inoltre viene disattivato il timer di ritrasmissione, per evitare ritrasmissioni dovute alla scadenza del timer nel caso in cui un segmento fosse attualmente temporizzato (righe 861 – 869).

`snd_nxt` viene settato al numero di sequenza iniziale del segmento che sembra andato perso (il campo riscontro del segmento contenente l'ACK duplicato) e la finestra di congestione viene settata a 1 MSS. Questo impone a `tcp_output` la spedizione del solo segmento andato perso (righe 871 – 873).

Effettuata la ritrasmissione, la finestra di congestione viene impostata al valore della soglia di slow start sommato al numero di segmenti che sono stati *messi in cache* dall'altro capo della comunicazione (righe 874 – 875). Con *messi in cache*, intendiamo il numero di segmenti *out-of-order* che l'entità TCP con cui comunichiamo ha ricevuto e per i quali ha generato ACK duplicati. Questi segmenti vengono salvati in una coda di riassemblaggio poiché non possono essere passati al processo fino a che non arriva il segmento mancante.

Il valore di `snd_nxt` viene poi impostato al massimo tra il suo valore precedente (`onxt`) e quello attuale[27]. A questo punto l'elaborazione è terminata e possiamo scartare il segmento contenente l'ACK duplicato (righe 876 – 878).

E quando riceviamo ulteriori ACK duplicati? Il segmento viene ritrasmesso solamente quando raggiungiamo la soglia dei 3 ACK duplicati. Ogni successivo ACK duplicato significa che un altro pacchetto ha lasciato la rete. La finestra di congestione viene incrementata di 1 MSS, `tcp_output` spedisce il prossimo segmento in sequenza e l'ACK duplicato viene scartato (righe 879 – 883).

Quando, invece, il segmento contiene un ACK duplicato ma la sua lunghezza è

27 Il suo valore attuale potrebbe essere stato modificato dalla chiamati di `tcp_output`

diversa da 0 oppure il valore della *advertised window* è cambiato, solamente il primo dei cinque test è verificato. Net/3 dunque si limita ad azzerare il conto degli ACK duplicati (righe 884 – 885).
Infine notiamo che il `break` (riga 886) viene eseguito solamente in tre casi:

- solamente il primo dei cinque test è verificato;
- solamente i primi tre dei cinque test sono verificati;
- l'ACK è duplicato, ma ci troviamo al di sotto della soglia dei 3 ACK duplicati.

Per ognuno di questi casi l'ACK è comunque un duplicato e dunque l'elaborazione dell'ACK viene interrotta e l'elaborazione *salta* al passo `step6`.

7.5 ACK Processing: ACK non-duplicati

L'elaborazione degli ACK continua come segue:

```
-------------------------------------------------------------tcp_input.c
 888       /*
 889        * If the congestion window has inflated to account
 890        * for the other side's cached packets, retract it.
 891        */
 892       if (tp->t_dupacks > tcprexmtthresh &&
 893           tp->snd_cwnd > tp->snd_ssthresh)
 894           tp->snd_cwnd = tp->snd_ssthresh;
 895       tp->t_dupacks = 0;

 896       if (SEQ_GT(ti->ti_ack,tp->snd_max)) {

 898          goto dropafterack;
 899       }
 900       acked = ti->ti_ack - tp->snd_una;
-------------------------------------------------------------tcp_input.c
```

Se il numero di ACK duplicati consecutivi eccede la soglia di 3, questo è il primo ACK non duplicato dopo una sequenza di almeno quattro riscontri duplicati. L'algoritmo di *Fast Recovery* è dunque terminato. Poiché la finestra di congestione è stata incrementata di un segmento per ogni ACK duplicato dopo il terzo, se attualmente supera la soglia di slow start, dobbiamo riportarla al valore di `ssthresh`. Il contatore relativo agli ACK duplicati consecutivi viene riportato a 0 (righe 888 – 895).
Ricordiamo, poi, che un riscontro era accettabile se cadeva nell'intervallo (`snd_una`,`snd_max`]. Se il campo ACK del segmento è superiore al valore di `snd_max`, allora l'altro capo sta riscontrando dati che non sono ancora stati inviati. Nonostante questa situazione accada molto raramente (e sia relativa principalmente alle connessioni ad alta velocità) dobbiamo tenere in considerazione questa eventualità e scartare l'ACK appena ricevuto (righe 896 – 899).
Una volta verificato che il valore del campo riscontro è accettabile, possiamo andare a calcolare la quantità (`acked`) di byte riscontrati (riga 900).
A questo punto possiamo andare ad aggiornare il timer di ritrasmissione:

```
------------------------------------------------------------tcp_input.c
916       /*
917        * If all outstanding data is acked, stop retransmit
918        * timer and remember to restart (more output or persist).
919        * If there is more data to be acked, restart retransmit
920        * timer, using current (possibly backed-off) value.
921        */
922       if (ti->ti_ack == tp->snd_max) {
923          tp->t_timer[TCPT_REXMT] = 0;
924          needoutput = 1;
925       } else if (tp->t_timer[TCPT_PERSIST] == 0)
926          tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;
------------------------------------------------------------tcp_input.c
```

