A.FICARRA

Una Classe C++ per la comunicazione fra processi in rete

IRA 265/98

INDICE

APPENDICI

Introduzione

Il modello predominante per le comunicazioni fra processi che girano su computer collegati in rete é il ben noto modello "*client-server*". Il processo *server* e il processo *client* (che possono anche risiedere nello stesso computer, ma che più spesso girano su computer differenti) operano separatamente l'uno dall'altro, ma sono anche in grado di interagire e di comunicare fra loro. Tuttavia i due processi non sono "simmetrici", avendo compiti ben distinti, sia per quello che riguarda i rispettivi "tempi" di attivazione del collegamento, sia per il differente ruolo svolto a collegamento avvenuto. Infatti:

- il *server* si attiva per primo, identificandosi con un codice numerico (numero di *port*) che informa della sua esistenza il computer che lo ospita (*host*);
- il *server* esegue eventuali operazioni di inizializzazione e poi si pone in stato di attesa;
- da un qualunque computer della rete a cui é collegato l'*host*, il *client* fa richiesta di connessione al *server*, specificando il numero di *port* e l'indirizzo di rete dell'*host*;
- se la richiesta é accettata, entrambi i processi generano un *socket*, cioè un "punto finale di comunicazione" (secondo la definizione del sistema Unix di Berkeley); tramite questo *socket*, identificato da un numero, ogni processo vede l'altro come il punto di arrivo (o di partenza) dei dati trasmessi (o ricevuti);
- una volta stabilita la connessione, il *server* torna in stato di attesa, in quanto il meccanismo di interazione fra i due processi prevede che sia sempre il *client* a "fare la prima mossa", inviando una richiesta al *server* (a parte casi particolari di messaggi non sollecitati (*unsolicited*) inviati dal *server* a tutti i *clients* connessi);
- il *client* invia al proprio *socket* (il *server*) la richiesta di un "servizio"; il *server* esegue il servizio e, se necessario, invia la risposta al proprio *socket* (il *client*);
- ad uno stesso *server* possono connettersi più *clients*, ma questo fatto non crea ambiguità, in quanto il *server* é in grado ogni volta di identificare il *socket* da cui proviene la richiesta e a cui inviare la risposta.

Nei casi reali la situazione é talvolta più complessa di quella descritta dal modello: infatti, in un ambiente di informatica distribuita, un'organizzazione efficiente potrebbe, per evitare inutili duplicazioni, assegnare un ruolo specifico ad ogni computer della rete (per esempio per l'acquisizione di dati da strumenti, come realmente accade alla stazione radioastronomica di Medicina), e riservare ai processi di comunicazione il compito di scambiare fra i computer le informazioni necessarie. In una tale organizzazione non si può più parlare di semplice modello *client-server*, ma piuttosto di interscambio di dati fra diverse applicazioni; per cui, una stessa applicazione potrebbe trovarsi a gestire uno più processi *server* e, contemporaneamente, essere *client* di altri *servers* della rete.

In ambiente **Visual C++**, la libreria **Microsoft Fundation Class** (MFC) mette a disposizione due classi per la gestione dei processi di comunicazione in rete:

- la classe **CAsyncSocket** incapsula in oggetti le funzioni *socket* della libreria **API** di **Windows** (ciascun oggetto é un *socket*), senza peraltro fornire funzionalità aggiuntive; le operazioni di connessione, invio e ricezione sono eseguite in modo *asincrono*, cioé con il semplice lancio dell'operazione, senza attesa del suo completa-

mento, che verrà invece notificato come "*evento*" in un messaggio di **Windows**; é compito dell'applicazione implementare le funzioni di *callback* per la gestione degli *eventi*.

- la classe **CSocket**, derivata da **CAsyncSocket**, fornisce un livello di astrazione più alto della sua classe base, e si incarica di gestire molti aspetti della comunicazione, che sarebbero invece demandati all'applicazione nel caso di uso diretto dell'**API** o anche della sola classe **CAsyncSocket**; d'altra parte, usando **CSocket**, le operazioni possono essere eseguite solo in modo *sincrono* (il flusso del programma si arresta e non riprende finché l'operazione non é completata).

Sono pertanto sorte due esigenze: quella generale di disporre di strumenti software adeguati, per gestire in modo semplice situazioni complesse (per esempio, reti di elaboratori collegati a una grande varietà di strumenti); e quella specifica di una classe di comunicazione su *socket* che riunisca i vantaggi di un alto livello di astrazione (come **CSocket**) a quelli dell'esecuzione di operazioni asincrone (o, meglio, gestibili con *time-out*) che la classe **CSocket** non consente.

L'autore del presente Rapporto ha progettato e implementato una **Dynamic Link Library** (*dll*) che contiene alcune classi per la gestione delle comunicazioni su *socket*; queste classi possono creare un numero qualsivoglia di processi *server* e/o *client* nella stessa applicazione; ogni processo utilizza le funzionalità di **CSocket** e dispone di un proprio buffer per la memorizzazione dei dati in partenza e in arrivo; ogni processo gira in un *thread* separato e quindi permette al programma di accedere ai dati, di fatto, in modo asincrono. Verso l'applicazione collegata, la *dll* "esporta" una sola classe, di nome **CSockEnv** ("inviluppo" di *socket*), che consente di eseguire in modo semplice tutte le operazioni sopradescritte; in particolare, in ogni processo *client*, il corrispondente *socket* viene visto dall'applicazione come un semplice "file", in cui l'operazione di connessione al *server* diventa una funzione di **Open** (con restituzione del numero di *socket* come identificatore del "file") e le operazioni di invio e ricezione di dati diventano rispettivamente funzioni di **Write** e di **Read**.

Questo Rapporto, rivolto ai programmatori in Visual C++, descrive soprattutto l'uso di **CSockEnv** da parte dell'applicazione collegata alla *dll*; per gli interessati contiene anche la descrizione della struttura, proprietà e metodi delle classi "nascoste" nella libreria.

Per concludere, osserviamo che la **CSockEnv** genera processi *client* e *server* che non sono vincolati a protocolli specifici (salvo l'obbligo di comunicare su reti TCP/IP) e quindi le applicazioni che la utilizzano, pur dovendo essere sviluppate in Visual C++, possono, senza alcuna restrizione, collegarsi a processi *client* e *server* che girano su piattaforme diverse (per esempio, a Medicina, un'applicazione *server* che usa **CSockEnv**, é accessibile da un *client* generato dal *Field-System*, che, come é noto, gira in ambiente **Linux**).

Caratteristiche generali della libreria

La **Dynamic Link Library**, presentata in questo Rapporto, si chiama **SockEnv.dll** ed occupa circa 38 KB. Può essere utilizzata da un programma eseguibile (**.exe**) che si collega dinamicamente alla libreria MFC.

La libreria é una *dll* di "*estensione*" della MFC; ricordiamo che le *dll* si dividono in due categorie: quelle "*regolari*", che non *esportano* classi C++ (pur potendole contenere al loro interno), ma solo funzioni C, e possono essere *linkate* da programmi scritti anche in altri linguaggi (per esempio in Visual Basic); e quelle di "*estensione*", che *esportano* classi C++, ma possono essere *linkate* solo da programmi in Visual C++ che utilizzano la libreria MFC. Nel nostro caso la classe *esportata* dalla *dll* si chiama **CSockEnv** (derivata dalla classe **CFrameWnd** della MFC), e quindi le funzioni-membro di *accesso pubblico* di **CSockEnv** sono gli unici canali di comunicazione fra il programma esterno e la libreria.

Nell'applicazione collegata alla *dll*, la classe **CSockEnv** deve essere *istanziata* una volta sola, all'inizio (come vedremo in dettaglio più avanti, bisogna utilizzare una *macro*, definita nella stessa *dll*, che crea l'oggetto di **CSockEnv** e ne restituisce l'indirizzo; questo sarà in seguito usato come puntatore alle funzioni-membro di **CSockEnv** chiamate dall'applicazione); in questa fase bisogna anche specificare se l'applicazione potrà generare solo processi *server*, o solo processi *client*, o entrambi.

Tramite la **SockEnv.dll**, l'applicazione collegata può (per i dettagli delle funzioni, vedere Appendice B):

a) nella funzionalità *server*

- creare un numero qualsivoglia di processi *server*, che, ovviamente, stazioneranno nello stesso computer (*host*) in cui gira l'applicazione; ogni *server* é identificato da un numero specifico (*port number*) e, uguale per tutti, dall'indirizzo *Internet* dell'*host*; per creare un processo *server* bisogna chiamare la funzione **ActivateServer** con il *port number* come argomento; non possono essere attivi contemporaneamente due *servers* con lo stesso *port number* nello stesso *host*, anche se generati da due applicazioni diverse (la *dll* se ne accorge e restituisce un messaggio di errore);
- fornire (obbligatoriamente) una funzione (con nome di fantasia, ma convenzionalmente d'ora in poi la chiameremo **Service**), che sarà chiamata dalla *dll* ogni volta che un *client* avrà fatto pervenire una richiesta; la funzione ha il compito, per ogni *server* attivato, di eseguire il "servizio" richiesto dal *client* e di restituire alla *dll* l'eventuale risposta, che la stessa *dll* provvederà a trasmettere al *client*; gli argomenti di chiamata della **Service** saranno descritti più avanti;
- visualizzare la lista dei *servers* attivi (funzione **ShowServers**);
- visualizzare una serie di informazioni (funzione **ShowConnected**) sui *clients* "remoti" connessi ai *server* attivi (attenzione: per comodità indicheremo sempre, con il termine "remoto", il processo della "controparte", anche se non é escluso che un processo possa essere *client* di un *server* che gira nello stesso computer o addirittura nella stessa applicazione);

- ottenere un *array* con gli indirizzi *Internet* dei *clients* connessi a un certo *server* (funzione **GetClientsArray**);
- spedire un messaggio *unsolicited* a tutti i *clients* connessi a un certo *server* (funzione **SendUnsolicited**);
- disattivare un *server* (funzione **CloseServer**); comunque, al termine del programma, tutti i *servers* ancora attivi sono disattivati automaticamente.

b) nella funzionalità *client*

- connettersi a un numero qualsivoglia di processi *server* remoti; per iniziare la connessione a un *server*, bisogna chiamare la funzione **Open**, con il numero di *port* e l'indirizzo *host* del *server* come argomenti; se l'operazione va a buon fine, la **Open** restituisce un numero (*handle*) che identifica il *socket* del *server* a cui l'applicazione si é connessa come *client*; l'*handle* dovrà essere usato in tutte le successive operazioni di trasmissione e ricezione di dati;
- inviare dati a un *server* (funzione **Write**, con due possibili *overload*, rispettivamente per i dati binari e per le stringhe di caratteri); da parte dell'applicazione l'operazione é *asincrona* e restituisce una valore *booleano* che ne indica esclusivamente l'ammissibilità (per esempio se l'*handle* specificato corrisponde realmente a un *socket*), ma l'invio effettivo dei dati avviene in modo *sincrono* in quanto é eseguito da un *thread* interno della *dll*;
- leggere i dati ricevuti da un *server*, con (funzione **Read**) o senza (funzione **Test**) rimozione degli stessi dal buffer di input; entrambe le funzioni sono *sincrone*, ma provviste di *time-out*, il cui valore può essere specificato fra gli argomenti (*defaults*: infinito nella **Read**, zero nella **Test**; notare che, se il valore é zero, l'operazione diventa *asincrona*); la funzione **Test** può essere usata anche per controllare la presenza di dati nel buffer; infine la funzione **RemoveFromBuffer** può servire per rimuovere, tutti o in parte, i dati dal buffer;
- comunicare a un *server* l'identificatore del messaggio di *Windows* in cui ricevere gli eventuali dati *unsolicited* (funzione **ReceiveUnsolicited**); questi, come conseguenza, verranno scorporati dal buffer di input e messi in un altro buffer, prima di essere inviati all'applicazione; con la funzione **ClearUnsolicited** é possibile azzerare il buffer dei dati *unsolicited*;
- ottenere il numero di *port* del *server* corrispondente a un certo *handle* (funzione **GetClientPort**);
- visualizzare la lista (funzione **ShowClients**) di tutti i *clients* dell'applicazione connessi ai *servers* remoti;
- chiudere una connessione da parte *client* (funzione **Close**); comunque, al termine del programma, tutti i *clients* ancora connessi sono disconnessi automaticamente.

c) nella gestione degli errori

- ottenere una stringa con il messaggio di errore, nel caso che l'ultima operazione non sia andata a buon fine (funzione **GetErrorText**).

