ADPnetwork: una rete per Amiga

di Andrea de Prisco

seconda puntata

Dopo la propedeutica introduzione su ADPnetwork fatta lo scorso mese, in questa seconda puntata analizzeremo più dettagliatamente il funzionamento dei processi più importanti e mostreremo lo schema di collegamento delle macchine, attualmente effettuato per mezzo della porta seriale

Bisogna dire una cosa: se Amiga non fosse stato un computer multitasking, realizzare una rete come ADPnetwork sarebbe stato davvero difficile. Proprio ieri mattina, ad esempio, ho iniziato a studiare la struttura di un potenziale software di gestione di un microcontroller capace di gestire in parallelismo simulato un certo numero di eventi assolutamente asincroni (quindi indipendenti) tra loro. Nonostante alla fine abbia avuto io la meglio, la mia mente s'era così abituata a ragionare in termini di processi paralleli (e indipendenti anch'essi) che le maledizioni inoltrate a quel povero microcontroller ormai nessuno riusciva più a contarle. Fortunatamente non dovremo aspettare ancora molto (almeno me lo auguro) prima di vedere i primi processori dotati di linguaggio macchina multitasking, anche se realizzato a semplici colpi di time sharing. Provate ad immaginare, ad esempio, cosa significa per un singolo programma controllare eventi differenti senza mai effettuare attese attive a scapito di altre operazioni da portare a termine: ad esempio un bel programma che riceve flussi di dati da due canali, elaborando gli eventuali dati provenienti dal canale 1 e, simultaneamente, redirigere l'input del canale 2 su un terzo canale d'uscita. Può succedere ad esempio che mentre dal canale 1 arrivano dati da elaborare sul canale 2 arrivino dati da ridirigere sul canale 3 e tutte le operazioni debbano essere compiute molto velocemente onde evitare perdite di dati da riempimenti di buffer.

Certo, in informatica, tutto quello che si può ben dire si può ben fare, ma volete mettere una soluzione al problema realizzata con due o [illegible] processi cooperanti quanto è più elegante e raffinata di una soluzione monotask capace di andare avanti solo ed esclusivamente a colpi di interrupt e variabili globali?

Fine dello sfogo.

Riassunto della puntata precedente

ADPnetwork, lo ripetiamo per chi si fosse sintonizzato solo ora, adotta uno schema di funzionamento «circolare» in cui ogni macchina ha un nodo precedente, dal quale riceve il flusso dei dati, e uno successivo al quale trasmette, o ritrasmette, dati. Ogni macchina analizzerà i dati in arrivo dalla rete e dovrà essere in grado di riconoscere messaggi per sé da inoltrare agli opportuni server, oppure da rimettere in circolo non riconoscendosi come destinatario. In questo modo è sia possibile che qualsiasi mac-

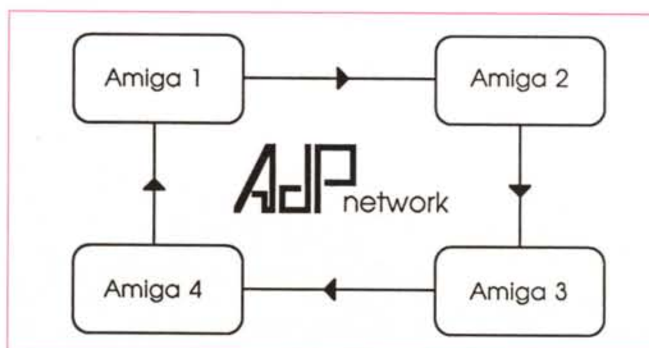

*Figura 1
Quattro Amiga collegati in rete via ADPnetwork.*

china dialoghi con qualsiasi altra macchina della rete, sia che in ogni istante più macchine effettuino operazioni sulla rete. Facendo un rapido esempio, se colleghiamo (figura 1) quattro macchine in rete attraverso ADPnetwork (la 1 con la 2, la 2 con la 3, la 3 con la 4 e quest'ultima con la 1) la macchina 2 per dialogare con la 4 passerà attraverso la 3 così come quest'ultima per «parlare» con la 2 passerà attraverso la 4 e la 1. Analogamente è possibile che CONTEMPORANEAMENTE la macchina 1 esegua un'operazione sulla 2 e che la 3 esegua una qualsiasi altra operazione sulla macchina 4. Questo grazie al fatto che la struttura di comunicazione è solo apparentemente condivisa da tutte le macchine: in realtà ogni nodo è banalmente proprietario del collegamento fisico con la macchina successiva, tutto qui.