Se il campo riscontro del segmento ricevuto (`ti_ack`) uguaglia il massimo numero di sequenza spedito da TCP (`snd_max`), tutti i dati inviati sono stati riscontrati. In tal caso il timer di ritrasmissione viene disattivato e il flag `needoutput` viene posto a 1. Questo causa l'invocazione di `tcp_output` alla fine della funzione corrente. Poiché non ci sono più dati in attesa di essere riscontrati, TCP potrebbe avere altri dati da inviare che non potevano essere spediti in precedenza per mancanza di spazio nel buffer (righe 916 – 924).
Nel caso, invece, che non vengano riscontrati tutti i dati presenti nel send buffer, il timer di ritrasmissione viene resettato usando il valore corrente di `t_rxtcur` (righe 925 – 926).
Net/3, una volta sistemati i timer, prevede di passare all'implementazione degli algoritmi di Slow Start e Congestion Avoidance (per quanto riguarda l'incremento della finestra di congestione):

```
------------------------------------------------------------tcp_input.c
927       /*
928        * When new data is acked, open the congestion window.
929        * If the window gives us less than ssthresh packets
930        * in flight, open exponentially (maxseg per packet).
931        * Otherwise open linearly: maxseg per window
932        * (maxseg^2 / cwnd per packet), plus a costant
933        * fraction of a packet (maxseg / 8) to help larger windows
934        * open quickly enough
935        */
936       {
937          u_int cw = tp->snd_cwnd;
938          u_int incr = tp->t_maxseg;

939          if (cw > tp->snd_ssthresh)
940             incr = incr * incr / cw + incr / 8;
941          tp->snd_cwnd = min(cw + incr,
                               TCP_MAXWIN << tp->snd_scale);
942       }
------------------------------------------------------------tcp_input.c
```

Una delle regole di Slow Start e Congestion Avoidance è che per ogni ACK ricevuto dobbiamo incrementare la finestra di congestione. Per default, la finestra viene incrementata di 1 MSS per ogni riscontro (Slow Start). Ma se il valore corrente della finestra di congestione è più grande di `snd_ssthresh`, allora andiamo ad incrementare `snd_cwnd` di un segmento massimo diviso per

il valore attuale della finestra di congestione (più una costante frazionale). Il termine

```
incr * incr / cw
```

equivale a

```
t_maxseg * t_maxseg / snd_cwnd
```

ovvero al valore di 1 MSS diviso per la finestra di congestione, tenendo in considerazione che `snd_cwnd` è mantenuto in byte e non in numero di segmenti (righe 927 – 942).

La parte seguente `tcp_input` di provvede a rimuovere i dati riscontrati:

```
-------------------------------------------------------------tcp_input.c
 943      if (acked > so->so_snd.sb_cc) {
 944         tp->snd_wnd -= so->so_snd.sb_cc;
 945         sbdrop(&so->so_snd, (int) so->so_snd.sb_cc);
 946         ourfinisacked = 1;
 947      } else {
 948         sbdrop(&so->so_snd, acked);
 949         tp->snd_wnd -= acked;
 950         ourfinisacked = 0;
 951      }
 952      if (so->so_snd.sb_flags & SB_NOTIFY)
 953         sowwakeup(so);
 954      tp->snd_una = ti->ti_ack;
 955      if(SEQ_LT(tp->snd_nxt,tp->snd_una))
 956         tp->snd_nxt = tp->snd_una;
-------------------------------------------------------------tcp_input.c
```

Se il numero di byte riscontrati è superiore al numero di byte di dati presenti all'interno del send buffer, allora `snd_wnd` viene decrementato del numero di byte presenti nel send buffer e TCP viene a conoscenza del fatto che il FIN inviato è stato riscontrato. Questo metodo per riscontrare il FIN inviato funziona solamente perché il FIN occupa un byte nello spazio dei numeri di sequenza (righe 943 – 946).

Altrimenti, se il numero di byte riscontrati è inferiore o uguale al numero di quelli presenti nel send buffer, i byte riscontrati vengono rimossi dal buffer (righe 947 – 951).

In seguito, `sowwakeup` sveglia tutti i processi in attesa del send buffer. `snd_una` è aggiornato in modo che contenga il byte più vecchio non riscontrato. Se questo nuovo valore è superiore a `snd_nxt` allora quest'ultimo è aggiornato (righe 951 – 956).