La *dll* dichiara e implementa quattro classi (oltre a **CSockEnv**), a cui l'applicazione non accede direttamente, ma che sono necessarie per operare in *threads* separati e gestire le comunicazioni fra i *threads*; queste classi derivano tutte, per ereditarietà, da classi della MFC, e sono:

- **CSocketThread**, derivata da **CWinThread**;

- **CServerSock**, derivata da **CSocket**;
- **CClientSock**, derivata da **CSocket**;
- **CWinSock**, derivata da **CFrameWnd**;

Nel momento in cui l'applicazione crea un nuovo *server* (**ActivateServer**) o si connette a un *server* come *client* (**Open**), la *dll* genera un *thread*, creando un'istanza della classe **CSocketThread**; nel primo caso la funzione **InitInstance** di **CSocketThread** (che inizializza il *thread*) crea un oggetto della classe **CServerSock**, il quale non fa altro che porsi in attesa delle richieste di connessione di eventuali *clients* remoti; nel secondo caso la **InitInstance** crea direttamente un *socket*, come oggetto della classe **CClientSock**, dove si trovano tutte le funzionalità per la bufferizzazione, l'invio e la ricezione dei dati sulla rete. Infine, anche la classe **CWinSock** é *istanziata* dalla **InitInstance** e serve, come vedremo fra poco, per gestire le comunicazioni (via *messaggi* di *Windows*) fra il *thread* dell'applicazione e quelli dei *sockets*.
Se un *server* attivato dall'applicazione riceve una richiesta di connessione da un *client*, la funzione **OnAccept** di **CServerSock** crea un *socket*, sempre come oggetto di **CClientSock**, ma con la differenza che, questa volta, il *server* é locale e il *client* é remoto.
In sostanza a ogni *socket* corrisponde sempre un oggetto della classe **CClientSock**, sia nella funzionalità *server* che in quella *client*; mentre, però, come *client* locale ogni *socket* gira in un *thread* separato (in quanto creato direttamente da **CSocketThread**), tutti i *sockets* di *clients* remoti connessi a uno stesso *server* girano nello stesso *thread*; questo non é un problema, in quanto la **CSocketThread** implementa una funzione di accodamento che permette di servire in successione separata le diverse richieste che dovessero pervenire a un *server* contemporaneamente.

Nella trattazione che segue useremo spesso alcuni termini, a cui attribuiamo convenzionalmente il seguente significato:

- con il termine *application*, intendiamo il codice del programma esterno, che si collega a **SockEnv.dll**;
- con il termine *dll* intendiamo il codice del programma che si trova in **SockEnv.dll**;
- con il termine *main* intendiamo l'esecuzione del *thread* principale, cioè quello creato appena si lancia l'applicazione;
- con il termine *thread* intendiamo l'esecuzione di un generico *thread* creato da **CSocketThread**.

E' bene osservare che non sempre l'*application* gira nel *main* o la *dll* gira nel *thread*, anche se la maggior parte delle volte é così; infatti, per esempio:

- la funzione **Service** si trova nell'*application*, ma gira nel *thread*, in quanto é chiamata da varie funzioni di **CSocketThread** (**ProcessReceive**, **ProcessClose**, **ExitInstance**);
- tutte le funzioni di *accesso pubblico* di **CSockEnv** (**ActivateServer** ecc...) si trovano nella *dll*, ma girano nel *main*, in quanto sono chiamate dall'applicazione.

La comunicazione fra *main* e *thread* é gestita dalle classi **CSockEnv** (parte *main*) e **CWinSock** (parte *thread*); entrambe queste classi derivano da **CWnd** (tramite **CFrameWnd**) e quindi sono in grado di ricevere *messaggi* di *Windows* che attraversano i *threads*.

FUNZIONE DI SERVIZIO FORNITA DALL'APPLICAZIONE

Abbiamo detto (pag.3) che l'*application*, se utilizza la modalità *server*, deve fornire una particolare funzione (che, per comodità, continuiamo a chiamare **Service**). Tale funzione deve essere dichiarata come membro della classe **CMainFrame**, e il suo nome deve comparire come parametro nella stessa *macro* che crea, all'inizio, l'unica istanza della classe **CSockEnv** (vedere Appendice A). Pur trovandosi nell'*application*, la **Service** non ha il compito di girare nel *main* (cioè non deve essere chiamata direttamente dall'applicazione), ma nel *thread*, in quanto può essere eseguita da tre funzioni di **CSocketThread** (le chiamate, nella *dll*, sono del tipo "puntatore a funzione" e agganciano la funzione reale in fase di esecuzione (*late binding*), utilizzando il nome che é stato comunicato alla *dll* tramite la *macro* di cui sopra):

1. **ProcessReceive** (caso generale) - Questa funzione, chiamata quando il *server* riceve dati da un *client* remoto, trasferisce i dati alla **Service** e, in base ai parametri di ritorno, decide:
   a) se spedire una risposta al *client*;
   b) quali dati rimuovere dal buffer di input;
   c) chiudere il *socket* o proseguire;
   d) eseguire o meno una nuova chiamata della **Service**.
   La gestione del buffer di input (possibilità di accumulare i dati per elaborarli insieme successivamente) e, all'opposto, la facoltà di chiamare la **Service** più volte in corrispondenza della stessa ricezione, costituiscono due caratteristiche peculiari della flessibilità di questo sistema, in quanto svincolano il momento della ricezione dei dati da quello della loro elaborazione, e in tal modo permettono di prevedere una vasta gamma di situazioni, al di là della semplice "domanda-risposta" (si possono generare più risposte a una stessa domanda o viceversa).
2. **ProcessClose** - Questa funzione, chiamata quando il *socket* é stato chiuso dalla parte *client*, chiama la **Service** per informare l'applicazione dell'avvenimento (e consentirle, se necessario, di intraprendere le azioni conseguenti).
3. **ExitInstance** - Quando il *thread* in cui gira il *server* é in procinto di terminare (per chiusura del *server* o dell'intera applicazione), viene chiamata la **Service** per spedire, se necessario, un messaggio al *client* (questa eventualità é molto rara, considerato il fatto che, da parte *client*, il messaggio é memorizzato in un buffer che viene cancellato alla chiusura del *socket* e quindi può essere ricevuto solo se il *client* é in lettura mentre il messaggio sta arrivando; alternativamente, se questa informazione é indispensabile, si potrebbe creare, nell'applicazione che ospita il *client*, un *thread* separato, che acceda continuamente al buffer di input senza rimuovere i dati, e invii un messaggio all'applicazione del caso che identifichi il messaggio di chiusura del *server*).

In sostanza, il sistema presenta la flessibilità e la versatilità necessarie per consentire all'applicazione di costruire la **Service** "su misura" per le esigenze specifiche. Come vedremo in dettaglio nella prossima descrizione dei parametri, la funzione ha la possibilità di scambiare con la *dll* una grande varietà di informazioni, e quindi di rispondere in modo adeguato a situazioni diverse nel processo di comunicazione fra *server* e *client*.

La funzione deve essere dichiarata nel seguente modo:

BOOL Service (const int nPort, int& status, const char* bufin, int& lenin, char*& bufout, int& lenout)

dove il significato dei parametri é il seguente:

nPort numero della *port* che identifica il *server*; nel caso che più *servers* siano attivi nello stesso tempo, il valore di questo parametro permette alla **Service** di riconoscere il *server* coinvolto nell'operazione;

status riferimento alla variabile-membro **m_status** del *socket*; il suo valore (inizializzato a zero dal *costruttore* del *socket*) é utilizzabile dalla **Service** (sia in ingresso che in uscita) per distinguere momenti e situazioni diverse del processo di comunicazione (può riconoscere, per esempio, se sta comunicando con il *socket* per la prima volta, oppure se si sono verificate o meno alcune condizioni ecc...); per le normali operazioni, deve essere **≥ 0**, in quanto i valori negativi sono riservati ai seguenti casi particolari:
status = -1 (ingresso) : il *server* sta chiudendo (chiamante: **ExitInstance**), e quindi, se é necessario, la **Service** deve preparare il messaggio di chiusura da inviare al *client* prima della cancellazione del *socket*;
status = -2 (ingresso) : il *socket* é stato chiuso dalla parte del *client* remoto (chiamante: **ProcessClose**); se necessario, la **Service** ne prenderà atto, operando di conseguenza (ovviamente non deve preparare nessun messaggio di risposta);
status = -1 (uscita) : (chiamante: **ProcessReceive**) la **Service** ha riconosciuto, nel messaggio inviato dal *client*, la richiesta di chiusura del *socket* (é prevista infatti la possibilità che alcuni *clients*, soprattutto se non generati da questa *dll*, non si chiudano da soli, ma richiedano che tale operazione venga fatta dal *server*); in questo caso, la **ProcessReceive**, invia al *client* l'eventuale messaggio di risposta e poi cancella il *socket*;
status = -2 (uscita) : (chiamante: **ProcessReceive**) la **Service** ha riconosciuto, nel messaggio inviato dal *client* remoto, la richiesta di spedirgli tutti i messaggi *unsolicited* che il *server* produrrà - inoltre i messaggi dovranno essere racchiusi fra parentesi graffe;
status = -3 (uscita) : (chiamante: **ProcessReceive**) come il precedente, salvo il fatto che i messaggi dovranno essere spediti inalterati;
status = -4 (uscita) : (chiamante: **ProcessReceive**) la **Service** ha riconosciuto, nel messaggio inviato dal *client*, la richiesta di non spedirgli più i messaggi *unsolicited*;
in tutti gli altri casi (**status** ≥ 0, chiamante: **ProcessReceive**), l'uso di questo parametro é libero e lasciato alle esigenze specifiche dell'applicazione; l'aggiornamento del valore in uscita di **sta-**

tus nella variabile **m_status** del *socket* é effettuato soltanto se risulta **status** ≥ 0;

bufin (solo in ingresso) indirizzo del buffer di input (dati trasmessi dal *client*); se la **Service** é chiamata dalla **ProcessClose** (**status=-2**), il buffer contiene una stringa (*null-terminated*) con il nome del computer che ospitava il *client* defunto; se la **Service** é chiamata dalla **ExitInstance** (**status=-1**), il parametro é privo di significato;

lenin in ingresso: lunghezza (in byte) del buffer di input; se la **Service** é chiamata dalla **ProcessClose** (**status=-2**), **lenin** contiene il numero di identificazione del *socket* (dalla parte *server*); se la **Service** é chiamata dalla **ExitInstance** (**status=-1**), il parametro é privo di significato;
in uscita: (significativo solo se la **Service** é chiamata dalla **ProcessReceive**) numero di bytes da rimuovere dal buffer di input (in particolare, se la **Service** lascia invariato il valore di **lenin**, tutto il contenuto del buffer verrà rimosso);

bufout (solo in uscita) indirizzo del buffer di output (risposta da inviare al *client*), oppure **NULL** in caso di nessuna risposta; é compito dell'applicazione provvedere all'allocazione di memoria per il buffer di output; inoltre, l'area indirizzata non può essere *dinamica* (cioè non può essere creata nello *stack* della **Service**), ma deve essere dichiarata *static*, o (preferibilmente) deve essere membro in una classe; il passaggio del parametro *by reference* garantisce che le impostazioni di **bufout** eseguite nella **Service** verranno trasferite nella funzione chiamante; se la **Service** é chiamata dalla **ExitInstance** (**status=-1**), in **bufout** deve essere trasferito (se esiste) l'indirizzo del messaggio di chiusura del *socket*; se la **Service** é chiamata dalla **ProcessClose** (**status=-2**), il parametro é privo di significato;

lenout (solo in uscita) lunghezza (in byte) del buffer di output, oppure **0** in caso di nessuna risposta da inviare al *client*; poiché, in ingresso, **lenout** potrebbe anche essere diverso da zero, é prudente che la **Service** lo riazzeri all'inizio (impostandolo poi al valore opportuno solo se é necessario inviare il messaggio di risposta al *client*); se la **Service** é chiamata dalla **ProcessClose** (**status=-2**), il parametro é privo di significato;