L'attuale release funzionante di ADPnetwork, la 3.0, permette a qualsiasi processo in esecuzione su qualunque macchina di inviare messaggi a qualsiasi altro processo in esecuzione su qualsivoglia altra macchina collegata alla rete. Ogni messaggio può essere di lunghezza arbitraria e per inoltrarlo via rete il processo mittente dovrà naturalmente specificare in nodo destinatario, la porta mttb esistente su quel nodo (creata cioè dal processo destinatario) e consegnare il messaggio al software di rete. Sarà poi questo che, impacchettandolo opportunamente in frame di rete ed utilizzando l'interfaccia d'uscita verso la macchina «successiva» farà in modo (naturalmente con la complicità di tutti i processi di rete di tutte le macchine «attraversate») che il messaggio giunga a destinazione e sulla giusta porta. Se una macchina decide di uscire dalla rete, basta che sconnetta il suo ingresso e la sua uscita e li colleghi tra di loro: in questo caso si ripristina automaticamente il collegamento circolare e tutte le rimanenti macchine possono continuare ad adoperare la rete.

Packer e Spacker

Sempre lo scorso mese vi ho anticipato che i due processi Packer e Spacker del Software di Rete (SDR), front-end verso AmigaDOS, si preoccupano rispettivamente di formare i frame di rete da spedire e di ricostruire i messaggi in arrivo man mano che giungono i vari frame da altre macchine. È ovvio inoltre che impacchettamento e spacchettamento dei messaggi deve essere una coppia di funzioni non visibili ai processi AmigaDOS, ai quali importa solo di spedire un messaggio ad una determinata macchina e riceverne da altre. Oltre a questo, il processo Spacker deve essere in grado di mantenere più liste degli arrivi, dal momento che possono arrivare più richieste da più macchine i cui frame, come detto, non sono regolamentati da un ordine di arrivo identico a quello delle rispettive partenze. I frame di rete viaggiano infatti in maniera indipendente l'uno dall'altro e, dato che ogni macchina è autorizzata ad annullare frame di transito contenenti errori di trasmissione e ad inserire tra un frame ed un altro in transito anche propri frame per altre macchine, capirete bene che la conquista dell'arrivo per questi sarà quantomeno faticata. Pensate ad esempio ad una rete di trasporto merci basata su ferrovia e navigazione (ad esempio un bel collegamento con le isole). Dalle stazioni di partenza vengono formati vari convogli per le relative destinazioni, spezzati però in più parti quando si tratta di attraversare tragitti marini (un intero treno, per lungo, non entra in una nave...). Immaginate inoltre (per rendere l'esempio più vicino al traffico su rete) che per motivi di ottimizzazione ogni volta che c'è da caricare una nave di carri ferroviari si cerchi sempre di riempire al massimo ogni nave, prelevando carri anche da convogli diversi. Ovviamente, però, a destinazione i convogli dovranno arrivare sempre e comunque con tutti i vagoni al loro posto e nel medesimo ordine della partenza, dunque le stazioni d'arrivo dovranno raccogliere man mano i vagoni che arrivano e ricomporre i convogli prima di strillare «in arrivo sul terzo binario...».

Eh, già! un solo binario non basta proprio: può succedere che arrivino prima un po' di pezzi del treno 208, poi tre vagoni del treno 665, poi ancora del treno 208, poi un carro del 256...

Spacker funziona proprio allo stesso modo. Arrivato un qualsiasi frame, la prima operazione che compie è vedere se già ha inizializzato una lista d'arrivo relativa a quel determinato messaggio. L'identificazione unica del messaggio è riportata all'interno di ogni frame: basta guardare il campo mittente e il campo MsgID, incrementato dal mittente stesso ogni nuova spedizione. Tale informazione è presente anche nelle liste d'attesa dello Spacker. O, meglio, è presente se la lista era già stata istanziata prece-

ADPnetwork allo SMAU. Due Amiga collegati in rete visualizzavano le immagini di AMIGAllery memorizzate sull'hard disk della macchina a sinistra.

dentemente oppure viene inizializzata se si tratta del primo frame relativo a quel messaggio. Altre informazioni utili che possiamo prendere dal primo frame arrivato sono la lunghezza effettiva del messaggio originale (rapportato all'esempio dei treni, la lunghezza del convoglio al momento della partenza) in modo da allocare una giusta quantità memoria per ricostruire il messaggio, e la posizione relativa del frame arrivato all'interno del messaggio originario. Allocata, dunque, la giusta quantità di memoria, il corpo del frame arrivato viene ricopiato nella giusta posizione all'interno del messaggio