7.6 Elaborazione dei dati ricevuti

`tcp_input` prosegue elaborando i dati ricevuti (se ci sono) per aggiungerli in fondo al buffer di ricezione (se ricevuti in ordine) o piazzandoli nella coda di riassemblaggio (se ricevuti *out-of-order*):

```
-------------------------------------------------------------tcp_input.c
1094  dodata:
1095    /*
1096     * Process the segment text, merging it into the TCP sequencing
```

```
1097     * queue and arranging for ack of receipt if necessary.
1098     * This process logically involves adjusting tp->rcv_wnd as
1099     * data is presented to the user (this happens in tcp_usrreq.c,
1100     * case PRU_RCVD). If a FIN has already been received on this
1101     * connection then we just ignore the text.
1102     */
1103    if ((ti->ti_len || (tiflags & TH_FIN)) &&
1104        TCPS_HAVERCVDFIN(tp->t_state) == 0) {
1105        TCP_REASS(tp, ti, m, so, tiflags);

1112    } else {
1113        m_freem(m);
1114        tiflags &= ~TH_FIN;
1115    }
-----------------------------------------------------------tcp_input.c
```

I dati ricevuti vengono elaborati se:

- la lunghezza dei dati ricevuti è superiore a 0 o il flag FIN è settato, e
- un FIN non è ancora stato ricevuto su questa connessione.

La macro `TCP_REASS`[28] processa i dati (righe 1094 – 1105). Se il dato ricevuto è in sequenza (in altre parole è il prossimo dato atteso per questa connessione), `rcv_nxt` viene incrementato e i dati vengono aggiunti in *append* al socket buffer di ricezione. Se i dati ricevuti non sono in ordine, la macro invoca la funzione `tcp_reass` per aggiungere i dati alla coda di riassemblaggio relativa alla socket (e in questo modo potrebbe coprire dei buchi causati da altre ricezioni non ordinate).
Se una delle due condizioni non viene verificata, i dati vengono scartati (righe 1112 – 1113).

28 Ved. Capitolo 8

8. Funzioni di supporto

8.1 Introduzione

In questo capitolo andiamo ad introdurre due funzioni *di supporto*. Tale definizione deriva dal fatto che non sono facilmente collocabili all'interno del nostro testo ma ne sono parte integrante e costituiscono una porzione fondamentale per la comprensione del documento.
Inizialmente parleremo del metodo con cui viene gestita la coda di riassemblaggio (ved. Sezione 6.6). Successivamente ci concentreremo poi la nostra attenzione sull'unica porzione della teoria non ancora analizzata: la scadenza di un timer di ritrasmissione.

8.2 Macro `TCP_REASS` e funzione `tcp_reass`

Come abbiamo visto, i segmenti TCP possono arrivare *out-of-order* ed è compito di TCP riordinare i segmenti per poterli passare al processo. Vediamo un esempio: supponiamo che A, un'entità TCP ricevente, comunichi a B, entità TCP mittente, una finestra di 4096 con 0 come prossimo numero di sequenza atteso. A questo punto A riceve il segmento con i byte 0 – 1023 (segmento in ordine) seguiti dai byte 2048 – 3071 (segmento non in ordine). A non può scartare i dati ricevuti *out-of-order* se sono contenuti all'interno della finestra di ricezione. Procederà quindi a porli in una lista di riassemblaggio per la connessione, in attesa che il segmento mancante arrivi (con i byte 1024 – 2047). Una volta ricevuto il segmento mancante procederà a riscontrare i byte 1024 – 3071. In questo paragrafo andiamo andiamo a esaminare il codice che manipola la coda di riassemblaggio di TCP.
Quando i dati vengono ricevuti da `tcp_input`, la macro `TCP_REASS`, viene invocata per sistemare i dati nella coda di riassemblaggio.

tcp_input.c
```
 53 #define  TCP_REASS(tp, ti, m, so, flags) { \
 54    if ((ti)->ti_seq == (tp)->rcv_nxt && \
 55       (tp)->seg_next == (struct tcpiphdr *)(tp) && \
 56       (tp)->t_state == TCPS_ESTABLISHED) { \

 58       (tp)->rcv_nxt += (ti)->ti_len; \

 62       sbappend(&(so)->so_rcv, (m)); \
 63       sorwakeup(so); \
 64    } else { \
 65       (flags) = tcp_reass((tp),(ti),(m)); \

 67    } \
 68 }
```
tcp_input.c

`tp` è un puntatore al blocco di controllo TCP relativo alla connessione mentre `ti` è un puntatore alla struttura `tcpiphdr` del segmento ricevuto.