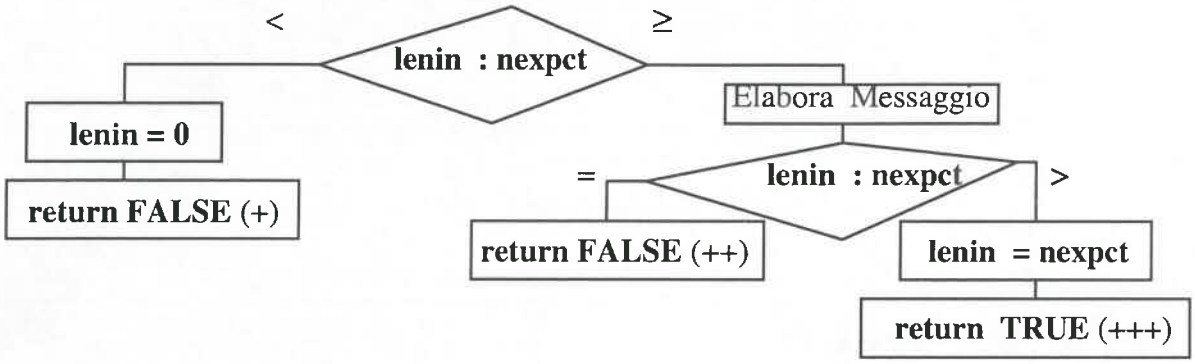
return (significativo solo se la **Service** é chiamata dalla **ProcessReceive**)
TRUE : informa la **ProcessReceive** che, dopo l'eventuale invio del messaggio di risposta, la **Service** deve essere chiamata nuovamente; questa opzione può essere utilizzata, per esempio, quando la lunghezza del buffer di input supera quella che la **Service** si aspetta di trovare (cioè sono arrivati altri dati nel frattempo e quindi ad un'unica ricezione devono corrispondere più risposte);

FALSE : la **Service** ha esaurito le risposte che competono al messaggio ricevuto (oppure il messaggio non é ancora completo) e quindi anche la **ProcessReceive** deve terminare (ovviamente, anche in questo caso può esserci o meno un messaggio di risposta) e sarà nuovamente chiamata alla prossima ricezione.

Un caso a parte é quello in cui la **Service** ritorna il parametro **status** con il valore di **-5**: la **ProcessReceive** aggiorna come al solito il buffer di input in base al valore restituito di **lenin**, ma non procede con la spedizione di dati al *socket*; invece chiama di nuovo la **Service**, trasmettendole, nei parametri **bufin** e **lenin**, il nome dell'*host* che ospita il *client* e il numero del *socket* (esattamente come la **ProcessClose**, con **status=-2**). La **Service**, riconoscendo il valore (in ingresso) di **status = -5**, può memorizzare le informazioni trasmesse in **bufin** e **lenin**; successivamente deve impostare il valore desiderato di **status** (diverso da **-5**, altrimenti si creerebbe un *loop* infinito) e ritornare alla **ProcessReceive** (in questo caso il valore di ritorno é ininfluente). Questa ripristina **bufin** e **lenin** al loro normale significato e chiama per la terza volta la **Service** che può procedere nell'elaborazione e restituire l'eventuale messaggio di risposta (in **bufout** e **lenout**) da inviare al *socket*. Questa "intromissione" nel processo normale di comunicazione fra *client* e *server* é stata prevista per consentire alla **Service** (e quindi all'*application* in cui gira il *server*) di memorizzare, e utilizzare all'occorrenza, i dati di identificazione dei vari *client* connessi (per esempio, l'applicazione può sapere quale *client* si é disconnesso, confrontando tali dati con quelli forniti dalla **ProcessClose**).
Vediamo ora due esempi significativi di utilizzo di questa opzione e il modo esatto di procedere, per evitare *loop* indesiderati o errori:

1) Un particolare *client* vuole notificare la propria presenza al *server* e a questo scopo invia uno specifico comando, che la **Service** riconosce: questa deve restituire alla **ProcessReceive** il valore **-5** in **status** e lasciare inalterato **lenin**, in modo che il buffer di input sia cancellato; nella seconda esecuzione, la **Service** riconoscendo **status=-5**, può memorizzare le informazioni di identificazione del *client*, che trova in **bufin e lenin**, e poi porre **status** a un qualsiasi valore non negativo; nella terza esecuzione la **Service**, trovando il buffer di input vuoto (**lenin=0**), non deve fare nulla e ritornare **FALSE** alla **ProcessReceive**.
2) Si può anche trarre informazioni su un *client* "a sua insaputa": a questo scopo si stabilisce che la variabile **m_status** del *socket* resti zero finché il *server* non riceve messaggi e assuma valori maggiori di zero dopo il primo messaggio ricevuto: quando un *client* manda il suo primo (normale) messaggio, la **Service**, riconoscendo **status=0**, pone **status=-5**, ma questa volta azzera il valore di **lenin**, per lasciare inalterato il buffer di input; nella seconda esecuzione memorizza le informazioni sul *client* e pone **status=1**; la terza esecuzione si svolge normalmente come se nulla fosse successo, in quanto la **Service** trova il messaggio inviato dal *client* immutato nel buffer di input, ma questa volta, essendo **status=1**, procede senza deviare dalla sequenza normale di operazioni.

Per concludere il discorso sulla **Service**, osserviamo che un tipo abbastanza frequente di interazione fra *client* e *server* si basa sul processo "domanda-risposta" ("*solicited message*"): il *client* invia al *server* un pacchetto di dati (che chiamiamo "messaggio") di lunghezza aspettata (non necessariamente costante, ma comunque desumibile dallo stesso pacchetto o dai precedenti - per esempio é molto frequente che i primi caratteri di un messaggio contengano l'informazione sulla lunghezza del messaggio stesso); indicando con **nexpct** tale lunghezza e, al solito, con **lenin** sia la lunghezza del buffer (in ingresso alla **Service**) che il numero di caratteri del buffer da cancellare (in ritorno dalla **Service**), non sempre **nexpct** e **lenin** coincidono: infatti un singolo messaggio può arrivare frazionato in più spedizioni oppure più messaggi possono arrivare in un'unica spedizione. Occorre perciò che la **Service** lavori in modo da prevedere tutte queste possibilità e ad ogni messaggio (non ad ogni ricezione!) faccia corrispondere le operazioni di elaborazione del messaggio e di invio dell'eventuale risposta. Il seguente diagramma a blocchi mostra una schema generale di comportamento della **Service** nelle diverse situazioni:



(+) Lascia inalterato il buffer di input e attende la prossima ricezione di dati dal *client* per il completamento del messaggio

(++) Ha elaborato il messaggio e non vi sono altri dati nel buffer (non modificando il valore di **lenin** provoca la cancellazione dell'intero contenuto del buffer)

(+++) Ha elaborato il messaggio, ma vi sono altri dati nel buffer, e quindi elimina dal buffer solo i dati relativi al messaggio e chiede che la **Service** venga chiamata di nuovo per l'elaborazione del messaggio successivo

Nel caso che la lunghezza del messaggio non sia un dato a priori, ma sia desumibile dal messaggio stesso, lo schema é un po' più complicato: in pratica occorre che la variabile **nexpct** non sia *volatile*, ma *statica* (o, meglio ancora, membro di una classe), per modo che il che il suo valore possa essere determinato durante una esecuzione della **Service** e utilizzato nella esecuzione successiva (infatti il messaggio deve essere spezzato in due parti: dalla prima, di lunghezza nota, si determina **nexpct**, e dalla seconda, di lunghezza variabile, si estraggono i dati).

MESSAGGI "UNSOLICITED"

Si dice "*unsolicited*" un messaggio (o in generale un qualsiasi pacchetto di dati) che il *server* spedisce ai *clients* "spontaneamente", senza cioè che i *clients* ne abbiano fatto richiesta. A questa categoria appartengono per esempio i messaggi di errore, gli avvisi generali ecc...
In ogni caso la scelta se ricevere o meno i messaggi *unsolicited* spetta a ogni singolo *client*, che deve segnalare al *server* quando inizia (ed eventualmente quando termina) la sua disponibilità. Le due parole-chiavi di inizio e fine sono lasciate alla libera scelta dell'applicazione, in quanto definite nella funzione **Service**, che, quando le riconosce, deve trasferire l'informazione alla *dll* mediante il parametro di uscita **status**, ponendolo **-2** (o **-3**) se il *client* ha chiesto di ricevere i messaggi *unsolicited* con (o senza) parentesi graffe in testa e in coda, **-4** se il *client* ha chiesto di non riceverli più (di *default* non li riceve).
Nella *dll* il trattamento dei messaggi *unsolicited* avviene in due parti distinte: la parte *server* e la parte *client*; mentre la prima é necessaria affinché i messaggi *unsolicited* siano ricevuti dai *clients* che ne hanno fatto richiesta, la seconda é opzionale: se viene utilizzata, ogni messaggio *unsolicited* é separato automaticamente dagli altri pacchetti inviati dal *server*, e poi spedito all'applicazione in un messaggio di *Windows*; in caso contrario la separazione non avviene e spetta all'applicazione distinguere i messaggi *unsolicited* dagli altri dati. La non obbligatorietà della parte *client* preserva la caratteristica di versatilità del sistema, che può funzionare anche quando uno dei due *socket* (in questo caso il *client*) non é generato dalla *dll*.

PARTE SERVER

1) E' giunta da un *client* la richiesta di ricevere i prossimi messaggi *unsolicited*;
2) La **Service**, fornita dall'applicazione, riconosce la richiesta e imposta il valore **-2** o **-3** nel parametro **status**;
3) Conseguentemente la **ProcessReceive** imposta il valore **2 o 1** nella variabile **m_Unsolicited** del *socket*;
4) Ogni volta che l'applicazione necessita da inviare un messaggio *unsolicited*, deve utilizzare la funzione **SendUnsolicited**, trasferendole, fra i parametri, il numero della *port* del *server* da selezionare e il contenuto del messaggio;
5) La **SendUnsolicited** (che gira del *main*) trasmette il messaggio alla funzione **ProcessUnsolicited** di **CSocketThread** (che gira nel *thread* del *server* selezionato);
6) La **ProcessUnsolicited** esamina tutti i *sockets* connessi, inviando il messaggio a quelli in cui il valore di **m_Unsolicited** é **> 0** (in particolare, se trova **m_Unsolicited = 2**, racchiude il messaggio fra parentesi graffe **);**
7) Se il *client* chiede di interrompere la ricezione dei messaggi *unsolicited*, la **Service**, riconoscendo tale richiesta, imposta il valore **-4** nel parametro **status**;
8) Conseguentemente la **ProcessReceive** ripristina il valore di *default* (**0**) della variabile **m_Unsolicited** del *socket*.