ADPmttb 2.0 terza parte

Le nuove funzioni ADPmttb (cfr. MC n. 88) presentate questo mese riguardano la spedizione e ricezione di messaggi di tipo stringa e il controllo del non determinismo. La funzione che ci permetterà di spedire una qualsiasi stringa (null terminated) ad un altro processo è la Send. Accetta tre soli parametri e precisamente il modo di spedizione (MODE_SYNC, MODE_ASYNC o il nuovo MODE_RVE), la porta mttb su cui spedire il messaggio e il messaggio stesso ovvero il puntatore alla stringa da spedire. Analogamente, la funzione Receive permette di aspettare (o non aspettare) una stringa in arrivo su una porta. I parametri della Receive sono ancora 3 ovvero il modo di ricezione (MODE_WAIT, MODE_NOWAIT o, ancora, MODE_RVE), la porta da cui prelevare l'eventuale messaggio e una variabile stringa per contenere il messaggio in arrivo. Per i quattro modi già conosciuti vi rimando all'articolo pubblicato su MC di settembre. Il nuovo MODE_RVE (utilizzabile solo con Send e Receive) implementa la forma di comunicazione a rendez-vous esteso. Utilizzando questo modo (che deve essere impostato nella Send e nella Receive implicate) chi effettua la Send ordina al processo che aspetta sulla Receive di spedirgli il messaggio. Si ha, in pratica, un capovolgimento dei ruoli con la differenza però che il processo che esegue la Receive sta lì ad aspettare che qualche altro processo lo interroghi e la risposta è spedita effettivamente all'autore della Send chiunque esso sia. Per capire meglio, facciamo un piccolo esempio: il processo Pippo crea una sua porta Pluto per spedire la data odierna a chi gliela chiede. Al suo interno troveremo una istruzione (magari all'interno di un loop) di questo tipo:

Receive(MODE_RVE, "Pluto", DataDiOggi);

dove DataDiOggi è una stringa contenente appunto la data. Qualsiasi processo può eseguire a questo punto una:

Send(MODE_RVE, "Pluto", variabile);

per ottenere al suo ritorno una copia di DataDiOggi nella sua 'variabile'.

Le rimanenti due funzioni, MultiReceive e MultiWait permettono di attendere eventi su più porte (massimo 5). Con la prima potremo effettivamente ricevere fino a 5 messaggi contemporaneamente (sempre di tipo stringa null terminated) con la seconda, di uso più generale, semplicemente aspettare su fino a 5 porte l'arrivo di un qualsiasi messaggio che poi preleveremo (se lo riterremo opportuno) con la funzione apposita (Receive, ReceiveBlock, ReceiveChar, ReceivePointer, ecc. ecc.). La sintassi in tutt'e due i casi è molto semplice. Per la MultiReceive dovremo indicare innanzitutto il modo di ricezione (il solito MODE_WAIT o MODE_NOWAIT), il numero di porte su cui operare e poi una sequenza di coppie «porta, variabile» come nelle normali Receive. Ad esempio con la linea:

MultiReceive(MODE_WAIT, 3, "Pippo", var1, "Pluto", var2, "Minnie", var3);

aspetteremo su almeno una delle tre porte citate messaggi di tipo stringa da porre nelle variabili indicate. Da notare che se tutt'e tre le porte contengono messaggi, al ritorno dalla funzione troveremo in ogni variabile il relativo messaggio arrivato, se arriva un solo messaggio ne troveremo uno nella variabile corrispondente (stringa vuota nelle altre) e così via per ogni possibile combinazione: è una vera e propria lettura parallela delle n porte indicate (con 'n', ripeto, minore o uguale a 5). Indicando MODE_NOWAIT come primo parametro, avremo l'effetto di ritrovare tutte stringhe vuote se al momento della chiamata tutte le porte indicate non contengono messaggi.

La sintassi della MultiWait è un tantino più semplice: si indica solo il numero delle porte su cui operare e la lista delle porte interessate. Da questa funzione si torna non appena una o più porte presentano messaggi in arrivo.