Se le seguenti tre condizioni sono vere:

- il segmento è stato ricevuto in ordine (il numero di sequenza è quello atteso), e
- la coda di riassemblaggio della connessione è vuota (`seg_next` punta a se stesso e non ad un mbuf), e
- lo stato della connessione è `ESTABLISHED`,

possiamo aggiornare il valore di `rcv_nxt`, aggiungere i dati ricevuti al buffer di ricezione e svegliare i processi in attesa di quest'ultimo (righe 54 – 63).
In alternativa, se una delle condizioni non è verificata viene invocata la funzione `tcp_reass` per aggiungere il segmento alla coda di riassemblaggio:

```
------------------------------------------------------------tcp_input.c
 69 int
 70 tcp_reass(tp, ti, m)
 71 struct tcpcb *tp;
 72 struct tcpiphdr *ti;
 73 struct mbuf *m;
 74 {
 75    struct tcpiphdr *q;
 76    struct socket *so = tp->t_inpcb->inp_socket;
 77    int   flags;

 84    /*
 85     * Find a segment that begins after this one does.
 86     */
 87    for (q = tp->seg_next; q != (struct tcpiphdr *) tp;
 88       q = (struct tcpiphdr *) q->ti_next)
 89       if (SEQ_GT(q->ti_seq, ti->ti_seq))
 90          break;
------------------------------------------------------------tcp_input.c
```

La procedura inizia cercando la posizione corretta del segmento all'interno della coda (righe 84 – 90).

```
------------------------------------------------------------tcp_input.c
 91    /*
 92     * If there is a preceiding segment, it may provide some of
 93     * our data already. If so, drop the data from the incoming
 94     * segment. If it provides all of our data, drop us.
 95     */
 96    if ((struct tcpiphdr *) q->ti_prev != (struct tcpiphdr *) tp) {
 97       int   i;
 98       q = (struct tcpiphdr *) q->ti_prev;
 99       /*conversion to int (in i) handles seq wraparound*/
100       i = q->ti_seq + q->ti_len - ti_ti->seq;
101       if (i > 0) {
102          if (i >= ti->ti_len) {

105             m_freem(m);
106             return (0);
107          }
108          m_adj(m, i);
109          ti->ti_len -= i;
110          ti->ti_seq += i;
111       }
112       q = (struct tcpiphdr *)(q->ti_next);
```

```
113    }

116    REASS_MBUF(ti) = m;    /*XXX*/
-------------------------------------------------------------tcp_input.c
```

Nel caso sia presente un segmento precedente a quello puntato da `q`, i dati contenuti in tale segmento potrebbero intersecare quelli contenuti nel nuovo segmento. Il puntatore `q` viene perciò spostato al segmento precedente nella lista e il numero di byte intersecati viene calcolato e posto in `i`. Se il valore calcolato è superiore a 0, c'è intersezione. A questo punto i dati intersecati vengono rimossi dal nuovo segmento (eliminando l'intero segmento se l'intersezione è completa) (righe 91 – 111).
L'indirizzo del mbuf `m` viene poi memorizzato all'interno dell'header per mezzo della macro `REASS_MBUF`[29] (riga 116).

```
-------------------------------------------------------------tcp_input.c
117    /*
118     * While we overlap succeeding segments trim them or,
119     * if they're completely covered, dequeue them.
120     */
121    while (q != (struct tcpiphdr *)tp) {
122       int   i = (ti->ti_seq + ti->ti_len) - q->ti_seq;
123       if (i <= 0)
124          break;
125       if (i < q->ti_len) {
126          q->ti_seq += i;
127          q->ti_len -= i;
128          m_adj(REASS_MBUF(q), i);
129          break;
130       }
131       q = (struct tcpiphdr *) q->ti_next;
132       m = REASS_MBUF((struct tcpiphdr *) q->ti_prev);
133       remque(q->ti_prev);
134       m_freem(m);
135    }
136    /*
137     * Stick new segment in its place
138     */
139    insque(ti, q->ti_prev);
-------------------------------------------------------------tcp_input.c
```

Così come era avvenuto per il nuovo segmento rispetto al suo predecessore, se quelli che diventeranno segmenti successivi al nuovo nella coda di riassemblaggio hanno dati intersecanti con il nuovo segmento, quei dati vanno rimossi (righe 117 – 135).
A questo punto abbiamo riservato lo spazio per il nostro nuovo segmento e dunque possiamo aggiungerlo alla coda.
Analizziamo, perciò, la parte finale di `tcp_reass`. Questa porzione si occupa di passare i dati al processo (se possibile).

```
-------------------------------------------------------------tcp_input.c
140  present:
141    /*
```

29 La macro `REASS_MBUF` è

```
#define REASS_MBUF(ti)  (*(struct mbuf **)&((ti)->ti_t))
```

```
142     * Present data to user, advancing rcv_nxt through
143     * completed sequence space.
144     */

147    ti = tp->seg_next;
148    if (ti == (struct tcpiphdr *)tp || ti->ti_seq != tp->rcv_nxt)
149       return (0);

152    do {
153       tp->rcv_nxt += ti->ti_len;

155       remque(ti);
156       m = REASS_MBUF(ti);
157       ti = (struct tcpiphdr *) ti->ti_next;
158       if (so->so_state & SS_CANTRCVMORE)
159          m_freem(m);
160       else
161          sbappend(&so->so_rcv, m);
162    }while(ti != (struct tcpiphdr *)tp &&
                ti->ti_seq == tp->rcv_nxt);
163    sorwakeup(so);
164    return (flags);
165 }
-----------------------------------------------------------tcp_input.c
```

