Blackjack C/S

Università di L'Aquila
Facoltà di Ingegneria

Reti di calcolatori

Prof. Gabriele Di Stefano

A.A. 2001/2002

Tesina a cura di:

Salvatore Meschini
Valerio Tarquini

Prima parte

Introduzione

In questo documento vengono descritte gli strumenti impiegati nella creazione, le scelte progettuali e le modalità di utilizzo del programma Blackjack C/S. Un'attenzione particolare è riservata all'analisi dell'implementazione.

Abstract

Il progetto Blackjack C/S è diviso in due parti distinte ma interagenti: un server ed un client. Al server è delegata la gestione delle partite individuali, in altre parole rappresenta il banco con cui ogni giocatore deve confrontarsi tramite il client. Il server è pensato come una completa stazione di controllo in quanto, oltre all'amministrazione delle regole di gioco, consente ad un operatore di: inviare messaggi ai giocatori collegati, terminare una partita in corso, salvare la lista degli eventi su file. Il client ha essenzialmente il compito di permettere al giocatore di sfidare il computer in una versione semplificata del Blackjack (con la possibilità di modificare la posta in gioco). Il progetto è stato realizzato in Delphi e richiede una versione recente del sistema operativo Windows. I test sono stati effettuati su sistemi Windows 98/ME/2000, alcune caratteristiche avanzate richiedono la presenza di Windows 2000.

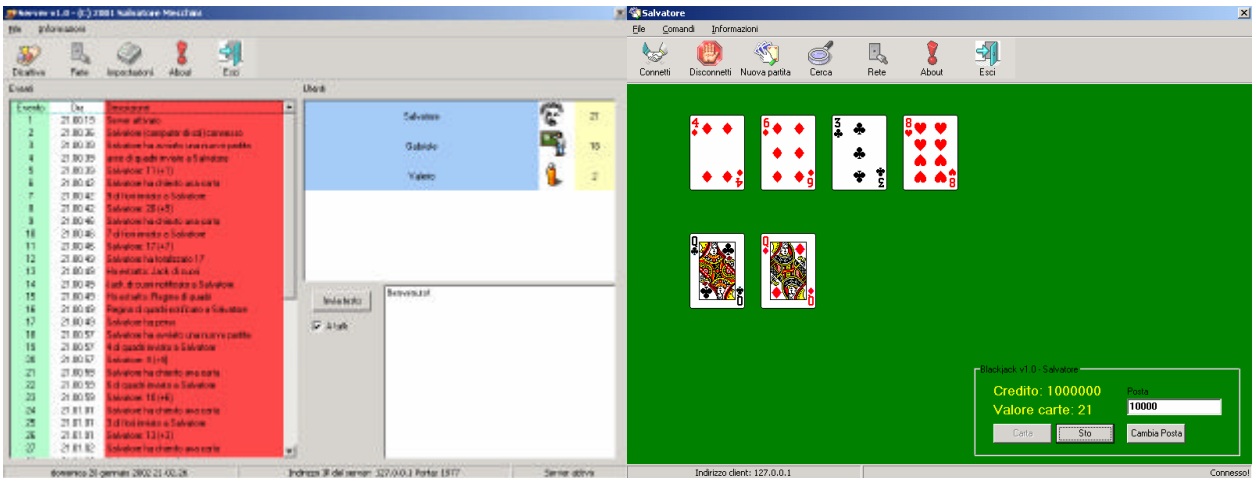


Figura 1 Le applicazioni Server e Client

Gli strumenti

La tecnologia RAD (Rapid Application Development) ha cambiato il modo di programmare quasi quanto l'introduzione del paradigma ad oggetti, partendo da questo presupposto è stato preferito l'ambiente Delphi a linguaggi altrettanto potenti (es. C++ e Java) ma privi di strumenti RAD comparabili.

Delphi

Borland Delphi è un ambiente di sviluppo RAD basato sul moderno linguaggio Object Pascal, la versione ad oggetti del Pascal, e dotato di una serie di strumenti, librerie, classi e wizard che facilitano la vita del programmatore. E' l'ideale per sviluppare in tempi rapidi programmi stabili, performanti, user-friendly… Grazie alla VCL (Visual Component Library) è garantita l'integrazione con il sistema Windows. Recentemente è stato rilasciato Kylix 2, l'analogo di Delphi per sistemi LINUX, un modo per ottenere un'ottima portabilità del codice legacy. Impiegando Delphi la "programmazione visuale" non è limitata a semplici applicazioni ma è estesa a:

- Database locali e distribuiti/SQL/Report
- Corba/DCOM
- Servizi Web (SOAP, WebSnap, …) e applicazioni per Web server
- Applicazioni Internet
- XML
- ActiveX/COM/COM+/Office Automation
- Multithreading
- Grafica (OpenGL, DirectX, …)