Tutto qui.

```
/************************************
*                                   *
*                                   *
*         A D P m t t b   2.0       *
*                                   *
*        MultiTasking  ToolBox      *
*            (terza parte)          *
*                                   *
*        ---------------------      *
*        (c) 1989  ADPsoftware      *
*        ---------------------      *
*                                   *
************************************/

int  Send(UBYTE,char *,char *);
int  Receive(UBYTE,char *,char *);
int   MultiReceive(UBYTE,int,char *,char *,char  *,char *,char *,char *,char *,
char *,char *,char *);
int  MultiWait(int,char *,char *,char *,char *,char *);

/*********************************
*                                *
*          S   E   N   D         *
*                                *
*********************************/

#include "MTTB.h"

Send(mode,porta,msg)
UBYTE  mode;
char   *msg,*porta;
{
int  result;
char portaR[20];
if (mode & MODE_RVE)
   {
   sprintf(portaR,"RPof%lx",FindTask(0));
   NewPort(portaR);
   if ((result = SendBlock(MODE_ASYNC,porta,portaR,20))>=0)
     result = ReceiveBlock(MODE_WAIT,portaR,msg);
   EndPort(portaR);
   return(result);
   }
else return(SendBlock(mode,porta,msg,strlen(msg)+1));
}

/*********************************
*                                *
*     R  E  C  E  I  V  E        *
*                                *
*********************************/

Receive(mode,porta,vtg)
UBYTE  mode;
char   *porta, *vtg;
{
int  l;
char portaR[20];

if (mode & MODE_RVE)
   {
   if ((l=ReceiveBlock(MODE_WAIT,porta,portaR)) < 0) return(l);
   return(SendBlock(MODE_ASYNC,portaR,vtg,strlen(vtg)+1));
   }
else
   {
   l=ReceiveBlock(mode,porta,vtg);
   if (!l) strcpy(vtg,"");
   return(l);
   }
}

/*****************************************
*                                        *
*   M  U  L  T  I  R  E  C  E  I  V  E   *
*                                        *
*****************************************/

MultiReceive(mode,count,p0,v0,p1,v1,p2,v2,p3,v3,p4,v4)
UBYTE mode;
char *p0,*v0,*p1,*v1,*p2,*v2,*p3,*v3,*p4,*v4;
```

in arrivo. Relativamente a quella lista d'attesa è incrementato della giusta «dose» il campo «Arrived» che contiene costantemente la quantità di byte effettivamente arrivati a destinazione. Non appena tale campo raggiunge l'effettiva lunghezza del messaggio (e può succedere anche subito, dopo il primo frame, se esso viaggiava effettivamente su un solo pacchetto) il messaggio è considerato arrivato in toto, inoltrato all'effettiva porta destinataria creata da un processo in esecuzione su quella macchina occupata per la ricostruzione.

Sender e Dispatcher

Il processo Sender, del quale data la sua estrema semplicità non è stato preparato un diagramma di flusso, si occupa di effettuare le spedizioni di frame. Frame provenienti dalla stessa macchina, quindi contenenti richieste o risposte da recapitare ad altri nodi appositamente «impacchettati» dal processo Packer, oppure frame di transito scartati dal processo Dispatcher che non li ha riconosciuti come propri. Attualmente i frame in via di spedizione vengono accodati tutti sulla stessa porta, sia quelli che provengono dal Dispatcher che quelli provenienti dal Packer. In altre parole non si è voluto stabilire una priorità tra frame in transito e frame in partenza ed effettivamente chi dei due processi esegue per primo la Send verrà per primo servito dal Sender.