PARTE CLIENT

1) Se il *socket* é in modalità *client*, la variabile **m_Unsolicited** ha un diverso significato, e precisamente: se é **=0** (*default*), i messaggi *unsolicited* ricevuti devono esser memorizzati nel buffer generale di input del *socket*, insieme agli altri pacchetti inviati dal *server*, e spetta all'applicazione riconoscerli (se sono racchiusi fra parentesi graffe) ed estrarli, utilizzando le normali funzioni **Test** e **Read**; se invece é >0, **m_Unsolicited** rappresenta il numero di identificazione del messaggio di *Windows* a cui l'applicazione desidera che siano trasferiti tutti i messaggi *unsolicited* ricevuti dal *server*: ciò comporta automaticamente anche lo scorporo dei messaggi dal buffer di input del *socket* e la loro memorizzazione in un buffer separato (non accessibile dalle funzioni **Test** e **Read**);
2) L'applicazione può impostare il valore di **m_Unsolicited** chiamando la funzione **ReceiveUnsolicited**; é prudente che questa funzione venga chiamata prima di inviare al *server* la richiesta di ricezione dei messaggi *unsolicited* (ed é obbligatorio che tale richiesta contempli l'inclusione dei messaggi fra parentesi graffe): ciò garantisce che tutti i messaggi ricevuti verranno correttamente scorporati dal buffer di input; viceversa, non bisogna chiamare la funzione (o bisogna chiamarla ripristinando **m_Unsolicited=0**), se si prevede il carattere '**{**' anche nelle risposte "sollecitate" del *server* (oppure se il *server* può inviare dati binari), in quanto il meccanismo di scorporo potrebbe interpretare tale carattere come inizio di un messaggio *unsolicited* e produrre risultati scorretti;
3) Quando il *thread* in cui girano le funzioni di processo del *socket* é avvisato dell'arrivo di nuovi dati, la funzione **StoreRead**, chiamata dalla **ProcessReceive**, li legge in un buffer temporaneo con la funzione di libreria **Receive** e poi: se **m_Unsolicited=0**, accoda semplicemente il buffer temporaneo al buffer generale di input; se invece **m_Unsolicited>0**, riconosce ed estrae dal buffer temporaneo i (o le parti di) messaggi *unsolicited*, memorizzandoli in un buffer separato ("buffer dei messaggi"), e accoda nel buffer di input quello che resta del buffer temporaneo dopo lo scorporo; infine (sempre se **m_Unsolicited>0** e se l'ultimo messaggio *unsolicited* é completo, cioè termina con il carattere '**}**') invia al *main* il messaggio di *Windows* identificato da **m_Unsolicited**, trasmettendogli, nei due parametri standard, l'indirizzo e la lunghezza del "buffer dei messaggi"; per evitare ripetizioni, una variabile-membro del *socket* memorizza un numero che indica quale porzione del buffer é già stata spedita;
4) L'applicazione deve fornire una funzione di *mappatura* del messaggio **m_Unsolicited** (cioè dello stesso identificatore che fornisce come parametro nella chiamata della **ReceiveUnsolicited**); tale funzione deve ricevere, nel *main*, il messaggio inviato dal *thread* e può così accedere al buffer dei messaggi *unsolicited*;
5) La "pulitura" del buffer dei messaggi *unsolicited* é affidata all'applicazione, che può cancellare i messaggi già ricevuti chiamando la funzione **ClearUnsolicited**; l'operazione non può essere fatta dalla *dll* in quanto il messaggio **m_Unsolicited** trasporta (ovviamente) solo il puntatore al buffer e non il suo contenuto e quindi una cancellazione prematura del buffer lo renderebbe inaccessibile all'applicazione; per questo motivo esiste la **ClearUnsolicited**, che va chiamata alla fine della stessa funzione di *mappatura* del messaggio.

APPENDICE A

ISTRUZIONI DA INCLUDERE NELL'APPLICAZIONE

Le istruzioni necessarie affinché un'applicazione possa lavorare correttamente con la *dll*, sono molto semplici e limitate. Nell'elenco che segue, dove compare il nome di un file, si intende che il file sia contenuto nella stessa directory dell'applicazione (diversamente, bisogna specificare il nome completo). Inoltre supponiamo che lo scheletro dell'applicazione sia stato costruito da **AppWizard** e quindi tutti i files e le classi standard siano già presenti.

1. inserire la *direttiva* **#include "SockEnv.h"** nel file **Stdafx.h**;

2. dichiarare un membro di **CMainFrame** puntatore alla classe **CSockEnv** (supponiamo che l'istruzione sia: **CSockEnv* m_pEnv;**);

3. nel *costruttore* di **CMainFrame** porre **m_pEnv(NULL)** nella lista di inizializzazione;

4. inserire, alla fine della funzione **OnCreate** di **CMainFrame**, una delle seguenti *macro* (le lettere maiuscole sono obbligatorie):

CLIENT(m_pEnv)	l'applicazione funzionerà solo come *client*;
SERVER(m_pEnv,Service)	l'applicazione funzionerà solo come *server*;
CLIENTSERVER(m_pEnv,Service)	l'applicazione funzionerà sia come *client* che come *server*;

   dove **Service** é il nome della funzione di "servizio" che deve essere fornita dall'applicazione (se prevede di lavorare in modalità *server*), e deve essere dichiarata come funzione-membro di **CMainFrame**;
   la *macro* opera nel seguente modo:
   a) crea, nell'area di memoria *heap*, un oggetto, istanza della classe **CSockEnv**;
   b) memorizza in **m_pEnv** l'indirizzo dell'oggetto appena creato;
   c) informa la *dll* della modalità in cui l'applicazione intende operare e, in caso di modalità *server*, memorizza nella *dll* il nome della funzione di servizio;

5. nel *distruttore* di **CMainFrame** porre le seguenti istruzioni:

```
if ( m_pEnv != NULL )
{
   if ( ::IsWindow(m_pEnv->m_hWnd) ) m_pEnv-->DestroyWindow();
   else delete m_pEnv;
}
```

6. prima di compilare il programma, selezionare **Build** - **Setting** - **Link** e inserire il nome della libreria: **SockEnv.lib** (infatti lì si trova l'informazione di aggancio a **SockEnv.dll**)

7. in esecuzione occorre che il programma possa *linkare* il file **SockEnv.dll** (che deve trovarsi nella stessa directory, oppure sotto **Windows**, oppure in una directory definita nel *path*)

Se si desidera che l'applicazione lavori in una classe diversa da **CMainFrame** (per esempio, nella classe *documento*), suggeriamo di procedere nel modo seguente:

1. dichiarare nella classe doc. un membro (protetto) **m_pEnv** puntatore a **CSockEnv** e nella classe **CMainFrame** un membro (pubblico) **m_pDoc** puntatore alla classe doc. (inizializzarli con **NULL**);

2. se é prevista la modalità *server*, dichiarare nella classe doc. una funzione **Service**, identica a quella di **CMainFrame**;

3. dichiarare nella classe doc. una funzione **Init()**, chiamabile *una tantum* da menù oppure (per le partenze automatiche) da una funzione di mappatura del messaggio WM_ERASEBKGND della classe di *vista* (prima operazione intercettabile dopo la creazione delle finestre); la **Init()** deve ricavare, con **AfxGetMainWnd**, un puntatore a **CMainFrame** (per esempio **pWnd**) e poi eseguire le istruzioni:
   m_pEnv = pWnd->m_pEnv;
   pWnd->m_pDoc = this;

4. nella stessa **Init()** procedere con le eventuali inizializzazioni (attivazione dei *servers* e dei *clients* e altre operazioni specifiche);

5. se é prevista la modalità *server*, implementare la **Service** di **CMainFrame** con un'unica istruzione, che trasferisce il controllo alla **Service** della classe doc., nel modo seguente:
   return m_pDoc->Service(... stessi parametri ...);

6. se é prevista la modalità *server*, implementare la **Service** della classe doc. con le funzioni di servizio specifiche dei *servers* attivati dall'applicazione.

APPENDICE B

FUNZIONI-MEMBRO CHIAMABILI DALL'APPLICAZIONE

L'elenco che segue descrive in dettaglio le sequenze di chiamata di tutte le funzioni-membro *pubbliche* di **CSockEnv**, che sono chiamabili dall'applicazione:

SERVER

BOOL ActivateServer (int nPort)

nPort numero della *port* del *server* da attivare - a questa *port* deve corrispondere una funzione (fornita dall'*application*, unica per tutti i *server* ma con operazioni differenziate in base a **nPort**), che accede alle domande del *client* e prepara le risposte;

return **TRUE** : OK / **FALSE** : **nPort < 100** , oppure applicazione solo *client*, oppure *server* già in uso sullo stesso *host*, oppure errore nella creazione del *thread*, oppure errore nella creazione del *server* (l'errore si può riconoscere chiamando **GetErrorText**).

BOOL ShowServers ()

return **TRUE** : OK (visualizza la lista dei *server* attivi) / **FALSE** : lista vuota.

BOOL ShowConnected ()

return **TRUE** : OK (visualizza la lista dei *clients* remoti connessi ai *server* attivi; per ogni *client* sono indicati: la *port* del *server* a cui il *client* é connesso, il numero del *socket*, la data e ora della connessione, il numero di "pacchetti" spediti al *server*, l'indirizzo Internet del *client*) / **FALSE** : lista dei *server* attivi vuota.

BOOL SendUnsolicited (int nPort, const char* buf, int len)

nPort numero della *port* del *server* da selezionare;

buf indirizzo del buffer che contiene il messaggio *unsolicited* da inviare ai *clients*;

len lunghezza (in byte) del buffer di cui sopra;

return **TRUE** : OK / **FALSE** : l'applicazione é solo *client*, oppure nessun *server* attivo corrispondente a **nPort**.

BOOL GetClientsArray (CStringArray& ClAr, int nPort)

nPort numero della *port* del *server* da selezionare;

ClAr array di stringhe (vuotato inizialmente), che dovrà contenere i nomi (indirizzi Internet) dei computers che ospitano i *clients* connessi a **nPort** - l'array resta vuoto se nessun *client* é connesso;

return **TRUE** : OK / **FALSE** : l'applicazione é solo *client*, oppure nessun *server* attivo corrispondente a **nPort**.

BOOL CloseServer (int nPort=0)

nPort numero della *port* del *server* da chiudere - se **nPort=0** (*default*) crea una *dialog-box* che mostra la lista dei *servers* attivi e permette all'utente di selezionare il *server* da chiudere;

return **TRUE** : OK / **FALSE** : l'applicazione é solo *client*, oppure nessun *server* attivo, oppure nessuna selezione effettuata nella *dialog-box* (se **nPort=0**) , oppure nessun *server* attivo corrispondente a **nPort** (se **nPort > 0**).