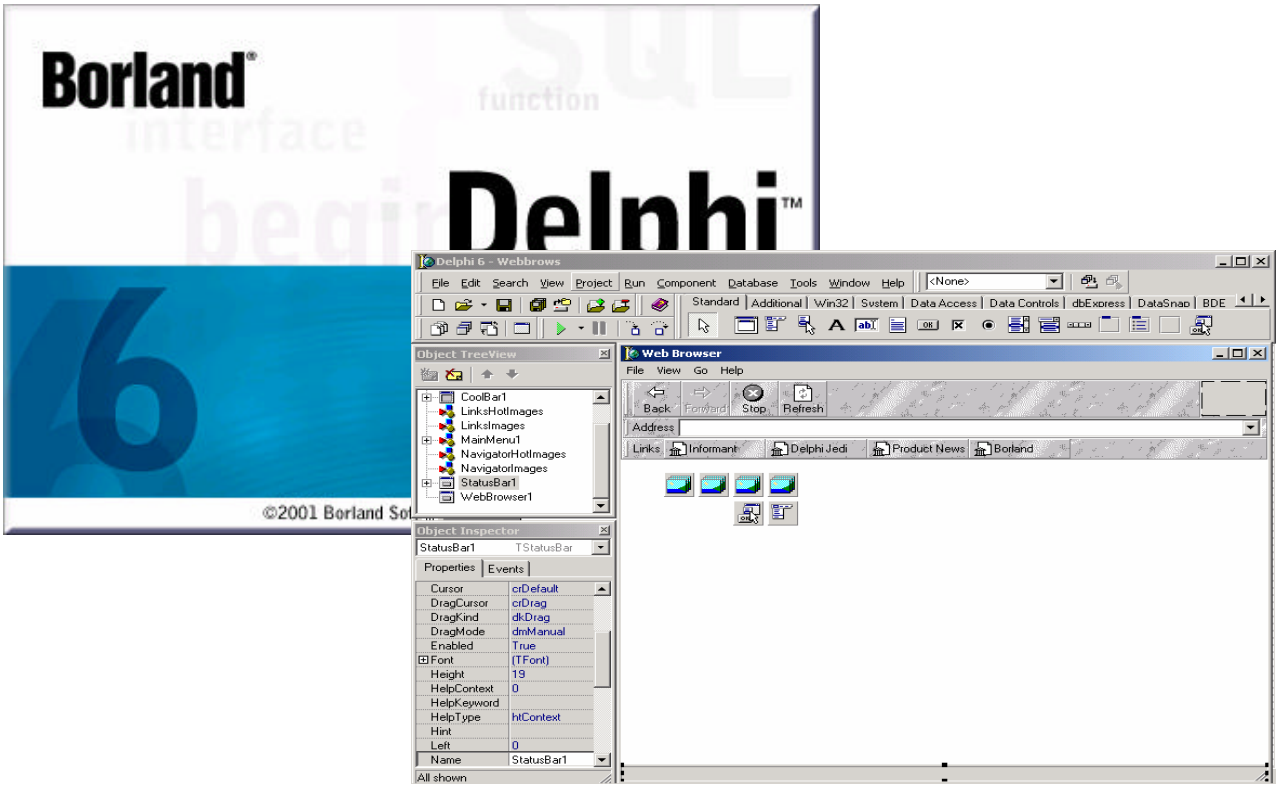


Figura 2 Un webbrowser realizzato in Delphi

L'unico svantaggio rispetto ai concorrenti C++/Java dei programmi realizzati in Delphi è la portabilità su piattaforme diverse da Windows/Linux. Ciò non rappresenta un problema per la maggior parte dei programmatori. La tesina per il corso di Reti di calcolatori è interamente sviluppata in Object Pascal. Ai componenti nativi sono state preferite le librerie ad alto livello Indy, incluse in Delphi a partire dalla versione 5.

Indy

Indy (Internet Direct) è un progetto open-source che ha lo scopo di implementare una suite di componenti per Internet basati sui *blocking sockets*. Indy è disponibile per Delphi, Kylix e C++ Builder. Include tutti i protocolli Internet più diffusi e un corposo insieme di classi e funzioni. L'uso dei *blocking sockets* implica che una chiamata a funzione non ritorna fino al momento in cui l'operazione non è completata, in tale caso il programma è reso inutilizzabile durante l'esecuzione di una chiamata. Il porting dei socket in ambiente Windows ha incontrato il problema dell'assenza della chiamata fork, solo con Win32 è stata introdotta una vera programmazione multi-threading. In Unix la chiamata fork (una sorta di multi-threading ma con processi al posto dei thread) è spesso utilizzata da client e daemon per sfruttare al meglio i blocking sockets. Analizziamo brevemente i vantaggi e gli svantaggi dei blocking sockets.

Vantaggi:

- Facilità di programmazione – Tutto il codice utente può essere eseguito sequenzialmente.
- Facilità di porting – I blocking sockets sono standard in Unix.
- Facilità di impiego nei thread – La loro sequenzialità si presta ad essere impiegata nei thread.

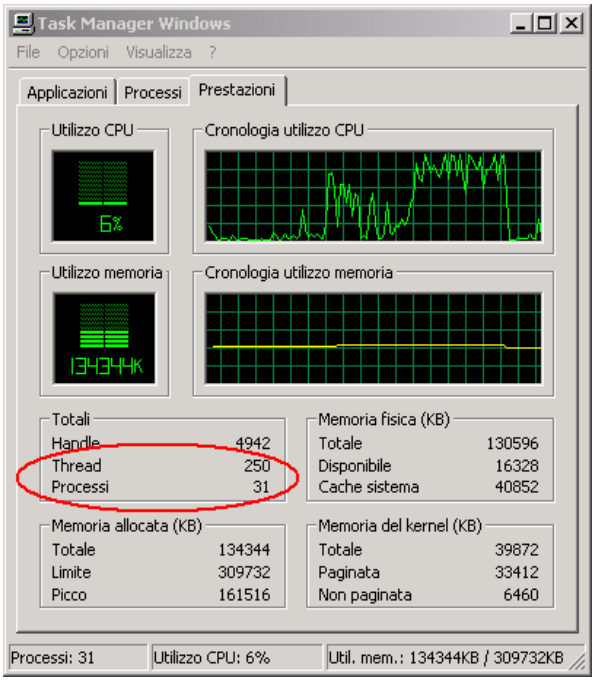
Svantaggi:

- I blocking sockets non ritornano finchè non hanno completamente eseguito il loro compito. Se una chiamata ad un blocking socket appartenente al thread principale dell'applicazione non si conclude l'interfaccia utente risulta bloccata. Per evitare questo inconveniente i progettisti della suite Indy hanno previsto la classe TidAntiFreeze.

Threads

Negli ambienti Windows il concetto di processo coincide con quello di "programma in esecuzione ". Ogni processo ha a disposizione risorse (memoria, disco, tempo di CPU, dispositivi di I/O,…) virtualizzate, in altre parole ogni programma in esecuzione possiede il proprio set di risorse virtuali non accessibili agli altri processi. Un processo esegue moduli disgiunti o condivisi, nel primo caso le porzioni di codice non sono in comune nel secondo si ricorre a librerie a collegamento dinamico (DLL). Il passo successivo è l'introduzione dei *threads*, una metodologia che consente ad uno stesso processo di avviare e gestire più *azioni* contemporaneamente. Una soluzione è quella di creare un processo ex-novo per ogni azione concorrente per poi stabilire una comunicazione tra processi. Una soluzione più elegante è rappresentata dai thread: uno stesso processo (applicazione) crea un nuovo thread per ogni azione concorrente da svolgere. La maggior parte dei sistemi si basa sul time slicing, anche

noto come pre-emptive multitasking, perciò anche una macchina con singola CPU può gestire il multi-threading. Ad ogni thread è garantito un breve lasso ti tempo, di solito 55ms, per eseguire il proprio compito prima che il controllo del processore passi ad un altro thread secondo un preciso meccanismo di attribuzione delle priorità. Per ottimizzare le macchine con più processori è necessario avvalersi della programmazione concorrente.



I componenti Indy sono stati progettati per essere gestiti all'interno di thread. Creare e distruggere thread richiede risorse di calcolo, di conseguenza per server caratterizzati da molte connessione di breve durata è opportuno utilizzare la classe TIdThreadMgrPool della libreria Indy. Tale classe non crea/distrugge thread ma li prendi in prestito da un insieme di thread inattivi. Un tipico server in Unix ha uno o più processi "in ascolto" e per ogni client genera un nuovo processo mediante *fork*. I server Indy sono comunemente incentrati su un thread "in ascolto" e allocano un thread per ogni connessione.

Figura 3 Ad un processo possono essere associati più thread

Come al solito Delphi fornisce più soluzioni per uno stesso problema, la gestione dei thread può avvenire, come negli altri linguaggi di programmazione, per mezzo della API Windows a basso livello o mediante la classe TThread. La classe TThread astrae il programmatore dagli aspetti tecnici riducendo le difficoltà e le "trappole" insite nella programmazione concorrente. In Delphi il thread è visto come un normale oggetto, non mancano perciò metodi, costruttori e proprietà implementabili visualmente. Il complesso meccanismo della sincronizzazione tra thread diversi è notevolmente semplificato dal metodo synchronize appartenente alla classe TThread.