Naturalmente nulla vieta di aggiungere una o più porte al processo Sender in modo da poter scegliere di volta in volta cosa inviare per prima, secondo uno

```
{
int     p,i,t=0;
UBYTE   pp[5];
ULONG   mask=0;
char   *vtg[5],*porta[5];
struct  MsgPort *port[5];
struct  adp_message *adpmsg;

if ((mode & MODE_WAIT || mode & MODE_NOWAIT)==0) return(OP_FAIL);

porta[0] = p0;
porta[1] = p1;
porta[2] = p2;
porta[3] = p3;
porta[4] = p4;
vtg[0]   = v0;
vtg[1]   = v1;
vtg[2]   = v2;
vtg[3]   = v3;
vtg[4]   = v4;

for (i=0;i<count;i++)
    {
    Forbid();
       port[i] = (struct MsgPort *)FindPort(porta[i]);
       strcpy (vtg[i],"");
       if (port[i] != 0 && port[i]->mp_MsgList.lh_Head->ln_Succ)
          {
          adpmsg = (struct adp_message *)GetMsg(port[i]);
          strcpy(vtg[i],adpmsg->testo);t++;
          if (adpmsg->mode & MODE_SYNC) ReplyMsg(adpmsg);
          else FreeMem(adpmsg,sizeof(struct adp_message)+adpmsg->len);
          }
    Permit();
    if (port[i] != 0)
       {
       pp[i] = port[i]->mp_SigBit;
       mask |= 1<<pp[i];
       }
    }


while ( t == 0 && (mode & MODE_WAIT) )
      {
      p = Wait( mask );

      for (i=0;i<count;i++)
          if ((port[i] != 0) && (p & (1<<pp[i])))
             if (adpmsg = (struct adp_message *)GetMsg(port[i]))
                {
                strcpy(vtg[i],adpmsg->testo);t++;
                if (adpmsg->mode & MODE_SYNC) ReplyMsg(adpmsg);
                else FreeMem(adpmsg,sizeof(struct adp_message)+adpmsg->len);
                }
      }
return(t);
}


 /****************************************
 *                                       *
 *        M U L T I W A I T              *
 *                                       *
 ****************************************/

MultiWait(count,p0,p1,p2,p3,p4)
char  *p0,*p1,*p2,*p3,*p4;

 {
 int     p,i,t=0;
 UBYTE   pp[5];
 ULONG   mask=0;
 char   *porta[5];
 struct  MsgPort *port[5];

 porta[0] = p0;
 porta[1] = p1;
 porta[2] = p2;
 porta[3] = p3;
 porta[4] = p4;

 for (i=0;i<count;i++)
     {
     Forbid();
        port[i] = (struct MsgPort *)FindPort(porta[i]);
        if (port[i] != 0  &&  port[i]->mp_MsgList.lh_Head->ln_Succ) t++;
     Permit();
     if (port[i] != 0)
        {
        pp[i] = port[i]->mp_SigBit;
        mask |= 1<<pp[i];
        }
     }


 while ( t == 0 )
       {
       p = Wait( mask );

       for (i=0;i<count;i++)
           if ((port[i] != 0)  &&  (p & (1<<pp[i])))
              if (port[i]->mp_MsgList.lh_Head->ln_Succ) t++;
       }
 return(t);
 }
```

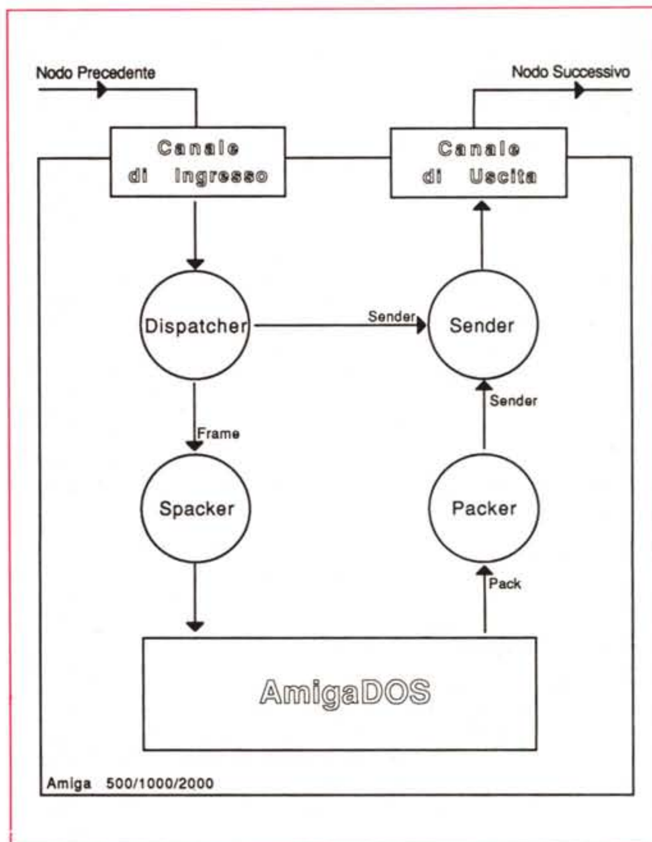

Figura 2 - I quattro processi di base di ADPnetwørk.

schema di priorità, volendo, variabile dinamicamente. Ad esempio si potrebbero favorire i frame in transito, dal momento che compongono una operazione iniziata sicuramente prima dell'operazione in corso sulla nostra macchina. Oppure si potrebbe stabilire di prendere un frame per porta e così intervallare frame in transito con frame in partenza senza mai favorire nessuna opportunità. Ancora, potremmo stabilire la priorità delle singole macchine in rete in modo da favorire determinate postazioni che eseguono lavori più urgenti di altre.

Queste sono comunque tutte scelte che potremo valutare meglio solo quando saremo prossimi ad una release «abbastanza definitiva» di ADPnetwork (il progetto, sebbene funzionante, è in continuo sviluppo per ottimizzare quanto più è possibile tutto l'ottimizzabile), testando così il comportamento in rete dei prodotti software di maggior interesse (che, fortunatamente per gli utenti e sfortunatamente per noi, non sono affatto pochi...).