CLIENT

UINT Open (CString str, int nPort, char* pClose=NULL, int nCloseLen=0)

str nome (indirizzo Internet) dell'*host* su cui é attivo il *server* da connettere;

nPort numero della *port* del *server* da connettere;

pClose indirizzo di un buffer da trasmettere al *server* per chiedere la chiusura del *socket* (necessario per tipi particolari di *server*, che non accettano che i *client* connessi si chiudano da soli) - se **pClose=NULL** (*default*) nessun dato verrà spedito durante il processo di chiusura (significa che il *server* é in grado di riconoscere quando un suo *client* si disconnette);

nCloseLen lunghezza (in byte) del buffer di cui sopra;

return se **> 0**, rappresenta il numero del *socket* (che dovrà essere usato in tutte le successive operazioni per identificare il *socket* univocamente) / se **= 0**, l'operazione é fallita: **nPort < 100** , oppure applicazione solo *server*, oppure **str** stringa vuota, oppure errore nella creazione del *thread*, oppure errore nella connessione del *socket*.

UINT Open (int nPort, CString str="", char* pClose=NULL, int nCloseLen=0)

come la precedente, salvo il fatto che l'argomento **str** é opzionale: se é omesso (o é una stringa nulla), significa che il *server* da connettere gira sullo stesso *host* dell'applicazione.

int Test (UINT handle)

handle numero di identificazione del *socket*;

return se **≥ 0**, rappresenta il numero di bytes presenti nel buffer del *socket* / se **< 0**, indica che si é verificato un errore (applicazione solo *server*, oppure numero di *socket* non identificato) .

int Test (UINT handle, char* buf, int len, UINT timeout=0)

handle numero di identificazione del *socket*;

buf indirizzo di un buffer in cui copiare i dati letti nel buffer del *socket* (questi, dopo la copia, non vengono rimossi);

len numero di bytes da copiare - deve essere positivo;

timeout tempo a disposizione (in millisecondi) per attendere che siano presenti almeno **len** bytes nel buffer del *socket* (che viene letto

ogni 10 millisecondi fino al verificarsi della suddetta condizione, oppure fino al raggiungimento di **timeout**) - se **timeout=0** (*default*), viene fatto un solo tentativo;

return il numero di bytes presenti nel buffer del *socket* detratti di **len** (può essere negativo se é stato raggiunto **timeout**); in pratica **return+len** rappresenta il numero totale di bytes nel buffer: se questo numero é negativo, si é verificato un errore (applicazione solo *server*, oppure **len** non positivo, oppure numero di *socket* non identificato).

int Read (UINT handle, char* buf, int len, UINT timeout=INFINITE)

handle numero di identificazione del *socket*;

buf indirizzo di un buffer in cui trasferire i dati, estratti (e rimossi) dal buffer del *socket*;

len numero di bytes da trasferire - deve essere positivo;

timeout tempo a disposizione (in millisecondi) per attendere che siano presenti almeno **len** bytes nel buffer del *socket* (che viene letto ogni 10 millisecondi fino al verificarsi della suddetta condizione, oppure fino al raggiungimento di **timeout**) - se **timeout=INFINITE** (*default*), i dati vengono attesi fino al loro arrivo (lettura *sincrona*);

return il numero di bytes presenti (prima del trasferimento) nel buffer del *socket* detratti di **len** (può essere negativo se é stato raggiunto **timeout**); in pratica **return+len** rappresenta il numero totale di bytes presenti nel buffer prima del trasferimento: se questo numero é negativo, si é verificato un errore (applicazione solo *server*, oppure **len** non positivo, oppure numero di *socket* non identificato).

int Read (UINT handle, CString& str, UINT timeout=INFINITE)

handle numero di identificazione del *socket*;

str riferimento a una stringa in cui trasferire i dati, estratti (e rimossi) dal buffer del *socket* - se, in ingresso, **str** non é vuota, la stringa letta é accodata a quella preesistente;

timeout tempo a disposizione (in millisecondi) per attendere che sia presente almeno **1** byte nel buffer del *socket* (che viene letto ogni 10 millisecondi fino al verificarsi della suddetta condizione, oppure fino al raggiungimento di **timeout**) - se **timeout=INFINITE** (*default*), i dati vengono attesi fino al loro arrivo (lettura *sincrona*);

return se **≥ 0**, rappresenta il numero di bytes residui nel buffer dopo l'estrazione della stringa (comprensiva di *terminator*) / se **= -1**, ha letto tutti i dati presenti nel buffer senza incontrare il *terminator* / se = **-2**, si é verificato un errore (applicazione solo *server*, oppure numero di *socket* non identificato, oppure **timeout** scaduto con buffer ancora vuoto).

BOOL RemoveFromBuffer (UINT handle, int len=0)

handle numero di identificazione del *socket*;

len numero di bytes da rimuovere dal buffer del *socket* - se **= 0** (*default*) tutto il contenuto del buffer deve essere rimosso;

return **TRUE** : OK / **FALSE** : applicazione solo *server*, oppure numero di *socket* non identificato.

BOOL Write (UINT handle, const char* buf, int len)

handle numero di identificazione del *socket*;

buf buffer dei dati da spedire al *server*;

len numero di bytes da spedire - deve essere positivo;

return **TRUE** : OK / **FALSE** : applicazione solo *server*, oppure **len** non positivo, oppure numero di *socket* non identificato.

BOOL Write (UINT handle, CString str)

handle numero di identificazione del *socket*;

str stringa da spedire al *server* (viene trasmesso anche il *terminator*);

return **TRUE** : OK / **FALSE** : applicazione solo *server*, oppure numero di *socket* non identificato.

BOOL ReceiveUnsolicited (UINT handle, UINT message)

handle numero di identificazione del *socket*;

message se **> 0**, rappresenta l'*id* del messaggio di *Windows* in cui devono essere trasmessi all'applicazione i messaggi *unsolicited* ricevuti dal *server* (che pertanto verranno scorporati dal buffer di input del *socket*) / se **= 0**, l'applicazione non vuole più ricevere messaggi (e i messaggi *unsolicited* resteranno insieme agli altri dati nel buffer di input)

return **TRUE** : OK / **FALSE** : applicazione solo *server*, oppure numero di *socket* non identificato.

BOOL ClearUnsolicited (UINT handle)

handle numero di identificazione del *socket* in cui rimuovere, dal buffer dei messaggi *unsolicited*, i messaggi già ricevuti; per liberare la memoria, é utile che questa funzione venga chiamata ogni volta che l'applicazione ha esaminato un messaggio (cioè alla fine della stessa funzione di *mappatura* del messaggio)

return **TRUE** : OK / **FALSE** : applicazione solo *server*, oppure numero di *socket* non identificato.

int GetClientPort (UINT handle)

handle numero di identificazione del *socket*;

return se **> 0**, rappresenta il numero della *port* del *server* a cui é connesso il *socket* / se **= 0**, si é verificato un errore (applicazione solo *server*, oppure numero di *socket* non identificato).

BOOL ShowClients ()

return **TRUE** : OK (visualizza la lista dei *clients* attivi; per ogni *client* sono indicati: la *port* del *server* a cui il *client* é connesso, il numero del *socket*, la data e ora della connessione, il numero di "pacchetti" spediti al *client*, l'indirizzo Internet del *server*) / **FALSE** : lista dei *client* attivi vuota.

BOOL Close (UINT handle)

handle numero di identificazione del *socket*;

return **TRUE** : OK (il *socket* é stato chiuso regolarmente) / **FALSE** : applicazione solo *server*, oppure numero di *socket* non identificato.

CLIENT E SERVER

CFrameWnd* GetAppWnd()

return puntatore alla *Main Window* dell'applicazione.
Questa funzione deve essere usata al posto di **AfxGetMainWnd()** quando l'applicazione sta lavorando in un *thread* (per esempio sta eseguendo la **Service**).

GESTIONE ERRORI

LPCSTR GetErrorText ()

return stringa con il messaggio dell'ultimo errore verificatosi (se vuota: nessun errore).

APPENDICE C

PROPRIETA' E METODI DELLE CLASSI

A maggiore chiarimento di quanto esposto finora, questa Appendice descrive in dettaglio le proprietà e i metodi di tutte le classi della *dll*. Se non é altrimenti specificato, si intende che il valore di ritorno delle funzioni sia **void**, che le funzioni delle classi **CWinSock**, **CClientSock**, **CServerSock** e **CSocketThread** girino nel *thread*, e che le funzioni della classe **CSockEnv** girino nel *main*.

CWinSock **Costruttore** (chiamato da **CSocketThread::InitInstance**)
crea la finestra per la *mappatura* dei messaggi inviati dal *main* - memorizza pointer all'oggetto *thread*;

OnStopThread (*mappa* il messaggio **ID_STOPTHREAD** lanciato da **CSockEnv::OnStopThread** del *main*)
chiama **DestroyWindow** : cancella anche l'oggetto se stesso;
chiama **PostQuitMessage** : interrompe il *thread* e chiama la sua **ExitInstance** ;

OnNewSend (solo *client*, *mappa* il messaggio **ID_NEWSEND** lanciato da **CClientSock::StoreWrite** del *main*)
chiama **CSocketThread::ProcessSend** con il puntatore al *socket* per l'invio dei dati al *server* remoto;

OnUnsolicited (solo *server*, *mappa* il messaggio **ID_UNSOLICITED** lanciato da **CSockEnv::SendUnsolicited** del *main*)
chiama **CSocketThread::ProcessUnsolicited** trasmettendogli indirizzo e lunghezza del buffer da inviare ai *clients*.

CClientSock **Costruttore** (se *client*, chiamato da **CSocketThread::InitInstance**, se *server*, chiamato da **CSocketThread::ProcessAccept**)
memorizza pointer all'oggetto *thread* e inizializza i buffers di input e di output;

OnReceive (*callback*, risponde alla notifica di arrivo di dati)
chiama **CSocketThread::ProcessReceive** con il puntatore a se stesso;

BOOL StoreRead (chiamato da **CSocketThread::ProcessReceive**)
chiama **CSocket::Receive** - accoda i dati ricevuti nel buffer di input - ritorna **FALSE** se non ha ricevuto dati - se *client* e **m_Unsolicited > 0** (*id* del messaggio nel quale notificare al *main* l'arrivo dei messaggi *unsolicited* inviati dal *server*), chiama **SendUnsolicitedMessage**, trasmettendogli i dati memorizzati da **Receive**, prima dell'accodamento nel buffer di input;

SendUnsolicitedMessage (*protected*, solo *client*, chiamato da **StoreRead**)
separa dai dati di input le stringhe dei messaggi *unsolicited*, (riconoscibili in quanto racchiuse fra parentesi graffe), e le memorizza e accoda in un buffer separato - restituisce a **StoreRead** i dati che restano dopo la separazione (**StoreRead** li accoderà nel buffer di input) - se i dati nel buffer separato terminano con '}' (messaggio chiuso), li invia al *main*, impostandoli nel messaggio **m_Unsolicited** e selezionando la parte non già inviata precedentemente;

ClearUnsolicited (solo *client*, chiamato da **CSockEnv:: ClearUnsolicited** del *main*)
elimina dal buffer dei messaggi *unsolicited* la parte già ricevuta dal *main* e riduce (o elimina) l'allocazione di memoria del buffer;

int LoadRead (solo *client*, chiamato da **CSockEnv::ReadOrTest** del *main*)
estrae i dati dal buffer di input (con o senza rimozione) - restituisce il numero di bytes residui nel buffer (può essere negativo se il numero di dati da leggere é maggiore della lunghezza del buffer) - chiama **CClientSock::RemoveFromInBuffer** se é prevista la rimozione dei dati dal buffer;

int LoadRead (solo *client*, chiamato da **CSockEnv::Read** del *main*)
estrae e rimuove una stringa dal buffer di input (procede fino alla fine del buffer oppure fino a quando incontra un **'\0'**) - ritorna **-2** se il buffer é vuoto, **-1** se non ha incontrato **'\0'**, oppure il numero di bytes residui nel buffer - chiama **CClientSock::RemoveFromInBuffer**;