`ti` parte dal primo segmento della lista. Se la lista è vuota, o se il numero di sequenza del primo segmento della lista (`ti->ti_seq`) è diverso dal numero di sequenza atteso (`rcv_nxt`), i dati non possono essere passati all'utente e quindi la funzione ritorna 0 (righe 147 – 149).
In caso contrario, il ciclo parte dal primo segmento della lista (che sappiamo essere in ordine) e aggiunge (*in append*) i dati al socket buffer di ricezione. Il ciclo si arresta quando la lista è vuota oppure il numero di sequenza del prossimo segmento è *out-of-order* (righe 152 – 162).
Quando tutti gli mbuf possibili sono stati inseriti nel socket buffer di ricezione, `sorwakeup` sveglia tutti i processi in attesa di ricevere dati dalla socket.

8.3 Funzione `tcp_timers`

Cosa accade quando scade un timer di ritrasmissione? In Net/3 tutti i timer sono gestiti all'interno della funzione `tcp_timers`:

```
-----------------------------------------------------------tcp_timer.c
120 struct tcpcb *
121 tcp_timers(tp, timer)
122 struct tcpcb *tp;
123 int   timer;
124 {
125    int   rexmt;
126    switch (timer) {

140          /*
141           * Rexsmission timer went off. Message has not
142           * been acked within rexmit interval. Back off
143           * to a longer rexmit interval and rexmit one segment.
144           */
145       case TCPT_REXMT:
```

```
157         tp->t_timer[TCPT_REXMT] = tp->t_rxtcur;

171         tp->snd_nxt = tp->snd_una;

176         /*
177          * Close the congestion window down to one segment
178          * (we'll open it by one segment for each ack we get).

183          *
184          * There are two phases to the opening: Initially we
185          * open by one mss each ack. This makes the window
186          * size increase exponentially with time. If the
187          * window is larger than the path can handle, this
188          * exponential growth results in dropped packet(s)
189          * almost immediately. To get more time between
190          * drops but still "push" the network to take advantage
191          * of improving conditions, we switch from exponential
192          * to linear window opening at some threshhold size.
193          * For a threshhold, we use half the current window
194          * size, truncated to a multiple of the mss.
195          *
196          * (the minimum cwnd that will give us exponential
197          * growth is 2 mss. We don't allow the threshhold
198          * to go below this.)
199          */
200         {
201            u_int win =
                  min(tp->snd_wnd,tp->snd_cwnd) / 2 / tp->t_maxseg;
202            if (win < 2)
203               win = 2;
204            tp->snd_cwnd = tp->t_maxseg;
205            tp->snd_ssthresh = win * tp->t_maxseg;
206            tp->t_dupacks = 0;
207         }
208         (void) tcp_output(tp);
209         break;

256    }
257    return (tp);
258 }
-------------------------------------------------------------tcp_timer.c
```

Quando viene rilevata la scadenza del timeout di ritrasmissione, tale timer viene riavviato (riga 157) in quanto andremo a ritrasmettere i dati che risultano andati persi. A tale scopo, inoltre, `snd_nxt` viene impostato a `snd_una` (riga 171).
A questo punto entra in gioco il controllo della congestione. Come specificato in [5], Net/3 prevede che, una volta scaduto il timer di ritrasmissione, la connessione debba ritornare in Slow Start e la relativa soglia deve essere aggiornata (righe 176 – 205). Inoltre il conteggio degli ACK duplicati deve ricominciare da 0 (riga 206).
Sistemate tutte le relative variabili del blocco di controllo TCP, la funzione procede a ripetere la spedizione invocando `tcp_output` (riga 208).

Parte IV – Conclusioni

9. Conclusioni

> *"We have come a long way. Nine chapters stuffed with code is a lot to negotiate. If you didn't assimilate all of it the first time through, don't worry—you weren't really expected to. Even the best of code takes time to absorb, and you seldom grasp all the implications until you try to use and modify the program. Much of what you learn about programming comes only from working with the code: reading, revising and rereading."* (Kernighan & Plauger, *Software Tools*, 1976).