Per quel che riguarda il processo Dispatcher, dando uno sguardo al suo diagramma di flusso, possiamo notare come sia anch'esso concettualmente molto semplice. Il suo lavoro è quello di aspettare sul canale di ingresso della rete l'arrivo di un frame. Arrivato il frame, la prima operazione che compie è controllare se il mittente è uguale al nome dello stesso nodo su cui gira il processo. In caso affermativo, infatti, ciò significa semplicemente che il frame ha fatto tutto il giro della struttura ad anello (passando per tutte le macchine in rete) con conseguente deduzione che il destinatario del messaggio in transito non esiste. Quando si verifica una situazione del genere ovviamente l'SDR invia un apposito messaggio d'errore al processo (in esecuzione sulla stessa macchina) che aveva richiesto un'operazione su una macchina inesistente. Se invece il mittente del messagio è diverso dal nome del nodo in questione, il secondo test che effettua il Dispatcher è naturalmente sul destinatario. In questo modo smista i frame per altre macchine direttamente al Sender e i frame per quel nodo al processo Spacker che provvederà a ricostruire il messaggio originario man mano che arrivano i vari frame.

Gli altri Processi

ADPnetwork, come già detto più volte lo scorso mese, non si ferma certo qui. Esistono infatti altri due processi di importanza strategica che rendono la rete sufficientemente fault tolerant. La descrizione fatta finora praticamente riguarda la release 2.0 che al minimo

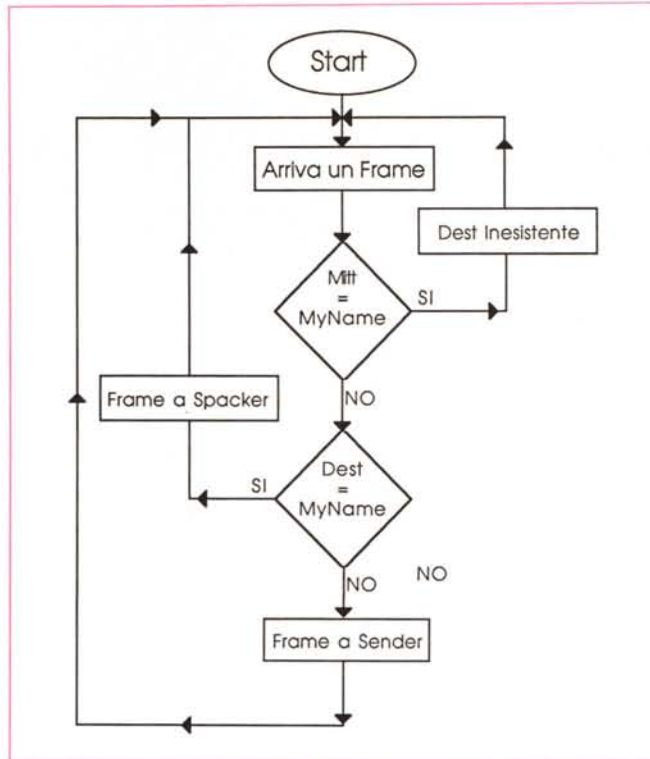

Diagramma di flusso del processo Dispatcher.