BOOL StoreWrite (solo *client*, chiamato da **CSockEnv::Write** del *main*)
ritorna **FALSE** se il buffer non é vuoto - inserisce dati nel buffer di output - invia al *thread* il messaggio **ID_NEWSEND** con il puntatore a se stesso;

LoadWrite (solo *client*, chiamato da **CSocketThread::ProcessSend**)
chiama **CSocket::Send** tramite cui spedisce i dati presenti nel buffer di output e azzera il buffer;

RemoveFromInBuffer (se *server*, chiamato da **CSocketThread::ProcessReceive**, se *client*, chiamato da **CClientSock::LoadRead** del *main* e da **CSockEnv::RemoveFromBuffer** del *main*)
rimuove un dato numero di bytes dalla testa del buffer di input;

int GetInBuffer (chiamato da **CSocketThread::ProcessReceive**)
restituisce il numero di bytes e l'indirizzo del buffer di input;

int GetInBufferLength (solo *client*, chiamato da **CSockEnv::ReadOrTest** del *main* e da **CSockEnv::RemoveFromBuffer** del *main*)
restituisce il numero di bytes del buffer di input;

CTime GetStartTime (chiamato da **CConLstDlg::ShowClient** del *main*)
restituisce data e ora di collegamento del *socket*;

int GetTransCount (chiamato da **CConLstDlg::ShowClient** del *main*)
restituisce il numero di pacchetti ricevuti dal *socket* (aggiornato da **OnReceive**);

LocalClose (chiamato da **CSocketThread::ExitInstance**)
esegue la chiusura locale di un *socket* - se il *pacchetto* di chiusura non é **NULL**, lo invia al *socket* e poi, se é *client*, chiama **CSocket::Receive** in *loop*, finché non viene chiuso dal *server* - alla fine si autocancella;

OnClose (*callback*, risponde alla notifica di chiusura remota di un *socket*)
chiama **CSocketThread::ProcessClose** con il puntatore a se stesso;

Distruttore rimuove dall'area *heap* i buffers di input e di output;

int m_status (variabile membro *public*, azzerata dal **Costruttore**, utilizzata da **CSocketThread::ProcessReceive**, da **CSocketThread::ProcessClose** e da **CSocketThread:: ExitInstance**) - serve per comunicare (sia in entrata che in uscita) con la funzione di servizio fornita dall'*application*;

UINT m_Unsolicited (variabile membro *public*, azzerata dal **Costruttore**)
se *server*, é una variabile che indica se e come inviare al *socket* i messaggi *unsolicited* - impostata da **CSocketThread::ProcessReceive**, utilizzata da **CSocketThread::ProcessUnsolicited**;
se *client*, rappresenta l'*id* del messaggio di *Windows* in cui trasmettere i messaggi *unsolicited* ricevuti dal *server* (se zero, non inviare i messaggi e non separarli dal buffer di input) - impostata da **CSockEnv::ReceiveUnsolicited** del *main*, utilizzata da **CClientSock::StoreRead** e da **SendUnsolicitedMessage**;

CServerSock Costruttore (chiamato da **CSocketThread::InitInstance**)
memorizza pointer all'oggetto *thread*;

OnAccept (*callback*, risponde alla notifica di richiesta di connessione di un *socket*)
chiama **CSocketThread::ProcessAccept**.

CSocketThread **Costruttore** (se *server*, chiamato da **CSockEnv::ActivateServer** del *main*, se *client*, chiamato da **CSockEnv::Open** del *main*)
inizializza le proprie variabili membro - se *client*, trasferisce in un buffer interno l'eventuale pacchetto di dati da utilizzare per la richiesta di chiusura del *socket*;

InitInstance (chiamato ... come sopra)
lavora in accesso esclusivo - crea un oggetto **CWinSock** - se *server*, crea un oggetto **CServerSock** e chiama **CSocket::Listen** - se *client*, crea un oggetto **CClientSock**, chiama **CSocket::Connect** e (se non vi sono errori) chiama **CSockEnv::MapUpdate** - in entrambi i casi, se c'è un errore, cancella l'oggetto e pone il codice di errore in **m_nInitialStatus**; altrimenti pone **m_nInitialStatus = -1** (inizialmente era zero);

BOOL InThreadQueue (*protected*, chiamato da **ProcessAccept**, **ProcessReceive**, **ProcessSend**, **ProcessClose**, **ProcessUnsolicited**)
serve per accodare l'esecuzione delle funzioni di *callback* - chiama **CSockEnv::ExclusiveAccess** per impedire l'accesso contemporaneo ad altri *thread* (accesso esclusivo) - nello stesso *thread*, se viene lanciata un'operazione mentre un'altra é ancora in esecuzione, mette la nuova operazione in coda ed esce (con **FALSE**) - chiamato senza parametri, segue il percorso inverso: libera l'accesso esclusivo e, se ci sono operazioni in coda, le esegue;

ProcessAccept (solo *server*, chiamato da **CServerSock::OnAccept**)
lavora in accesso esclusivo - se c'è un errore invia alla *dll* del *main* il messaggio **ID_FATALERROR** e termina - crea un oggetto **CClientSock** e lo attiva trasmettendolo come parametro nella chiamata di **CSocket::Accept**;

ProcessReceive (chiamato da **CClientSock::OnReceive**)
lavora in accesso esclusivo - se *server*, chiama **CClientSock::StoreRead** e ricava indirizzo e lunghezza del buffer di input del *socket* chiamando **CClientSock::GetInBuffer** - poi esegue un *loop* di chiamate alla funzione di servizio fornita dall'*application* (non in accesso esclusivo) finché questa non ritorna **FALSE** - dopo ogni chiamata decide, in base ai parametri, se spedire una risposta al *socket* (chiamando **CSocket::Send**) e quali dati rimuovere dal buffer (chiamando **CClientSock::RemoveFromInBuffer** e poi ancora **CClientSock::GetInBuffer** per "ricaricare") - se a un certo punto la funzione ritorna il parametro **status=-1**, significa che é arrivata una richiesta di chiusura del *socket*, che viene rimosso dalla lista e cancellato - se invece la funzione ritorna il parametro **status=-2/-3/-4**, pone **CClientSock::m_Unsolicited = 2/1/0** : il suo valore verrà testato

per sapere se e come inviare al *socket* i messaggi *unsolicited* (vedere capitolo dedicato);
se *client*, chiama **CClientSock:: StoreRead**;

ProcessSend (solo *client*, chiamato da **CWinSock::OnNewSend**)
lavora in accesso esclusivo - chiama **CClientSock::LoadWrite**;

ProcessUnsolicited (solo *server*, chiamato da **CWinSock:: OnUnsolicited**)
lavora in accesso esclusivo - riceve il messaggio *unsolicited* da inviare ai *client* - nella lista dei *socket* seleziona quelli con **CClientSock:: m_Unsolicited > 0** (se é **=2** pone il carattere '{' in testa al messaggio e il carattere '}' in coda), e spedisce il messaggio ai *sockets* selezionati chiamando la **CSocket::Send**;

ProcessClose (chiamato da **CClientSock::OnClose**)
lavora in accesso esclusivo - se *server*, rimuove il *socket* dalla lista e, prima di cancellarlo, pone **m_status=-2** e chiama la funzione di servizio (non in accesso esclusivo) fornita dall'*application* (allo scopo di segnalarle l'evento, se il *client* non prevede una stringa di chiusura, oppure chiude improvvisamente causa aborto o altra malfunzione) -
se *client*, chiama **CSockEnv::MapUpdate**, cancella il *socket* e (dovendo anche terminare il *thread*) invia alla *dll* del *main* il messaggio **ID_STOPTHREAD** con il puntatore a se stesso;

CPtrList& GetConnectedList (solo *server*, chiamato da **CServLstDlg::OnInitDialog** del *main* e da **CConLstDlg::OnInitDialog** del *main*)
restituisce un riferimento alla lista dei *socket* connessi al *server*;

CClientSock* GetClientSock (solo *client*, chiamato da **CConLstDlg::OnInitDialog** del *main* e da **CSockEnv::CloseClient** del *main*)
restituisce un pointer al *socket* del *thread*;

UINT GetClientHandle (solo *client*, chiamato da **CSockEnv::Open** del *main* e da **CSockEnv::GetClientPort** del *main*)
restituisce il numero di *socket* del *thread*;

int GetInitialStatus (se *server*, chiamato da **CSockEnv::ActivateServer** del *main*, se *client*, da **CSockEnv::Open** del *main*)
restituisce il valore di **m_nInitialStatus**;

int GetPort (se *server*, chiamato da **CServLstDlg::OnInitDialog** del *main*, da **CConLstDlg::OnInitDialog** del *main*, da **CSockEnv::GetClientsArray** e da **CSockEnv:: ClosePort** del *main*, se *client*, da **CConLstDlg::OnInitDialog** del *main* e da **CSockEnv::GetClientPort** del *main*)

restituisce il valore di **m_nPort** (numero della *port* del *socket*);

BOOL IsServer (chiamato da **CSockEnv::OnStopThread** del *main*, da **CSocketThread::InitInstance** e da **CClientSock::LocalClose**)
se *server*, restituisce **TRUE**;

ExitInstance (chiamato a seguito dell'istruzione **PostQuitMessage** di **CWinSock:: OnStopThread**)
lavora in accesso esclusivo - se *server*, cancella l'oggetto **CServerSock** e percorre la lista dei *socket* connessi: per ognuno di essi pone **m_status=-1** ed esegue la funzione di servizio fornita dall'*application* per sapere se esiste un *pacchetto* di chiusura, poi chiama **CClientSock::LocalClose** - se *client*, chiama **CSockEnv::MapUpdate** e **CClientSock::LocalClose** trasmettendo il *pacchetto* di chiusura ricevuto da **CSockEnv::Open** del *main* - in entrambi i casi l'esecuzione di **LocalClose** non avviene in accesso esclusivo.