9.0 Grafo di riepilogo delle funzioni presentate

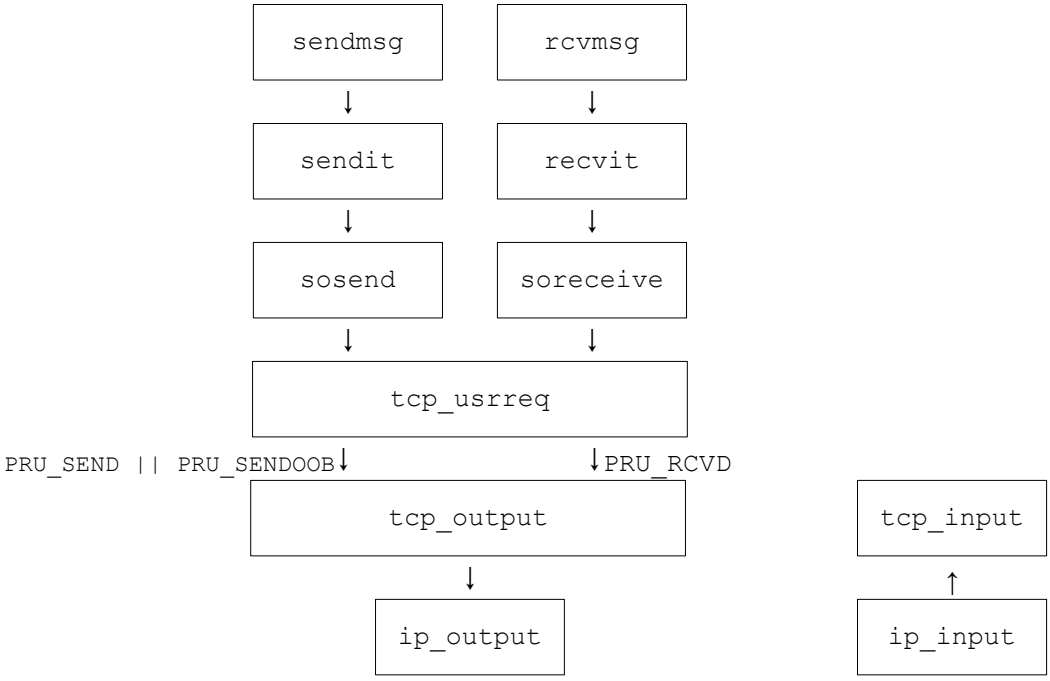


Figura 9.1
Relazione tra le funzioni presentate nel testo

9.1 Buffer di ricezione e di spedizione

Nella *Sezione 2.2* avevamo mostrato come la rappresentazione teorica dei buffer non suggerisse quasi niente a livello implementativo. Tale rappresentazione, inoltre, poteva essere fuorviante: limitandoci a considerare i buffer come una finestra di memoria a dimensione fissa poteva sembrare adeguato l'utilizzo di un array (eventualmente circolare). Ma quale potesse essere il contenuto di una singola cella era a dir poco "nebuloso".
Andando avanti con lo sviluppo del testo ci siamo resi conto che lo sviluppo della suite TCP/IP necessitava di poter manipolare una struttura dati dinamica

(per l'aggiunta e la rimozione di dati). Abbiamo, quindi, mostrato come Net/3 implementi tutti i meccanismi riguardanti i buffer di spedizione e ricezione utilizzando una struttura dati dinamica *ad-hoc*: la catena di mbuf.
Rimpiazzare la finestra a dimensione fissa con una singola catena dinamica risulta una buona soluzione per il problema del buffer di spedizione. Lo stesso, però, non si può dire per quanto riguarda la ricezione. L'implementazione effettiva di TCP, a differenza delle semplificazioni teoriche, deve tenere conto di una eventualità propria delle connessioni di rete: l'arrivo dei segmenti in maniera disordinata. Se scartassimo ogni segmento ricevuto avente numero di sequenza superiore a quello atteso, l'implementazione risulterebbe profondamente inefficiente. Abbiamo perciò mostrato come preveda di fornire, parallelamente alla catena che rappresenta il buffer di ricezione, una seconda catena di mbuf utilizzata come cache per i segmenti *out-of-order.*

9.2 Determinazione della finestra di ricezione

Studiando il testo [1] apprenderemmo che la finestra di ricezione (*RcvWin*) viene impostata alla quantità di spazio disponibile nel buffer di ricezione:

$$RcvWin = RcvBuffer - [highestByteRcvd - lastByteRead]$$

dove:

- *RcvBuffer* → dimensione massima del buffer di ricezione;
- *highestByteRcvd* → numero di sequenza successivo al più recente byte ricevuto e riscontrato;
- *lastByteRead* → numero di sequenza del byte più vecchio ricevuto e riscontrato.

Anche in questo caso, però, non teniamo conto dell'eventualità di arrivo di segmenti in maniera disordinata. Vediamo un esempio[30]: supponiamo che A e B siano i capi di una connessione TCP e che B comunichi ad A di avere una finestra di ricezione di 3 MSS e di essere in attesa del byte X. Supponendo che il segmento X sia "in volo", A, se il controllo di congestione lo permette, spedisce a B i due segmenti aventi numero di sequenza X+1 MSS e X+2 MSS. Supponiamo che attraversando la rete il segmento X+1 MSS vada perso e che X impieghi più tempo di X+2 MSS per giungere a destinazione. A questo punto abbiamo due situazioni diverse:

- *APPROCCIO TEORICO* → Il segmento X+2 MSS è stato scartato e quindi la nostra finestra di ricezione deve essere decrementata di un solo MSS (quello relativo al segmento X). A, quindi, riceverà da B un riscontro in cui *RcvWin* avrà valore di 2 MSS.