Alternative

Programmare in Delphi significa avere accesso ad una comunità di sviluppatori molto attiva e quindi ad una vasta gamma di componenti di ottima qualità. Una delle scelte progettuali ha riguardato il livello di astrazione dalla rete, sono state scartate tutte le alternative ad "alto livello" e quelle fornite di base con tutte le versioni di Delphi. Di seguito vengono brevemente descritte le principali classi standard per i socket:

- ESocketError è la classe per la gestione delle eccezioni che si verificano durante la creazione, distruzione e uso di oggetti socket.
- TBaseSocket incapsula le operazioni locali sui descrittori di file.
- TClientSocketThread consente di creare thread separate per ogni operazione.
- TCustomIpClient incapsula le funzioni di rete lato client
- TIpSocket incapsula le funzioni di supporto per reti
- TRawSocket incapsula un raw socket
- TTcpClient componente per client TCP
- TTcpServer componente per server TCP
- TUdpSocket componente per il protocollo UDP

Seconda Parte

Implementazione

Blackjack C/S è un progetto per il corso di Reti di calcolatori, si è perciò deciso di evitare, per quanto possibile, la programmazione visuale nella gestione delle connessioni. La libreria Indy prevede un insieme di componenti, fondati sui *command handler*, per la costruzione visuale delle applicazioni di rete e dei protocolli. I command handler permettono di specificare le richieste del client e le relative risposte del server senza scrivere codice. Il progetto non utilizza gli automatismi forniti da Delphi, è quindi caratterizzato da un notevole numero di righe non necessarie di codice (incluse solo per mostrare le operazioni degli applicativi e lo scambio di informazioni tra client e server). Il progetto è modularizzato in unit, le singole unit svolgono attività distinte coordinate però dal modello event-driven e ad oggetti di Delphi. Le unit fondamentali che compongono l'applicazione sono:

- ServerUnit – Contiene tutte le routine usate dal server;
- ClientUnit – E' il "cuore" del client;
- Protocollo – E' formata dai comandi scambiati tra client e server;
- BlackjackUnit – Espone le regole semplificate del gioco del blackjack;
- ConnettiUnit – Mostra la finestra che agevola la connessione al server;
- ConfigUnit – Salva su file la configurazione del server;
- InfoUnit – Funzioni necessarie per ottenere informazioni sul sistema e sulla rete;
- CreditiUnit – Mostra informazioni sul programma in una tipica finestra About;
- MsgUnit – Visualizza i messaggi inviati dal server;
- ChatUnit – Invia e visualizza i messaggi di "chat".

Alle unit elencate bisogna aggiungere i moduli Indy, i componenti usati nel realizzare l'interfaccia utente, le classi per le carte e il protocollo ICMP. Segue un esame approfondito del codice delle routine più rilevanti e del protocollo. Pur essendo una

tesina e non un programma con valore commerciale il codice risulta relativamente robusto grazie alla gestione delle eccezioni con il costrutto *try…except…finally*.

Protocollo

Il pacchetto dati è costituito da un record di due campi definito come:

```
// E' il pacchetto dati scambiato tra client/server
type
  TBloccoComunicazione = record
    Comando: integer;
    Messaggio: string[255];
  end;
```

Client e server si scambiano variabili di tipo TBloccoComunicazione durante una sessione TCP. Quasi tutti i comandi supportano anche una stringa opzionale, per esempio al comando CambiaPosta è associato un messaggio contenente la nuova posta, mentre al comando Fine il server aggiunge il motivo della disconnessione…

```
  // Costanti usate nel campo Comando del pacchetto dati
const
  NuovaConnessione = 0; // Tentativo di connessione
  NuovaPartita = 1; // Inizia una nuova partita
  Sto = 2; // Il giocatore si ferma
  NuovaCarta = 3; // Viene richiesta/servita una nuova carta
  CambiaPosta = 4; // La posta viene cambiata dal giocatore
  Vittoria = 5; // Il giocatore ha vinto
  Sconfitta = 6; // Il giocatore ha perso
  UsaIcona = 7; // Il giocatore ha scelto un'icona
  Fine = 8; // Fine della connessione
  CambiaCredito = 9; // Il credito residuo viene aggiorato
  MsgTesto = 10; // Messaggio di testo dal server
  CambiaPunteggio = 11; // Il punteggio viene aggiornato
  Chattata = 12; // Messaggio chat
  DatiInArrivo = 13; // Il server sta per inviare dei dati sui giocatori
  Dati = 14; // Dati relativi ai giocatori
```

Client

In questa sezione vengono descritte le funzioni di rete del client, l'interazione con l'utente ed il relativo codice sorgente sono illustrati nella terza parte.

Program Client

Il codice seguente è creato automaticamente dall'ambiente Delphi ed è necessario per instanziare l'oggetto applicazione. Essendo molto simile al codice relativo al server è stata omessa la dichiarazione del program Server. Le uniche righe da mettere in evidenza sono quelle che effettuano i controlli sulla versione della libreria Winsock e sulla risoluzione video.

```
program Client;

uses
  Forms, Windows,
  ClientUnit in 'ClientUnit.pas' {FinestraClient},
  ConnettiUnit in 'ConnettiUnit.pas' {FinestraConnessione},
  MsgUnit in 'MsgUnit.pas' {FinestraMsg},
  Infounit in 'InfoUnit.pas';

{$R *.res}

begin
  // E' richiesta una versione >= 2.0 della libreria Winsock:
  if VersioneWS < 2 then MessageBox(0, 'Libreria Winsock troppo datata (minimo
2.0)!', 'ERRORE', 16)
  else
  begin
  // La risoluzione minima e' 800x600
    VerificaRisoluzione;
  // Vengono richiamati diversi metodi dell'oggetto Application
    Application.Initialize;
    Application.Title := 'Client v1.0';
    Application.CreateForm(TFinestraClient, FinestraClient);
    Application.Run;
  end;
end.
```

Modalità operative del client

Questo paragrafo delinea le azioni compiute dal client per stabilire connessioni e scambiare dati con il server. Il nucleo del client è rappresentato da una semplice routine denominata InviaAlServer. Tramite chiamate a InviaAlServer si inviano i comandi che formato il protocollo di comunicazione ed eventuali messaggi di testo. Il codice della procedura InviaAlServer fa uso dell'oggetto client definito nella libreria Indy:

```
procedure TFinestraClient.InviaAlServer(codice: integer; messaggio: string);
var
  Informazioni: TBloccoComunicazione;
begin
  // Il comando deve essere inviato solo se la connessione risulta attiva
  if Client.Connected then
   begin
    Informazioni.Comando := Codice; // Vedi protocollo
    Informazioni.Messaggio := Messaggio; // Messaggio di testo opzionale
    Client.WriteBuffer(Informazioni, SizeOf(Informazioni), True); // Invia!
   end;
end;
```

Nota: Il parametro True del metodo WriteBuffer esclude l'utilizzo del buffer in scrittura, dunque i dati vengono immediatamente inviati al server.