CSockEnv **Costruttore** (chiamato dall'*application*)
riceve (e memorizza in proprie variabili membro): il puntatore alla finestra principale (**m_pAppWnd**), il puntatore alla funzione di servizio (**m_pExecServ**, se é **NULL** non si possono creare *server*), una variabile *booleana* (**m_HasClients**, se é **FALSE** non si possono creare *client*); l'oggetto é creato dall'*application* nell'area *heap* tramite una delle macro: **SERVER**, **CLIENT** o **CLIENTSERVER**, che restituisce l'indirizzo dell'oggetto - chiama **AfxSocketInit** - crea la finestra *dll* per la *mappatura* dei messaggi da inviare al *main* (**ID_STOPTHREAD**, **ID_FATALERROR**) - memorizza in **m_HostName** il nome del proprio *host computer* (restituito dalla funzione globale **gethostname**) - crea un'istanza di **CMutex** per gestire gli accessi esclusivi;

BOOL ExclusiveAccess (chiamato dall'*application* e da parecchie funzioni della *dll*, sia del *main* che del *thread*)
riceve un parametro (**timeout**): se é zero, blocca l'accesso agli altri *thread*, oppure si blocca se un altro *thread* ha eseguito la stessa funzione, aspettando indefinitamente; se é positivo, aspetta per **timeout** millisecondi (e ritorna **FALSE** a tempo scaduto se nel frattempo l'accesso non é stato sbloccato); se é negativo, sblocca l'accesso;

BOOL ActivateServer (solo *server*, chiamato dall'*application*)
riceve il numero della *port* - crea un *client* temporaneo che connette alla *port* di **m_HostName**: se ha successo, un altro *server* é attivo sulla stessa *port* dello stesso *host* e quindi imposta una condizione di errore e ritorna **FALSE**;

se non ha successo la situazione é OK e può procedere - in accesso esclusivo crea un'istanza di **CSocketThread** chiamando l'*overload server* del **Costruttore** - crea un *thread* e poi chiama in *loop* **CSocketThread::GetInitialStatus**, fintanto che questa funzione ritorna zero (cioè lascia lavorare, nel *thread*, la **InitInstance**); appena la funzione ritorna un valore diverso da zero, controlla: se é positivo (codice di errore), cancella il *thread* chiamando **CSockEnv::OnStopThread** con il puntatore al *thread*;

BOOL ShowServers (solo *server*, chiamato dall'*application*)
crea un'istanza della dialog-box associata a **CServLstDlg**, che mostra la lista dei *servers* attivati e il numero dei *socket* connessi - restituisce **FALSE** se la lista é vuota;

BOOL ShowConnected (solo *server*, chiamato dall'*application*)
crea un'istanza delia dialog-box associata a **CConLstDlg**, che mostra la lista dei *sockets* connessi ai *server* (con data, ora, numero del *socket*, numero di messaggi ricevuti e nome dell'*host*) - restituisce **FALSE** se la lista dei *servers* é vuota;

BOOL SendUnsolicited (solo *server*, chiamato dall'*application*)
riceve il numero della *port* del *server* che deve mandare un messaggio *unsolicited* e l'indirizzo e la lunghezza del messaggio, che trasmette al *thread* del *server* selezionato, nel messaggio **ID_UNSOLICITED** - restituisce **FALSE** se l'*application* non può attivare *server* oppure se non trova il *thread*;

BOOL GetClientsArray (solo *server*, chiamato dall'*application*)
percorre la lista dei *sockets* connessi a una data *port* di un *server* - restituisce **FALSE** se l'*application* non può attivare *server* - riempie un array di stringhe (che rimane vuoto se la lista é vuota) con i nomi (indirizzi Internet) dei computers che ospitano i *clients* connessi;

BOOL CloseServer (solo *server*, chiamato dall'*application*)
riceve il numero della *port* da chiudere: se é zero, crea un'istanza di **CServLstDlg** per consentire la selezione del *server* (ritorna **FALSE** se non c'è nessuna selezione) - chiama **ClosePort** e ritorna con il suo valore di ritorno;

CPtrList& GetServerList (solo *server*, chiamato da **CServLstDlg:: OnInitDialog** e da **CConLstDlg:: OnInitDialog**)
restituisce un riferimento alla lista dei *servers* attivati;

BOOL ClosePort (*protected*, solo *server*, chiamato da **CloseServer** e dal **Distruttore**)
lavora in accesso esclusivo - riceve il numero della *port* - seleziona il *thread* (*server* attivato) corrispondente (se il numero é zero seleziona il primo *thread* della lista) - ritorna **FALSE** se la lista é vuota o se non trova il *thread* -

chiama **CSockEnv::OnStopThread** con il puntatore al *thread*;

OnFatalError (solo *server*, *mappa* il messaggio **ID_FATALERROR** lanciato da **CSocketThread::ProcessAccept** del *thread*) mostra il testo dell'errore e poi imposta il messaggio **WM_CLOSE** sulla *Main Window*;

UINT Open (solo *client*, chiamato dall'*application*) riceve: il nome dell'*host server* a cui connettersi, il numero della *port* del *server*, l'indirizzo (*default* **NULL**) e la lunghezza (*default* **0**) di un buffer di dati da trasmettere al *server* come richiesta di chiusura (nei casi in cui il *server* vuole chiudere lui i suoi *clients*) - in accesso esclusivo crea un'istanza di **CSocketThread** chiamando l'*overload client* del **Costruttore** - crea un *thread* e poi chiama in *loop* **CSocketThread::GetInitialStatus**, fintanto che questa funzione ritorna zero (cioè lascia lavorare, nel *thread*, la **InitInstance**); appena la funzione ritorna un valore diverso da zero, controlla: se é positivo (codice di errore), cancella il *thread* chiamando **CSockEnv::OnStopThread** con il puntatore al *thread* - ritorna il numero del *socket* creato (restituito da **CSocketThread:: GetClientHandle**: servirà per tutte le successive operazioni di lettura o scrittura) oppure zero in caso di errore;

UINT Open (solo *client*, chiamato dall'*application*) come il precedente, con la differenza che il nome dell'*host server* e il numero della *port* sono invertiti di posizione: consente di omettere il nome del *server* (oppure di specificare una stringa vuota nel caso che i parametri successivi siano specificati), nel qual caso assume che il *server* si trovi nello stesso computer del *client*;

int Test (solo *client*, chiamato dall'*application*) riceve il numero del *socket* e restituisce il numero di bytes presenti nel buffer di input del *socket* - chiama **ReadOrTest** (che usa in modo particolare);

int Test (solo *client*, chiamato dall'*application*) serve per leggere i dati pervenuti nel buffer di input del *socket*, senza rimuoverli - riceve: il numero del *socket*, l'indirizzo e la lunghezza del buffer in cui trasferire i dati in lettura, il tempo massimo (*timeout*) in millisecondi da attendere affinché i dati siano disponibili (*default* **0**) - chiama **ReadOrTest** trasmettendo gli stessi parametri + **FALSE**;

int Read (solo *client*, chiamato dall'*application*) serve per leggere e rimuovere i dati pervenuti nel buffer di input del *socket* - riceve gli stessi parametri di **Test** (salvo il *default* di *timeout*, che é **INFINITE**) - chiama **ReadOrTest** trasmettendo gli stessi parametri + **TRUE**;

int ReadOrTest (*protected*, solo *client*, chiamato da **Test** e **Read**)

se é chiamato da **Test** e la lunghezza dei dati da leggere é zero, chiama **CClientSock::GetInBufferLength** ed esce; altrimenti procede con operazioni a intervalli di 10 millisecondi (fintanto che non ha letto tutti i dati oppure non ha esaurito il *timeout*) - per ciascuna operazione, in accesso esclusivo, chiama **MapSearch** e **CClientSock::LoadRead** - restituisce un numero (*nrem*) che può essere:

> 0 : ha letto tutti i dati e restano *nrem* bytes nel buffer del *socket*;

= 0 : ha letto tutti i dati presenti nel buffer del *socket*;

< 0 : deve ancora leggere *-nrem* bytes di dati;

-1000000 : parametri errati o numero di *socket* non identificato;

in caso di *nrem* negativo, imposta una condizione di errore (che l'*application* può ricavare chiamando **CSockEnv::GetErrorText**);

int Read (solo *client*, chiamato dall'*application*)

serve per leggere e rimuovere una *stringa* di dati pervenuti nel buffer di input del *socket* - riceve: il numero del *socket*, un riferimento alla variabile *stringa* da riempire, il tempo massimo (*timeout*) in millisecondi da attendere affinché i dati siano disponibili (*default* **INFINITE**) - procede come **ReadOrTest** - restituisce un numero (*nrem*) che può essere:

> 0 : ha incontrato il *terminator* della stringa e restano altri *nrem* bytes nel buffer del *socket*;

= 0 : ha letto tutti i dati presenti nel buffer del *socket* compreso il *terminator*;

-1 : ha letto tutti i dati presenti nel buffer del *socket* (almeno un byte) senza incontrare il *terminator*;

-2 : parametri errati, numero di *socket* non identificato, oppure *timeout* scaduto con buffer del *socket* ancora vuoto; in questo caso imposta una condizione di errore (che l'*application* può ricavare chiamando **CSockEnv::GetErrorText**);

BOOL RemoveFromBuffer (solo *client*, chiamato dall'*application*)

serve per rimuovere un dato numero di bytes dal buffer di input - riceve: il numero del *socket* e il numero di bytes da rimuovere (*default*: rimuovere tutto il buffer) - lavora in accesso esclusivo - chiama **MapSearch** e **CClientSock::RemoveFromInBuffer** - restituisce **FALSE** in caso di errore (numero di *socket* non identificato);

BOOL Write (solo *client*, chiamato dall'*application*)

serve per spedire un buffer di dati al *server* - riceve: il numero del *socket*, l'indirizzo e la lunghezza del buffer di scrittura - lavora in accesso esclusivo - chiama

MapSearch e **CClientSock::StoreWrite** - resta in attesa (liberando l'accesso) se la spedizione precedente non é ancora terminata - restituisce **FALSE** in caso di errore (parametri errati oppure numero di *socket* non identificato);

BOOL Write (solo *client*, chiamato dall'*application*)
serve per spedire una *stringa* (compreso il *terminator*) al *server* - riceve: il numero del *socket* e il valore della stringa da spedire - lavora in accesso esclusivo - chiama **MapSearch** e **CClientSock::StoreWrite** - resta in attesa (liberando l'accesso) se la spedizione precedente non é ancora terminata - restituisce **FALSE** in caso di errore (parametri errati oppure numero di *socket* non identificato);

BOOL ReceiveUnsolicited (solo *client*, chiamato dall'*application*)
serve per impostare il valore di **CClientSock::m_Unsolicited** che definisce se (e dove) spedire i messaggi *unsolicited* ricevuti dal *server* - - riceve: il numero del *socket* e l'*id* del messaggio di *Windows* in cui trasmettere i messaggi *unsolicited* (zero = non trasmettere) - lavora in accesso esclusivo - chiama **MapSearch** - restituisce **FALSE** in caso di errore (numero di *socket* non identificato);

BOOL ClearUnsolicited (solo *client*, chiamato dall'*application*)
serve per rimuovere i messaggi già ricevuti dal buffer degli *unsolicited messages* - riceve il numero del *socket* - lavora in accesso esclusivo - chiama **MapSearch** e **CClientSock::ClearUnsolicited** - restituisce **FALSE** in caso di errore (numero di *socket* non identificato);

BOOL Close (solo *client*, chiamato dall'*application*)
lavora in accesso esclusivo - serve per chiudere un *socket* - riceve il numero del *socket* - chiama **MapSearch** e **CloseClient** - restituisce **FALSE** in caso di errore (numero di *socket* non identificato);

BOOL CloseClient (*protected*, solo *client*, chiamato da **Close** e dal **Distruttore**)
lavora in accesso esclusivo (impostato dall'esterno) - riceve l'indirizzo del *socket* - seleziona il *thread* (*client*) corrispondente (se l'indirizzo é zero seleziona il primo *thread* della lista) - ritorna **FALSE** se la lista é vuota o se non trova il *thread* - chiama **CSockEnv::OnStopThread** con il puntatore al *thread*;

BOOL ShowClients (solo *client*, chiamato dall'*application*)
crea un'istanza della dialog-box associata a **CConLstDlg**, che mostra la lista dei *clients* connessi a *servers* remoti (con data, ora, numero del *socket*, numero di messaggi ricevuti e nome dell'*host*) - restituisce **FALSE** se la lista é vuota;

CPtrList& GetClientList (solo *client*, chiamato da **CConLstDlg:: OnInitDialog**)
restituisce un riferimento alla lista dei *clients* attivati;

int GetClientPort (solo *client*, chiamato dall'*application*)
riceve il numero del *socket* - chiama **CSocketThread:: GetClientHandle** - restituisce il numero della *port* del *server* a cui il *socket* é connesso, oppure **0** in caso di errore (numero di *socket* non identificato);