30 Per semplicità di esposizione asseriamo che:
- ogni segmento scambiato abbia dimensione 1 MSS;
- *CongWin* è sempre maggiore o uguale di 3 MSS;
- il processo in B non intervenga a consumare i dati presenti nel receive buffer nel lasso di tempo del nostro esempio.

- *APPROCCIO IMPLEMENTATIVO* → Il segmento X+2 MSS è contenuto nel buffer di ricezione e dunque deve essere tenuto di conto quando andiamo a calcolare lo spazio rimanente. Il riscontro inviato da B ad A avrà quindi il campo *RcvWin* settato a 1 MSS.

Dobbiamo inoltre notare come, a differenza di come presentato in teoria, l'aggiornamento del valore *RcvWin* comunicato da destinatario a mittente avvenga solamente se la differenza con il valore precedentemente comunicato è sostanziale (≥ 2 MSS).
Infine mostriamo nuovamente che Net/3 prevede di imporre un limite superiore al valore assumibile dalla finestra di ricezione imponendo che sia uguale a:

```
min(win, (long) TCP_MAXWIN << tp->rcv_scale)
```

La differenza tra il controllo di flusso presentato in [1] e quello codificato in Net/3 sta proprio nel modo in cui il destinatario calcola la finestra di ricezione da comunicare alla controparte. Il lato mittente, invece, si limita a implementare il controllo di flusso nella maniere più classica, ovvero considerando *RcvWin* come un limite superiore per l'invio di nuovi dati.

9.3 Aggiornamento del valore della finestra di congestione

Se ci limitassimo a considerare la tabella riassuntiva dell'algoritmo di controllo di congestione o andassimo ad analizzare direttamente [5] sembrerebbe necessario implementare direttamente una funzione del tipo:

```
CC_algorithm(...)
```

In realtà, la differenza che subito salta agli occhi è che l'implementazione di Net/3 "spalma" le funzionalità di controllo di congestione su varie funzioni (`tcp_output` e `tcp_input` su tutte). La motivazione per cui ciò avviene è che l'algoritmo di calcolo del valore di *CongWin* è facilmente rappresentabile come un automa a stati finiti in cui le transizioni sono dipendenti dagli eventi registrati da TCP. In particolare, quando dobbiamo effettuare calcoli relativi alla spedizione poniamo la porzione interessata dell'algoritmo all'interno di `tcp_output`. Viceversa, se l'evento scatenante è un evento di input, la porzione interessata andrà in `tcp_input`. O ancora, se l'evento è un timeout, la porzione di algoritmo deve andare in `tcp_timers`.
Notiamo inoltre come non venga esplicitata una variabile del tipo

```
int state
```

su cui andare a fare una serie di controlli. La scelta di distribuire il codice ci permette una rappresentazione implicita dello stato: a seconda della porzione in cui ci troviamo e del valore delle variabili relative riusciamo a capire quale sia l'azione da compiere. Un esempio che chiarisce bene questo tipo di implementazione è quello relativo al calcolo dell'incremento della variabile *CongWin* (*Sezione 7.5*).
Al di là, comunque, di questa scelta implementativa notiamo come l'unica

differenza dall'algoritmo classico per il calcolo di *CongWin* sia relativa ad una assunzione molto forte: nella teoria noi assumiamo di poter ricevere, ad ogni RTT, un numero di riscontri pari alla quantità di segmenti spediti. Questo comporta che, poiché in Congestion Avoidance vogliamo che l'incremento massimo sia di 1 MSS per ogni RTT, facciamo sì che ad ogni riscontro:

$$CongWin \mathrel{+}= MSS * MSS / CongWin$$

ovvero incrementiamo la finestra di congestione di una frazione di MSS relativa alla stessa finestra. Nella realtà questa situazione è poco probabile principalmente a causa del meccanismo di *delayed ACK* e della perdita di segmenti lungo la rete. Net/3, quindi, cerca di emulare il più possibile la situazione ottimale e, perciò, all'incremento classico aggiunge una frazione costante di MSS:

```
-------------------------------------------------------------tcp_input.c
 938          u_int incr = tp->t_maxseg;
 939          if (cw > tp->snd_ssthresh)
 940             incr = incr * incr / cw + incr / 8;
-------------------------------------------------------------tcp_input.c
```

9.4 Formula che combina *controllo di flusso* e *controllo di congestione*

Ricordiamo che nella *Sezione 2.4* avevamo mostrato come, al momento di andare ad inviare nuovi dati, il mittente TCP dovesse verificare che valesse:

$$nextSeqNum - sendBase \leq min(CongWin, RcvWin)$$

ovvero che la quantità di dati inviati ma non ancora riscontrati non ecceda il minimo tra i valori delle finestre di congestione e ricezione.
In Net/3 questo meccanismo è implementato così come è previsto da TCP [5]. Riguardando il codice riportato nella *Sezione 6.3*, infatti, notiamo come, dopo aver calcolato la quantità `off` di dati già presente nel buffer di spedizione (che corrisponde all'utilizzo della differenza tra *nextSeqNum* e *sendBase* all'interno della formula sopra riportata), si proceda al calcolo della finestra per mezzo dell'istruzione:

```
win = min(tp->snd_wnd, tp->cwnd);
```