Tutte le comunicazioni con il server passano attraverso chiamate a InviaAlServer. In particolare il client utilizza InviaAlServer per: connettersi, cambiare la posta,

notificare al server la scelta dell'utente (stai o nuova carta), iniziare una nuova partita, cambiare server. La variabile Informazioni è il pacchetto dati inviato al server ed è tipizzata come TBloccoComunicazione. A titolo di esempio viene proposta l'implementazione della procedura che stabilisce una connessione con il server:

```
procedure TFinestraClient.Connetti(nome: string; miaicona: integer;
  indirizzo, porta: string);
begin
  try
    if Client.Connected then exit; // Se il client è  già  collegato termina
Client.Host := Indirizzo; // Imposta Indirizzo e Porta del server
    Client.Port := StrToInt(Porta);
    Client.Connect(5000); // // VAI!!! Con un timeout = 5000ms
    ThreadGestioneClient := TThreadGestioneClient.Create(True); // Nuovo thread
    ThreadGestioneClient.FreeOnTerminate := True;
    ThreadGestioneClient.Resume;
  except
    on E: Exception do MessageDlg('Si è  verificato un errore!' + #13 +
E.Message,
        mtError, [mbOK], 0);end;
  InviaAlServer(NuovaConnessione, Nome);  // Invia dati login
  InviaAlServer(UsaIcona, IntToStr(MiaIcona)); // Ora invia l'icona end;
```

Quando il giocatore risulta effettivamente collegato/scollegato al server vengono generati degli eventi, il client risponde a tali eventi con le seguenti routine:

```
procedure TFinestraClient.ClientConnected(Sender: TObject);
begin
  // Aggiorna la barra di stato
  BarraStato.Panels[1].Text := 'Connesso!'
  // Altri dettagli estetici: cambia il nome della finestra/applicazione/…
  Caption:=Nome;
  Application.Title := Nome;
  CCBox.Caption := 'Blackjack v1.0 - ' + Nome + ' ';
end;

procedure TFinestraClient.ClientDisconnected(Sender: TObject);
begin
  // Aggiorna la barra di stato
  BarraStato.Panels[1].Text := 'Non connesso';
  PartitaInCorso := False;
end;
```

Per quanto detto in precedenza non è opportuno operare con i blocking threads nel thread principale dell'applicazione. Nel momento in cui si verifica la connessione il client provvede a generare un nuovo thread, il metodo Execute del thread è:

```
procedure TThreadGestioneClient.Execute;
begin
  while not Terminated do // Finchè il thread è attivo esegui:
   begin
    if not FinestraClient.Client.Connected then Terminate
    else
      try
        FinestraClient.Client.ReadBuffer(BC, SizeOf(BC)); // Ricevi dal server
        Synchronize(GestisciInput); // Gestisci i comandi inviati dal server
      except
       end;
   end;
```

```
end;
```

Il thread GestioneClient “ascolta” il server e quando riceve dei comandi validi li elabora sincronizzando la procedura GestisciInput:

```
procedure TThreadGestioneClient.GestisciInput;
begin
  case BC.Comando of
    Vittoria:
     begin
      PartitaInCorso := False;
      // Ora e' possibile scommettere
      FinestraClient.CambiaBtn.Enabled := True;
      MessageDlg(bc.Messaggio, mtInformation, [mbOK], 0);
     end;
    Sconfitta:
     begin
      PartitaInCorso := False;
      // Ora e' possibile scommettere
      FinestraClient.CambiaBtn.Enabled := True;
      MessageDlg(bc.Messaggio, mtInformation, [mbOK], 0);
     end;
    NuovaPartita:
     begin
      PartitaInCorso := True;
      // Rien ne va plus :-)))
      FinestraClient.CambiaBtn.Enabled := False;
      FinestraClient.CancellaCarte;
      FinestraClient.MostraCarta(BC.Messaggio, MaxX, Maxy);
     end;
    NuovaCarta:
     begin
      inc(CarteGiocate);
      inc(MaxX, 91);
      if (FinestraClient.Width - (Maxx + 71)) < 80 then
       begin
        inc(MaxY, 100);
        MaxX := 80;
       end;
      FinestraClient.MostraCarta(BC.Messaggio, MaxX, MaxY);
     end;
    Fine:
     begin
      FinestraClient.Client.Disconnect;
      messagedlg(bc.Messaggio, mtWarning, [mbOK], 0);
     end;
    CambiaCredito:
     begin
      FinestraClient.AggiornaCredito(Bc.Messaggio);
     end;
    DatiInArrivo:
     begin
      Giocatori.ListaGiocatori.ListBoxItems.Clear; // Pulisci la lista
      Giocatori.Caption:='Giocatori connessi: '+Bc.Messaggio;
     end;
    Dati:FinestraClient.AggiornaDati(Bc.Messaggio);
    Chattata:FinestraChat.ListaChat.Lines.Add(Bc.Messaggio);
    CambiaPunteggio: FinestraClient.AggiornaPunteggio(bc.messaggio);
    MsgTesto: FinestraClient.VisualizzaMsg(bc.messaggio);
   end;
end;
```

Server

Il server in estrema sintesi deve rispondere alle richieste inviate dal client. La procedura richiamata è simile alla InviaAlServer, l'unica differenze è il parametro AChi che indica a quale client (thread) inviare il pacchetto dati:

```
procedure TFinestraServer.InviaAlClient(Achi: TIdPeerThread; codice: integer;
  messaggio: string);
var
  Informazioni: TBloccoComunicazione;
begin
  Informazioni.Comando := Codice;
  Informazioni.Messaggio := Messaggio;
  Achi.Connection.WriteBuffer(Informazioni, SizeOf(Informazioni), True);
end;
```

I giocatori sono memorizzati in una lista, il generico elemento ha la struttura di un record:

```
type
  PGiocatore = ^TGiocatore;
  TGiocatore = record // Giocatore
    Nome: string[50]; // Nome scelto dal giocatore
    CollegatoAlle: TDateTime; // Ora in cui avviene la connessione
    Posta, Punteggio, CarteGiocate, PuntiCpu: integer; // Varie
    ListaCarte, CarteCpu: array[1..30] of integer; // Info sulla mano
    Thread: Pointer; // Puntatore al thread
  end;
```

Modalità operative del server

All'evento "nuova connessione" da parte di un client il server risponde richiamando il metodo ServerConnect:

```
procedure TFinestraServer.ServerConnect(AThread: TIdPeerThread);
var
  NuovoGiocatore: PGiocatore;
  BC: TBloccoComunicazione;
  Nome: string;
begin
  /// Ricevi le informazioni
  // if not AThread.Terminated and AThread.Connection.Connected then
  AThread.Connection.ReadBuffer(BC, SizeOf(BC));
  // Stampa le informazioni
  if Bc.Comando = NuovaConnessione then
   begin
    Nome := Bc.Messaggio;
    // Mi serve per aggiungerlo alla lista dopo aver ricevuto l'icona
    AggiornaLog(Bc.Messaggio + ' connesso');
   end;
  // if not AThread.Terminated and AThread.Connection.Connected then
  AThread.Connection.ReadBuffer(BC, SizeOf(BC));

  // Non possono esistere 2 giocatori con lo stesso nome
  if GiocatorePresente(Nome) then
   begin
    InviaAlClient(Athread,Fine,'Scegli un nome diverso!');
```

```
    Exit;
   end;

  // Il nome non può contenere il simbolo - (vedi InviaDatiGiocatori)
  if Pos('-',Nome) <> 0 then
   begin
    InviaAlClient(Athread,Fine,'Elimina il simbolo - dal nome!');
    Exit;
   end;

  // Disabilita l'aggiornamento del controllo visuale
  ListaUtenti.BeginUpdate;
  try
    // Visualizza nome e icona nell'apposito controllo
    if BC.Comando = UsaIcona then
     begin
      with ListaUtenti.ListBoxItems.Add do
       begin
        Strings.Add(Nome);
        ImageIndex := StrToInt(Bc.Messaggio);
        Strings.Add('0'); // Inizialmente il punteggio è sempre nullo
       end;
     end;
  finally
    // Mostra i cambiamenti nella lista degli utenti
    ListaUtenti.EndUpdate;
   end;

  // Crea un nuovo giocatore in fase di connessione
  GetMem(NuovoGiocatore, SizeOf(TGiocatore));

  NuovoGiocatore.CollegatoAlle := Now;
  NuovoGiocatore.Punteggio := 0;
  NuovoGiocatore.Thread := AThread;
  NuovoGiocatore.Nome := Nome;
  // Azzera la lista delle carte
  FillChar(NuovoGiocatore.ListaCarte, SizeOf(NuovoGiocatore.ListaCarte), 0);
  // Assegnalo ai dati cosi' possiamo gestirlo in seguito
  AThread.Data := TObject(NuovoGiocatore);
  // E prova ad aggiungerlo alla lista dei giocatori
  try
    Giocatori.LockList.Add(NuovoGiocatore);
  finally
    Giocatori.UnlockList;
   end;
  InviaDatiGiocatori;
end;
```