MapUpdate (solo *client*, chiamato da **CSocketThread::InitInstance** del *thread*, da **CSocketThread::ExitInstance** del *thread* e da **CSocketThread:: ProcessClose** del *thread*)
lavora in accesso esclusivo (impostato e rimosso dall'esterno) - aggiunge o rimuove un elemento in una *mappa* di corrispondenza fra puntatori a *socket* e numeri di *socket*;

BOOL MapSearch (*protected*, solo *client*, chiamato da **ReadOrTest**, **Read**, **Write**, **Close** e **RemoveFromBuffer**)
lavora in accesso esclusivo (impostato e rimosso dall'esterno) - riceve il numero del *socket* e un riferimento al suo indirizzo che imposta dopo averlo cercato nella mappa di corrispondenza - restituisce **FALSE** in caso di errore (numero di *socket* non identificato, in questo caso rimuove l'accesso esclusivo immediatamente);

CFrameWnd* GetAppWnd (chiamato da **CSocketThread:: ExitInstance** del *thread*, da **CSocketThread::ProcessClose** del *thread* e da **CSocketThread::ProcessReceive** del *thread* - può anche essere chiamato dall'*application*)
ritorna il puntatore alla *Main Window* (se *server*, si usa per chiamare la funzione di servizio, che si trova nell'*application* ma gira nel *thread* - se *client*, serve per spedire al *main* la notifica del ricevimento di messaggi *unsolicited*);

BOOL Allowed (*protected*, chiamato da varie funzioni)
riceve un parametro *booleano*: se é **TRUE**, ritorna **FALSE** se non può attivare *server*; se é **FALSE**, ritorna **FALSE** se non può attivare *client*;

BOOL IsListEmpty (*protected*, chiamato da varie funzioni)
riceve due parametri: un riferimento a una lista e un codice di errore: se la lista é vuota imposta l'errore e ritorna **TRUE**;

SetError (solo *server*, chiamato da **CSocketThread::ProcessAccept** del *thread*)
riceve un parametro che imposta come codice dell'ultimo errore;

LPCSTR GetErrorText (chiamato dall'*application* e da **CSockEnv:: OnFatalError**)
restituisce il testo corrispondente al codice dell'ultimo errore;

OnStopThread (se *server*, chiamato da **CSockEnv::ActivateServer** e da **ClosePort**; se *client*, chiamato da **CSockEnv::Open** e da **CloseClient**; *mappa* il messaggio **ID_STOPTHREAD** lanciato (solo *client*) da **CSocketThread::ProcessClose** del *thread*)
riceve come parametro l'indirizzo del *thread*, che rimuove, in accesso esclusivo, dalla sua lista (*server* o *client*) - invia al *thread* il messaggio **ID_ STOPTHREAD** - attende che l'*handle* del *thread* sia *segnalato* - cancella l'oggetto *thread*;

Distruttore chiama **ClosePort** in *loop* fintanto che la lista dei *server* non é vuota - chiama **CloseClient** in *loop* fintanto che la lista dei *client* non é vuota (prima di ogni chiamata di **CloseClient** imposta l'accesso esclusivo) - cancella l'oggetto di **CMutex**.

SERVPOINT m_pExecServ (variabile membro *public*, azzerata dal **Costruttore**, utilizzata da **CSocketThread:: ProcessReceive**, da **CSocketThread::ProcessClose** e da **CSocketThread::ExitInstance**) - punta alla funzione di servizio fornita dall'*application*;
SERVPOINT é un *tipo puntatore a funzione*, così definito:
typedef BOOL (CFrameWnd::* SERVPOINT)(const int, int&, const char*, int&, char*&, int&);

APPENDICE D

PROCEDURE DI CHIUSURA DI UN SOCKET

Un *socket* può essere chiuso con procedure diverse, in base a varie circostanze: anzitutto bisogna distinguere se il *socket* é visto dalla parte *server* o dalla parte *client*; in secondo luogo la chiusura può essere "locale" (cioè di iniziativa dell'applicazione), oppure segue alla presa d'atto che il *socket* remoto é stato chiuso; infine, é anche prevista la possibilità che in certe configurazioni il *socket* non si chiuda da solo, ma invii una richiesta di chiusura alla "controparte". Nell'esposizione che segue classifichiamo queste diverse situazioni in categorie, identificandole con delle sigle, e utilizziamo le sigle per elencare le corrispondenti procedure (espresse come successioni di chiamate a funzioni della *dll*; per i dettagli di comportamento delle funzioni, vedere Appendice C).

A. DA PARTE SERVER

1. **Per iniziativa server**

    a. Chiusura improvvisa
    b. Ordine al *client* di chiudersi (solo messaggio, poi chiude comunque)

2. **Per iniziativa client**

    a. Notifica di avvenuta chiusura
    b. Richiesta dal *client* di chiuderlo

B. DA PARTE CLIENT

1. **Per iniziativa server**

    a. Notifica di avvenuta chiusura
    b. Ordine dal *server* di chiudersi (non esiste nei nostri *client*)

2. **Per iniziativa client**

    a. Chiusura improvvisa
    b. Richiesta al *server* di essere chiuso

PROCEDURE

Legenda : **W** : istanza di **CWinSock**
C : istanza di **CClientSock**
S : istanza di **CServerSock**
T : istanza di **CSocketThread**
E : istanza di **CSockEnv**
A : istanza dell'*application*
[*main*] : da questo punto in poi gira nel *main*
[*thread*] : da questo punto in poi gira nel *thread*

A.1.a Errore iniziale : [*main*] **A - E.ActivateServer -** [*thread*] **T.InitInstance - T.delete S -** [*main*] **E.OnStopThread - W.SendMessage(ID_STOPTHREAD) - E.Wait -** [*thread*] **W.OnStopThread - W.DestroyWindow - PostQuitMessage -** [*main*] **E.delete T**

Errore fatale : [*thread*] **S.OnAccept - T.ProcessAccept - E.PostMessage(ID_FATALERROR) -** [*main*] **E.OnFatalError - A.PostMessage(WM_CLOSE) - A.delete E - E.Distruttore -**
(tutti i *server*)→ {{ **E.ClosePort - E.OnStopThread - W.SendMessage(ID_STOPTHREAD) - E.Wait -** [*thread*] **W.OnStopThread - W.DestroyWindow - PostQuitMessage - T.ExitInstance - T.delete S -**
(tutti i *connected*) →{**C.LocalClose - C.delete C**} -
[*main*] **E.delete T**}}

Chiusura server : [*main*] **A - E.CloseServer - E.ClosePort - E.OnStopThread - W.SendMessage(ID_STOPTHREAD) - E.Wait -** [*thread*] **W.OnStopThread - W.DestroyWindow - PostQuitMessage - T.ExitInstance - T.delete S -** (tutti i *connected*) →{**C.LocalClose - C.delete C**} - [*main*] **E.delete T**

Chiusura programma : [*main*] **A.delete E - E.Distruttore -**
(tutti i *server*)→ {{**E.ClosePort - E.OnStopThread W.SendMessage(ID_STOPTHREAD) - E.Wait -** [*thread*] **W.OnStopThread - W.DestroyWindow - PostQuitMessage - T.ExitInstance - T.delete S -**
(tutti i *connected*) →{**C.LocalClose - C.delete C**} - [*main*] **E.delete T**}}

A.1.b Errore fatale : [*thread*] **S.OnAccept - T.ProcessAccept - E.PostMessage(ID_FATALERROR) -** [*main*] **E.OnFatalError - A.PostMessage(WM_CLOSE) - A.delete E - E.Distruttore -**
(tutti i *server*)→ {{**E.ClosePort - E.OnStopThread - W.SendMessage(ID_STOPTHREAD) - E.Wait -** [*thread*]

W.OnStopThread - W.DestroyWindow - PostQuitMessage - T.ExitInstance - T.delete S -
(tutti i *connected*) →{**A.Service**(-1) - **C.LocalClose - C.Send - C.delete C**} - [*main*] **E.delete T**}}

Chiusura server : [*main*] **A - E.CloseServer - E.ClosePort - E.OnStopThread - W.SendMessage(ID_STOPTHREAD) - E.Wait -** [*thread*] **W.OnStopThread - W.DestroyWindow - PostQuitMessage - T.ExitInstance - T.delete S -** (tutti i *connected*) →{**A.Service**(-1) - **C.LocalClose - C.Send - C.delete C**} - [*main*] **E.delete T**

Chiusura programma : [*main*] **A.delete E - E.Distruttore -**
(tutti i *server*)→ {{**E.ClosePort - E.OnStopThread - W.SendMessage(ID_STOPTHREAD) - E.Wait -** [*thread*] **W.OnStopThread - W.DestroyWindow - PostQuitMessage - T.ExitInstance - T.delete S -**
(tutti i *connected*) →{**A.Service**(-1) - **C.LocalClose - C.Send - C.delete C**} - [*main*] **E.delete T**}}

A.2.a Chiusura remota : [*thread*] **C.OnClose - T.ProcessClose - A.Service**(-2) - **T.delete C**

A.2.b Richiesta di chiusura : [*thread*] **C.OnReceive - T.ProcessReceive - A.Service - T.delete C** (se é restituito -1 da **A.Service**)

B.1.a Chiusura remota : [*thread*] **C.OnClose - T.ProcessClose - E.MapUpdate - T.delete C - E.PostMessage(ID_STOPTHREAD) -** [*main*] **E.OnStopThread - W.SendMessage(ID_STOPTHREAD) - E.Wait -** [*thread*] **W.OnStopThread - W.DestroyWindow - PostQuitMessage -** [*main*] **E.delete T**

B.2.a Errore iniziale : [*main*] **A - E.Open -** [*thread*] **T.InitInstance -** [*main*] **E.OnStopThread - W.SendMessage(ID_STOPTHREAD) - E.Wait -** [*thread*] **W.OnStopThread - W.DestroyWindow - PostQuitMessage -** [*main*] **E.delete T**

Chiusura client : [*main*] **A - E.Close - E.CloseClient - E.OnStopThread - W.SendMessage(ID_STOPTHREAD) - E.Wait -** [*thread*] **W.OnStopThread - W.DestroyWindow - PostQuitMessage - T.ExitInstance - E.MapUpdate - C.LocalClose - C.delete C -** [*main*] **E.delete T**

Chiusura programma : [*main*] **A.delete E - E.Distruttore -**
(tutti i *client*) → {**E.CloseClient - E.OnStopThread - W.SendMessage(ID_STOPTHREAD) - E.Wait -** [*thread*] **W.OnStopThread - W.DestroyWindow - PostQuitMessage - T.ExitInstance - E.MapUpdate - C.LocalClose - C.delete C -** [*main*] **E.delete T**}

B.2.b Chiusura client : [*main*] **A - E.Close - E.CloseClient - E.OnStopThread - W.SendMessage(ID_STOPTHREAD) - E.Wait -** [*thread*] **W.OnStopThread - W.DestroyWindow - PostQuitMessage - T.ExitInstance - E.MapUpdate - C.LocalClose - C.Send - C.Receive** (in loop, fino a quando non dà errore) - **C.delete C -** [*main*] **E.delete T**

Chiusura programma : [*main*] **A.delete E - E.Distruttore -**
(tutti i *client*) → {**E.CloseClient - E.OnStopThread - W.SendMessage(ID_STOPTHREAD) - E.Wait -** [*thread*] **W.OnStopThread - W.DestroyWindow - PostQuitMessage - T.ExitInstance - E.MapUpdate - C.LocalClose - C.Send - C.Receive** (in loop, fino a quando non dà errore) - **C.delete C -** [*main*] **E.delete T**}

Ringraziamenti

L'autore ringrazia il collega **Giuseppe Maccaferri** per la preziosa collaborazione nelle fasi di progettazione e di test della libreria.