A questo punto Net/3 è in grado di calcolare la quantità di dati che può spedire (in base alla quantità di dati presenti nel buffer e alla finestra calcolata) tenendo conto dei dati già spediti e non ancora riscontrati:

```
len = min(so->so_snd.sb_cc, win) - off
```

In base all'entità di `len` decide poi se sia *conveniente* spedire oppure no. Notiamo, quindi, come `win` sia usato esattamente come richiesto da TCP (limite superiore per il calcolo della quantità di dati da spedire).

9.5 Altri meccanismi introdotti da Net/3

Attraversando il codice BSD che implementa la suite TCP siamo venuti a contatto con due ulteriori meccanismi non menzionati nel testo [1]: il controllo di flusso dei socket e la prevenzione di *race condition*.

Per mezzo dei limiti di *high-water* e *low-water*, infatti, Net/3 si preoccupa di non inserire una quantità di dati eccessiva nel buffer di ricezione. Il principio è analogo a quello del controllo di flusso di TCP: se il processo inserisse dati nel buffer di ricezione con una frequenza superiore a quella con cui TCP procede a inviarli sulla rete, il buffer si esaurirebbe (`sb_cc` raggiungerebbe `sb_max`). A questo proposito, quando debba andare ad aggiungere dati al socket buffer di spedizione, Net/3 verifica che ci sia spazio a sufficienza per mezzo della seguente condizione:

$$0 \leq \texttt{sb\_cc} \leq \texttt{sb\_hiwat}$$

Inoltre, poiché sulla stessa macchina possono essere in esecuzione più processi che operano con la connessione TCP, Net/3 si occupa di evitare che si vengano a creare interferenze tra le varie istanze. Ogni qualvolta, infatti, debba essere eseguita un'operazione critica (aggiunta/rimozione di dati) sui buffer di ricezione, vengono utilizzate le funzioni `sblock` e `sbunlock` per evitare ogni *race condition*.

Entrambi questi meccanismi di *ensuring* della connessione, previsti da Net/3, non sono esposti nel testo [1] in quanto sono propri dell'implementazione del sistema operativo e non sono specifiche di TCP [3].

10. Bibliografia e riferimenti

Concetti di base

[1] J. Kurose, K. Ross, Reti di calcolatori e Internet: un approccio top-down, Pearson (IV edizione)

[2] W. Richard Stevens, Gary R: Wright, *TCP/IP Illustrated, Vol. 1: The Protocol*, Addison Weasley Professional

[3] Information Sciences Institute, University of Southern California, *RFC 793: Transmission Control Protocol*, September 1981

[4] David D. Clark, *RFC 813: Window and acknowledgement strategy in TCP*, July 1982

[5] M. Allman, V. Paxon, E. Blanton, *RFC 5681: TCP Congestion Control*,September 2009

[6] B. Moraru, F. Copaciu, G. Lazar. V. Dobrota, *Practical analysis of TCP Implementations: Tahoe, Reno, NewReno*

Implementazione

[7] W. Richard Stevens, Gary R: Wright, *TCP/IP Illustrated, Vol. 2: The Implementation*, Addison Weasley Professional

[8] Douglas E. Comer, David L. Stevens, *Internetworking with TCP/IP, Volume II: Design, Implementation, and Internals*, Prentice Hall - Second Edition

[9] http://www.leidinger.net/FreeBSD/dox/netinet/html/files.html, *FreeBSD kernel IPv4 Code*

Linguaggio C e programmazione in UNIX

[10] Brian W. Kernighan, Dennis M. Ritchie, *Il linguaggio C*, Pearson – Prentice Hall

[11] Marc J. Rochkind, *Advanced Unix Programming*, Prentice-Hall Software Series

Dichiarazione di Copyright BSD

Tutto il codice sorgente presentato in questo documento proviene dalla distribuzione 4.4BSD-Lite. Questo software è liberamente accessibile (come riportato nella bibliografia) e contiene la seguente dichiarazione di copyright:

```
/*
 * Copyright (c) 1982, 1986, 1988, 1990, 1993, 1994
 *      The Regents of the University of California.  All rights
 *      reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above
 *    copyright notice, this list of conditions and the following
 *    disclaimer in the documentation and/or other materials provided
 *    with the distribution.
 * 3. All advertising materials mentioning features or use of this
 *    software must display the following acknowledgement:
 *      This product includes software developed by the University of
 *      California, Berkeley and its contributors.
 * 4. Neither the name of the University nor the names of its
 *    contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ''AS IS''
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 * TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 * PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS
 * OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) SUBSTITUTE GOODS HOWEVER CAUSED
 * AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
 * ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
 * POSSIBILITY OF SUCH DAMAGE.
 */
```