A connessione avvenuta tutte le richieste del giocatore fanno riferimento ad un thread specifico. Dopo aver ricevuto un pacchetto dati da un client collegato il server deve elaborarlo per determinare il comando impartito e agire di conseguenza richiamando apposite procedure:

```
procedure TFinestraServer.ServerExecute(AThread: TIdPeerThread);
var
  GiocatoreAttivo: PGiocatore;
  BC: TBloccoComunicazione;
  //G: TidPeerThread;
begin
  // Ricevi il comando da client
```

```
  AThread.Connection.ReadBuffer(BC, SizeOf(BC));
  // Determina chi è il giocatore corrente
  GiocatoreAttivo := PGiocatore(AThread.Data);
  case BC.Comando of
    // E' stata richiesta una nuova carta
    NuovaCarta:ServiNuovaCarta(GiocatoreAttivo,AThread);
    // Cambia la posta
    CambiaPosta:CambiaLaPosta(GiocatoreAttivo,BC.Messaggio);
    // Il giocatore si accontenta
    Sto: IlBancoGiocaContro(GiocatoreAttivo,AThread);
    // Invia a tutti i giocatori il messaggio di chat
    Chattata:Invia_A_Tutti_Il_Messaggio_Di(GiocatoreAttivo,BC.Messaggio);
    // Il giocatore vuole giocare una nuova partita
    NuovaPartita:AvviaNuovaPartita(GiocatoreAttivo,AThread,BC.Messaggio);
  end;
  InviaDatiGiocatori; // Aggiorna i dati relative ai giocatori
end;
```

Le procedure esaminate finora formano il nocciolo delle applicazioni client e server, ovviamente per avere un programma funzionale sono indispensabili molte altre procedure e funzioni. Per motivi di spazio verranno commentate solo quelle che presentato aspetti interessanti.

Terza parte

Funzioni complementari

La descrizione delle funzioni accessorie del programma Blackjack C/S rappresenta una sorta di manuale d'uso ed è corredata da porzioni di codice commentate al fine di mostrare le tecniche di programmazione di alcune funzioni aggiuntive come: l'avvio di una nuova partita, il calcolo del punteggio, il meccanismo di visualizzazione delle carte, l'invio di messaggi di testo, la disconnessione degli utenti da parte del server, la registrazione degli eventi, la visualizzazione delle informazioni di rete, la registrazione delle opzioni del server, la trasmissione dei messaggi di chat…

Avviare una nuova partita

Prima di iniziare una partita bisogna collegarsi al server, l'utente dispone di una comoda finestra per avviare la connessione. La finestra consente di scegliere un nome, un'icona (elemento puramente estetico), l'indirizzo e la porta del server. Durante la stessa connessione è possibile giocare un numero arbitrario di partite premendo il tasto contrassegnato dal testo "Nuova partita".

Alla pressione del pulsante Connetti è associato il seguente evento:

```
procedure
TFinestraConnessione.BitBtn1Click(Sender:
TObject);
begin
// E' stato specificato un nome?
  if NomeEdit.Text <> '' then
  begin
    // Cambia porta
    FinestraClient.PortaCorrente :=
PortaEdit.Text;
   // Chiama il metodo Connetti
    FinestraClient.Connetti(NomeEdit.Text,
ComboIcone.ItemIndex,
      HostEdit.Text, PortaEdit.Text)
  end
  else
    MessageDlg('Non puoi giocare senza dirmi
come ti chiami!', mtError, [mbOK], 0);
```

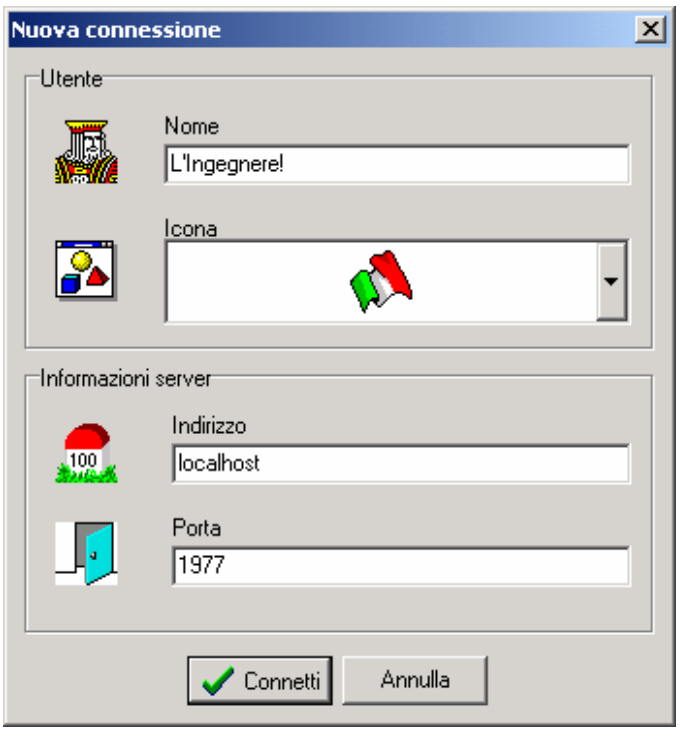


Figura 4 La finestra di connessione

L'utente può scegliere di disconnettersi manualmente o lasciare l'onere della disconnessione al server.