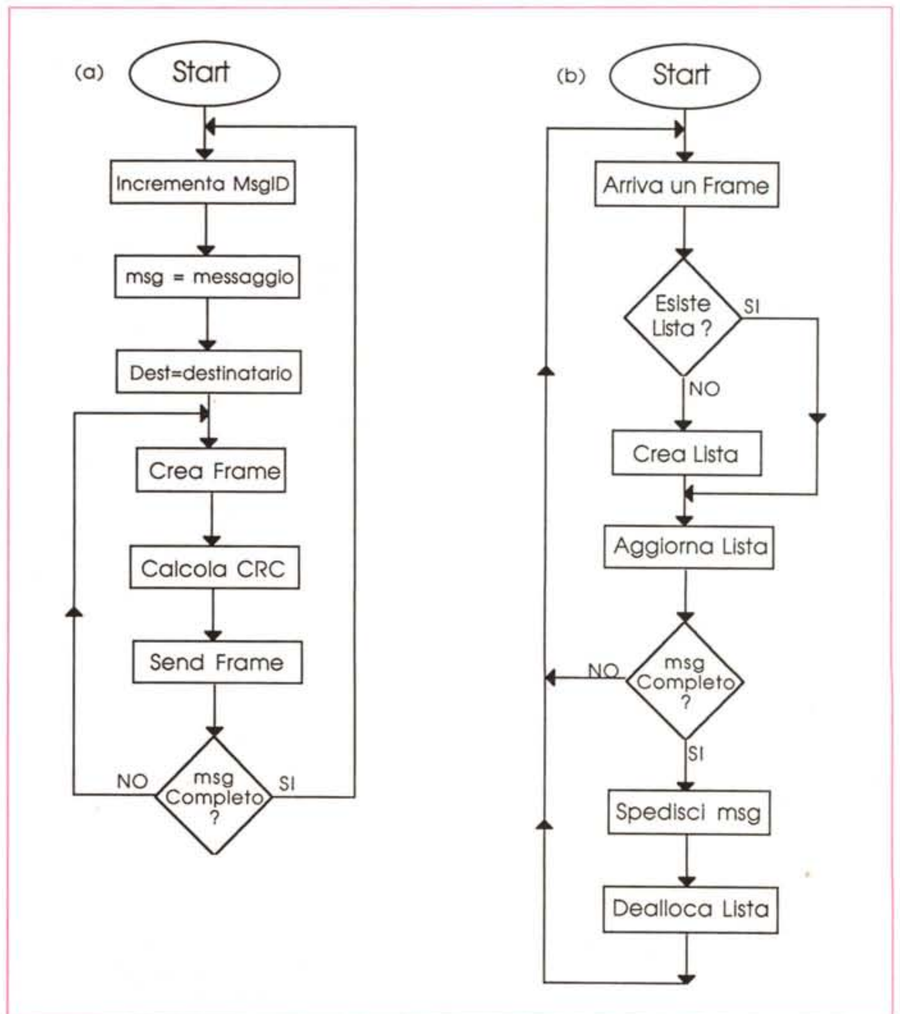

Diagramma di flusso dei processi Packer (a) e Spacker (b).

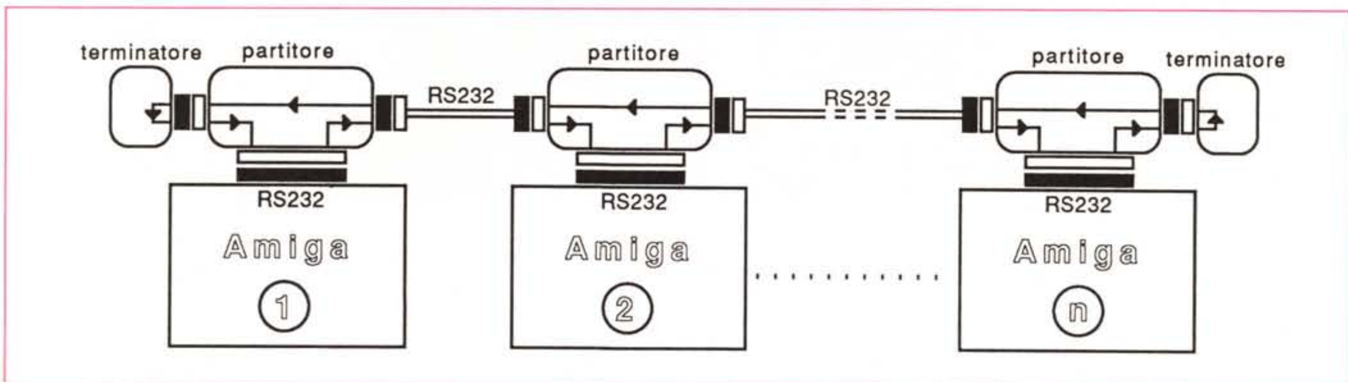

Figura 3 - Schema di collegamento di «n» Amiga attraverso la porta seriale.

errore di trasmissione dava forfait costringendo ad abortire le operazioni in esecuzione «coinvolte nell'intoppo».

Il processo Packer, in realtà, terminata la fase di impacchettamento del messaggio non butta via quest'ultimo ma lo passa così com'è al processo Replay. Contemporaneamente avvisa un altro processo, Timer, di avvertire il processo Replay dopo un prefissato intervallo di tempo. Ricevuto tale segnale di sveglia dal Timer, il processo Replay controlla quali frame sono giunti a destinazione e quali no, provvedendo a rispedire quelli «persi per strada». Ma come fa il processo Replay a stabilire quali frame sono arrivati e quali no?