Chat

E' previsto un sistema di messaggistica, il server ha il compito di trasmettere a tutti i giocatori connessi i messaggi inviati da un giocatore:

```
Chattata:
     begin
      AggiornaLog(GiocatoreAttivo^.Nome+' ha scritto '+Bc.Messaggio);
    with Giocatori.LockList do
      try
        for i := 0 to Count - 1 do // itera tra i giocatori collegati
         begin
          DaMessaggiare := Items[i]; // prendi un giocatore
          G := DaMessaggiare.Thread; // Ottieni il suo thread
          InviaAlClient(G, Chattata, GiocatoreAttivo^.Nome+': '+bc.messaggio);
         end;
      finally
        Giocatori.UnlockList;
       end;
     end;
```

Il calcolo dei punteggio

Il server deve aggiornare il proprio punteggio e quello del giocatore per permettere un corretto svolgimento della partita. La funzione che determina il punteggio riceve in ingresso un array di carte giocate ed il numero di tali carte e restituisce un intero:

```
function CalcolaPunti(P: array of integer; CarteGiocate: integer): integer;
var
  I, Parziale, ContaAssi, FattiValere11: integer;
begin
  Parziale := 0;
  FattiValere11 := 0; // Numero di assi che valgono undici
  ContaAssi := 0; // Numero di assi nella sequenza di carte giocate
  for I := 0 to CarteGiocate - 1 do
  begin
    if P[I] <> 1 then Inc(Parziale, P[I]) // Se la carta non è un asso
    else
    begin
      if Parziale + 11 <= 21 then // Gestisci separatamente gli assi
      begin
        Inc(Parziale, 11);
        Inc(FattiValere11);
      end
      else
        Inc(Parziale, 1); // Evitiamo di “sballare”
      Inc(ContaAssi);
    end;
  end;
  // Se abbiamo fatto un errore di valutazione:
  if (FattiValere11 > 0) and (Parziale > 21) then Dec(Parziale, 10);
  if (CarteGiocate = 2) and (ContaAssi = 1) and (Parziale = 21) then
    CalcolaPunti := BlackJack
  else if Parziale <= 21 then CalcolaPunti := Parziale
  else if Parziale > 21 then CalcolaPunti := Sballato;
end;
```

Visualizzazione delle carte

Le carte sono contenute in un file di risorse “inglobato” nell’eseguibile. Una possibile alternativa consiste nell’usare la libreria a collegamento dinamico cards.dll distribuita con i sistemi Windows. Il server pesca una carta dal mazzo e la invia al giocatore. Il formato della carta è stabilito dalla funzione PescaCarta presente in BlackJackUnit:

```
function PescaCarta: string;
var
  Temp, Temp1: integer;
  Seme: string;
begin
  Temp := Random(13);
  if temp = 0 then temp := temp + 1 + random(12); // Aumenta di un numero
casuale
  Temp1 := Random(3);
  case temp1 of
    0: Seme := 'F'; 1: Seme := 'Q';2: Seme := 'C';3: Seme := 'P';
 end;
  PescaCarta := IntToStr(Temp) + '-' + Seme;end;
```

L’asso di cuori per esempio è rappresentato dalla stringa 1-C, il re di fiori da 13-F e così via… Il client ha il compito di decodificare il formato e mostrare la carta nella posizione corretta X,Y:

```
procedure TFinestraClient.MostraCarta(cartadamostrare: string; X,
  Y: integer);
var
```

```
  Carta: TCard;
  Seme: TCardSuit;
  SemeStr: string;
  Valore: integer;
begin
  // Il sever invia la carta nella forma valore-seme,dobbiamo estrarre i 2
parametri
  Valore := StrToInt(copy(CartaDaMostrare, 1, pos('-', CartaDaMostrare) - 1));
  SemeStr := Copy(CartaDaMostrare, pos('-', cartadamostrare) + 1, 1);
  if SemeStr = 'F' then seme := Fiori
  else if SemeStr = 'C' then Seme := Cuori
  else if SemeStr = 'Q' then seme := Quadri
  else if SemeStr = 'P' then seme := Picche
  else
    seme := Cuori; // Questo else non dovrebbe mai essere eseguito
  // Crea la carta
  Carta := TCard.Create(FinestraClient);// Instanzia una nuova carta
  with carta do
   begin
    Parent := FinestraClient; // Deve essere visualizzata sul tavolo verde
    Value := Valore;
    Suit := Seme;
    Left := X; // X e Y indicano la posizione sul tavolo
    Top := Y;
    Show; // Mostra la carta!
   end;
end;
```

Inviare messaggi di testo

Il server è una stazione di controllo, un eventuale operatore potrebbe voler contattare un giocatore o tutti i giocatori inviando loro un messaggio.

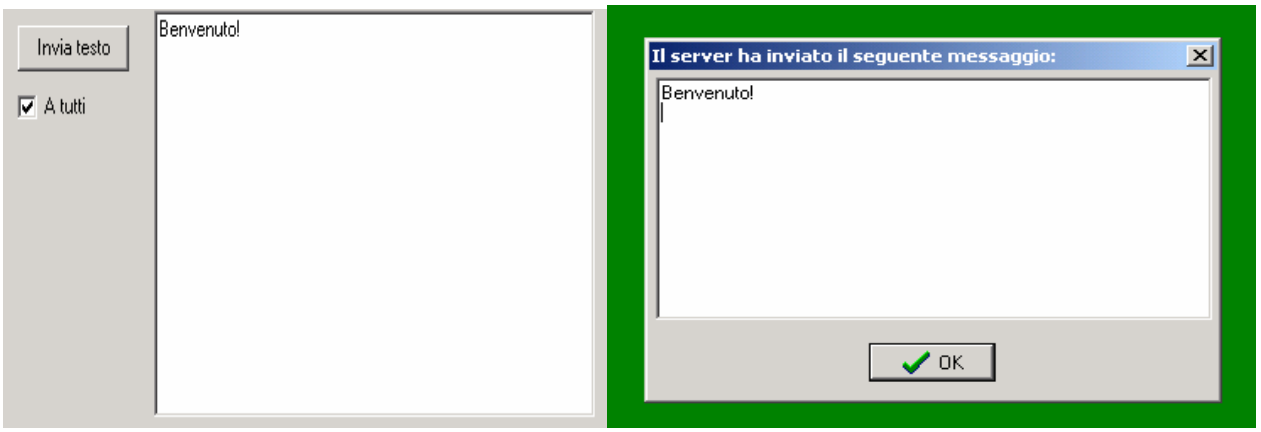


Figura 5 Il client riceve il testo inviato dal server

La procedura che invia un messaggio a uno o più giocatori:

```
procedure TFinestraServer.InviaMsgClick(Sender: TObject);
var
  i: integer;
  G: TidPeerThread;
  DaMessaggiare: PGiocatore;
```

```
begin
  if ATutti.Checked then // Il messaggio è per tutti?
   begin
    AggiornaLog('Ho inviato un messaggio a tutti');
    with Giocatori.LockList do
      try
        for i := 0 to Count - 1 do // itera tra i giocatori collegati
         begin
          DaMessaggiare := Items[i]; // prendi un giocatore
          G := DaMessaggiare.Thread; // Ottieni il thread
          InviaAlClient(G, MsgTesto, TestoDaInviare.Lines.CommaText);
         end;
      finally
        Giocatori.UnlockList;
       end;
   end
  else
   begin
    if ListaUtenti.ItemIndex = -1 then MessageDlg('Seleziona un utente dalla
lista',
        mtWarning, [mbOK], 0)
    else
      with Giocatori.LockList do
        try
          for i := 0 to Count - 1 do // itera tra i giocatori collegati
           begin
            DaMessaggiare := Items[i]; // prendi un giocatore
            G := DaMessaggiare.Thread; // Ottieni il suo thread
            if DaMessaggiare.Nome = ListaUtenti.ListBoxItems.Items
              [ListaUtenti.ItemIndex].Strings[0] then
             begin
              AggiornaLog('Ho inviato un messaggio a ' + DaMessaggiare.Nome);
              InviaAlClient(G, MsgTesto, TestoDaInviare.Lines.CommaText);
             end;
           end;
        finally
          Giocatori.UnlockList;
         end;end;end;
```

Dopo aver ricevuto un messaggio il client chiama la funzione VisualizzaMsg:

```
procedure TFinestraClient.VisualizzaMsg(M: string);
var
  FinestraMessaggio: TFinestraMsg;
begin
  FinestraMessaggio := TFinestraMsg.Create(FinestraClient);
  try
    FinestraMessaggio.MsgRicevuto.Lines.CommaText := M; // Assegna Testo
    FinestraMessaggio.ShowModal; // Mostra la finestra in figura 5
  Finally FinestraMessaggio.Free; end; end;
```

Disconnessione dei giocatori

La disconnessione degli utenti avviene per tre motivi diversi:
1) Se un server viene disattivato o chiuso scollega tutti gli utenti connessi;
2) Un operatore decide di terminare volontariamente una o più partite;
3) Il client si disconnette per problemi di rete o per volontà del giocatore;

Nel primo caso il server utilizza la procedura DisconnettiTutti:

```
procedure TFinestraServer.DisconnettiTutti;
var
  i: integer;
  G: TidPeerThread;
  DaDisconnettere: PGiocatore;
begin
  AggiornaLog('Avvio disconnessione...');
  // Invia il codice di disconnessione (Fine = 10):
  with Giocatori.LockList do
    try
      for i := 0 to Count - 1 do // itera tra i giocatori collegati
       begin
        DaDisconnettere := Items[i]; // prendi un giocatore
        G := DaDisconnettere.Thread; // Ottieni il thread
        if G <> nil then
          InviaAlClient(G, Fine, 'Il server ha chiuso la connessione...');
       end;
    finally
      Giocatori.UnlockList;
     end;
end;
```

Nel secondo caso l'operatore addetto al server deve selezionare un utente dalla lista, cliccare sul pulsante destro del mouse e scegliere l'opzione "Termina Partita".

Figura 6 E' possibile interrompere manualmente una partita

Il risultato viene raggiunto il modo simile a quanto visto per l'invio dei messaggi, la differenza sta nel comando inviato tramite InviaAlClient: Fine al posto di MsgTesto. Indipendentemente dal motivo della disconnessione il server esegue la procedura ServerDisconnect. Il giocatore disconnesso è identificato dal parametro AThread. E' importante sottolineare che la disconnessione è una delle operazioni che modifica l'interfaccia utente del server. Infatti il nome dell'utente deve essere rimosso dalla lista dei giocatori collegati e l'evento viene registrato nell'apposito controllo visuale. La registrazione delle informazioni include anche la durata del collegamento tra il giocatore disconnesso ed il server.

```
procedure TFinestraServer.ServerDisconnect(AThread: TIdPeerThread);
var
  GiocatoreAttivo: PGiocatore;

begin
  GiocatoreAttivo := PGiocatore(AThread.Data);
  AggiornaLog(GiocatoreAttivo^.Nome + ' (' + GiocatoreAttivo^.DNS + ')
disconnesso');
  AggiornaLog(GiocatoreAttivo^.Nome + ' èc rimasto collegato per ' +
```

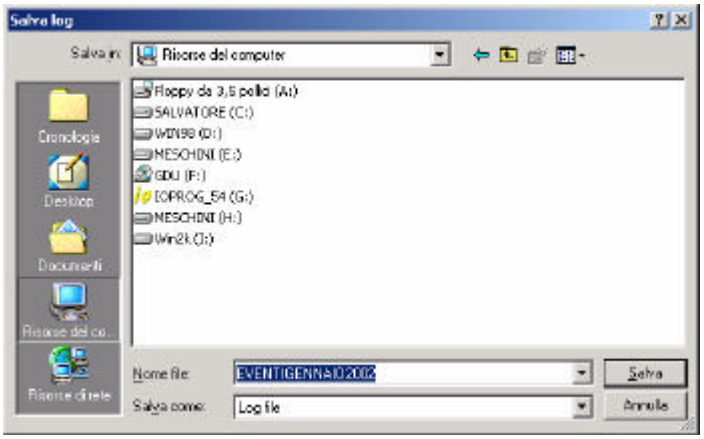
```
    TimeToStr(Now - GiocatoreAttivo^.CollegatoAlle));
  // Rimuovilo dalla lista utenti (quella con le icone)
  ListaUtenti.BeginUpdate;
  ListaUtenti.ListBoxItems.Delete(ListaUtenti.ListBoxItems.IndexInColumnOf(0,
    GiocatoreAttivo^.Nome));
  try
    Giocatori.LockList.Remove(GiocatoreAttivo);
  finally
    Giocatori.UnlockList;
    ListaUtenti.EndUpdate;
    FreeMem(GiocatoreAttivo);
    AThread.Data := nil;
   end;
end;
```

La registrazione degli eventi

Di solito un server è predisposto per la registrazione degli eventi. Il server del progetto Blackjack C/S non fa eccezione, ogni azione compiuta viene memorizzata in un controllo visuale. La lista degli eventi può essere salvata su file dall'operatore (vedi Fig. 6). Il risultato dell'operazione è un file di testo:

```
Evento
Ora
Descrizione
1
20.20.35
Server attivato
2
20.20.40
Sal (computer-di-sal) connesso
3
20.20.41
Sal ha avviato una nuova partita
4
20.20.41
4 di cuori inviato a Sal
5
20.20.41
Sal: 4 (+4)
6
20.20.43
Sal ha chiesto una carta
…
15
20.20.43
Ho estratto: 10 di fiori
16
20.20.43
10 di fiori notificato a Sal
17
20.20.43
Sal ha vinto
18
20.20.50
Sal (computer-di-sal) disconnesso
19
20.20.51
Sal è rimasto collegato per 0.00.10
```

Figura 7 Gli eventi possono essere salvati su file



Il metodo scelto per salvare la lista degli eventi su file non è certamente il più elegante ma mostra una tecnica *dirty* che aggira alcuni problemi di programmazione:

```
procedure TFinestraServer.SalvaEventi1Click(Sender: TObject);
var i:integer;
    Eventi:TStringList;
begin
// Crea una lista vuota di eventi
Eventi:=TStringList.Create;
// Mostra una finestra per la scelta del file
try
  // Aggiungi tutti gli eventi
  for i:=0 to ListaLog.ListBoxItems.Count-1 do
   Eventi.AddStrings(ListaLog.ListBoxItems.Items[i].Strings);
  // Se l'utente assegna un nome al file allora salva la lista
  if SalvaDlg.Execute then eventi.SaveToFile(SalvaDlg.FileName);
  finally
    Eventi.Free;
 end;
end;
```

Aggiornamento dati

Il client riceve dal server informazioni relative a tutti i giocatori connessi e le visualizza:

```
procedure TFinestraServer.InviaDatiGiocatori;
var i,j:Integer;
    Giocatore,Temp:PGiocatore;
    G:TIdPeerThread;
begin
with Giocatori.LockList do
      try
        for i := 0 to Count - 1 do // itera tra i giocatori collegati
         begin
          Giocatore := Items[i]; // prendi un giocatore
          G := Giocatore.Thread; // Ottieni il suo thread
          InviaAlClient(G,DatiInArrivo,inttostr(Count)); // Prepara il client
          for j:=0 to Count - 1 do
           begin
            Temp:=items[j];
            // Invia i dati
            InviaAlClient(G, Dati ,Temp^.Nome+'-'+inttostr(Temp^.Punteggio));
           end;
         end;
      finally
        Giocatori.UnlockList;
       end;
AggiornaLog('Dati inviati ai giocatori');
end;
```