Semplice: il Dispatcher della macchina destinataria, man mano che riceve i frame per quel nodo, provvede a reinviare un minipacchetto di ACK per ogni frame passato allo Spacker. Replay tiene nota degli ACK ricevuti e, conseguentemente, di quelli non ricevuti potendo così «dedurre» cosa manca al destinatario. Naturalmente se allo scadere del timeout tutti gli ACK erano tornati indietro, Replay non fa assolutamente nulla, se non deallocare la zona di memoria contenente copia del messaggio spedito. Bello, no?

Utilizziamo la porta seriale

La figura 1 mostra, come detto, lo schema di collegamento circolare di ADPnetwork. Tale collegamento è dettato dallo stesso Software di Rete che lavora «sapendo» che le macchine sono collegate in quella maniera. Tutto ciò credo sia fin troppo chiaro e già da un pezzo. Al fine di testare il corretto funzionamento dei vari processi in esecuzione sulle macchine, a scopo puramente sperimentale, sin dai primi passi è stata utilizzata l'interfaccia seriale presente su ogni macchina. Per essere più precisi, ogni Amiga dispone di due interfacce separate, ed utilizzabili separatamente, una di uscita ed una di ingresso. La prima la collegheremo alla macchina successiva, l'altra alla macchina precedente.

In questa situazione, viaggiando ai canonici 19200 bps, la rete funziona bene anche se molto lentamente per operazioni impegnative. Diciamo che la velocità ottenuta è circa un quarto della velocità di un'unità per microfloppy presente su ogni Amiga. Dunque se dobbiamo trasferire grossi file andiamoci pure a prendere un bel caffè, mentre per operazioni più rapide, se non siamo troppo abituati a velocità di altri sistemi possiamo anche chiudere un occhio (...una narice e un orecchio) e accontentarci della sola rete software.

Detto questo, onde evitare di realizzare cavi di forma stellare per connettere più macchine tra di loro (ognuna delle quali, come si sa, dotata di connettore unico della seriale) è stato ideato lo schema di collegamento di figura 3 utilizzante su ogni Amiga un banale partitore di ingresso e uscita, collegando le macchine intermedie con normali cavi seriali e dotando le macchine all'estremità di un opportuno terminatore. Da notare che, pur assomigliando ad un collegamento su bus, la comunicazione avviene attraverso la struttura ad anello di figura 1 e ciò può essere facilmente verificato seguendo, sempre in figura 3, le frecce presenti sui collegamenti elettrici che indicano la direzione del flusso di dati. Volendo aggiungere una nuova macchina, basterà utilizzare un nuovo partitore e un nuovo cavo seriale standard ed affettuare l'inserimento in qualsiasi punto; la struttura ad anello è sempre rispettata.

Conclusioni

Nel cedere (momentaneamente...) la parola a Marco Ciuchini e Andrea Suatoni che a partire dal prossimo numero ci «canteranno» del loro Net-Handler per la rete, voglio spendere due parole sul futuro di ADPnetwork.

L'utilizzo di tale Software di Rete attraverso la porta seriale è effettivamente troppo limitativo. E se qualcuno pensa di far viaggiare la stessa a velocità più elevate (in teoria potrebbe andare anche ad oltre 100 Kbaud) sappia che anche ipotizzando di riuscire a raggiungere tali valori di velocità non possiamo utilizzare gli Amiga collegati in rete SOLO per la rete. Le singole postazioni devono infatti continuare a funzionare come normali Amiga su cui caricare, assieme all'SDR anche le applicazioni per lavorare.

Né, evidentemente, alla Commodore pensavano di utilizzare la seriale per scopi diversi dal semplice interfacciamento con periferiche, come invece succede nei Macintosh capaci di far viaggiare (senza battere ciglio) la loro seriale a oltre 200 Kbaud, implementando su di essa (sin dalla nascita del Mac) la rete AppleTalk.

L'alternativa è naturalmente unica, ed è possibile (ancora una volta) grazie alla struttura particolamente aperta di Amiga: realizzare una scheda hardware da attaccare agli Amiga in modo da ottenere il duplice vantaggio di aumentare la velocità di trasferimento tra le macchine (utilizzando una forma di interfacciamento più evoluta) e demandare all'elettronica il riconoscimento dei frame. In questa maniera le macchine non interessate ad un trasferimento ma funzionanti, in quel momento, solo come ponte per la comunicazione non vengono affatto rallentate cone invece accade utilizzando la seriale e le risorse interne di calcolo. Comunque di questo e di altri aspetti del futuro di ADPnetwork avremo modo di parlarne più in là. Arrivederci...

MC