Visualizzazione delle informazioni di rete

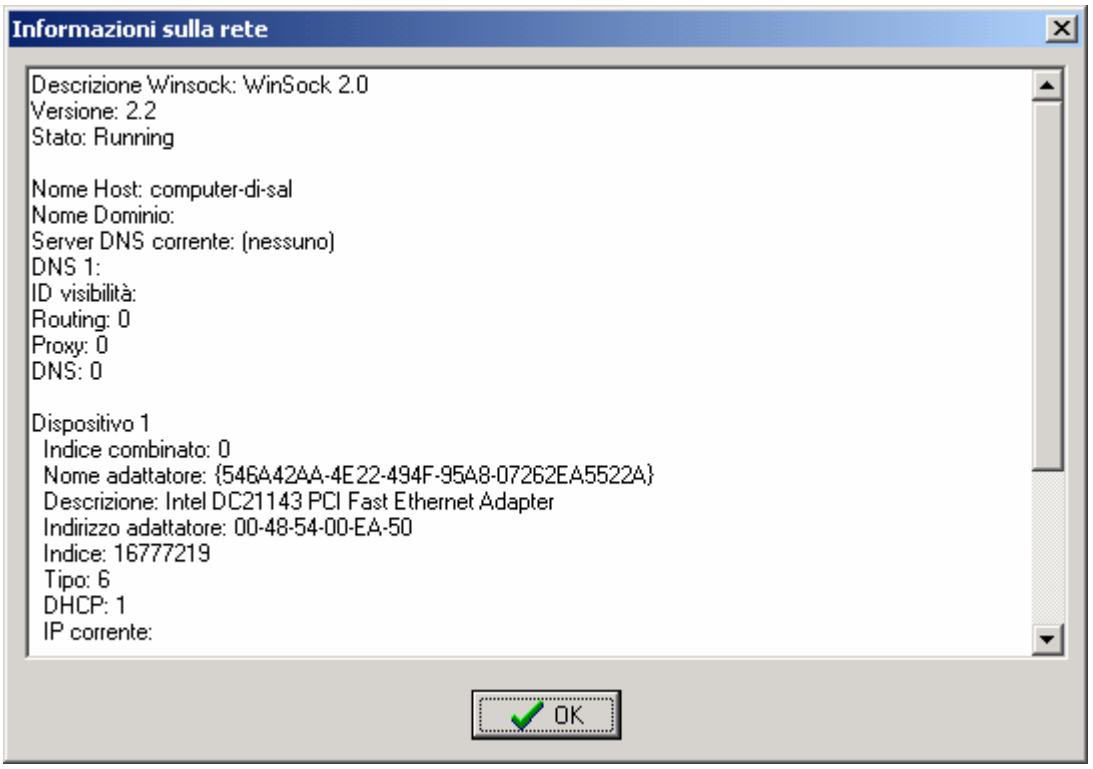
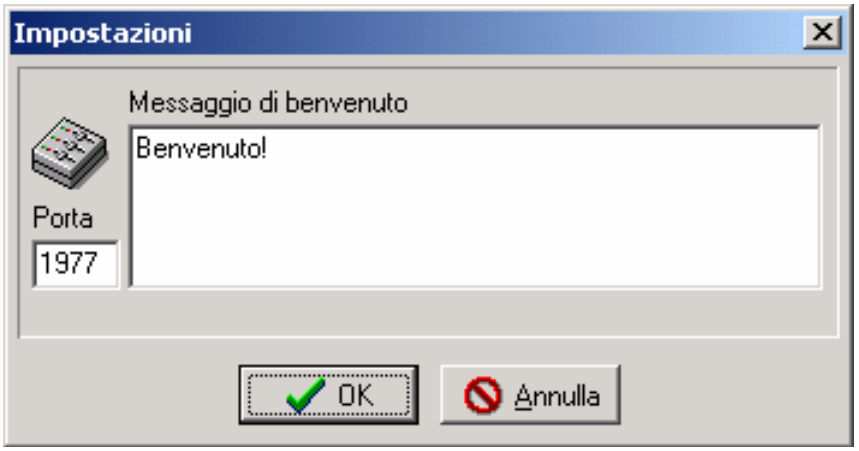


Figura 8 La finestra con le informazioni di rete

Le informazioni di rete sono state ottenute attraverso numerose chiamate a funzioni del sistema operativo. Le informazioni riguardano: lo stato dello stack Winsock, i dispositivi di rete presenti sulla macchina con vari dettagli, la configurazione della rete TCP/IP, etc. L'informazione più utile resta comunque l'indirizzo IP della macchina su cui il server (e/o il client) viene eseguito.

Configurazione del server

L'operatore ha la possibilità di scegliere la porta del server ed il messaggio di benvenuto da inviare ai giocatori. Le opzioni vengono automaticamente caricate durante la fase di avvio del server dal file server.ini (se presente).



```
procedure TFinestraServer.CaricaConfig; // CARICA LA CONFIGURAZIONE
var
  FileConfig: TIniFile;
  PortaTemp, MsgBenvenuto: string;
begin
  FileConfig := TIniFile.Create(ExtractFilePath(Application.exeName) +
'server.ini');
  try
    PortaTemp := FileConfig.ReadString('Setup', 'Porta', '1977');
    MsgBenvenuto := FileConfig.ReadString('Setup', 'Benvenuto', Benvenuto);
  finally
    FileConfig.Free;
   end;
  try
    // In caso di errori la porta viene impostata a 1977
    if ((StrToInt(PortaTemp) > 0) and (StrToInt(PortaTemp) < 65535)) then
      PortaDefault := StrToInt(PortaTemp);
  except
    on EConvertError do PortaDefault := 1977;
   end;
  Benvenuto := MsgBenvenuto;
  // Rendi effettive le scelte (porta e msg di benvenuto)
  Server.DefaultPort := PortaDefault;
  TestoDaInviare.Lines.Clear;
  TestoDaInviare.Lines.Add(Benvenuto);
  // Mostra indirizzo e porta del server (fondamentale per la connessione dei
client)
  BarraStato.Panels[1].Text := 'Indirizzo IP del server: ' +
    OttieniIp(NomeDominio) + ' Porta: ' + IntToStr(PortaDefault);
end;

procedure TConfig.ConfermaClick(Sender: TObject); // SALVA LA CONFIGURAZIONE
var
  FileConfig: TIniFile;
begin
  // Crea il file se necessario nella directory corrente
  FileConfig := TIniFile.Create(ExtractFilePath(Application.exeName) + 'server.ini');
  try
    // Verifica se la porta è valida
    if ((StrToInt(Porta.Text) > 0) and (StrToInt(Porta.Text) < 65535)) then
      FileConfig.WriteString('Setup', 'Porta', Porta.Text)
    else
      FileConfig.WriteString('Setup', 'Porta', '1977');
    FileConfig.WriteString('Setup', 'Benvenuto', Messaggio.Text);
  finally
    FileConfig.Free;
  end;end;
```