ADPmttb 2.0 *seconda parte*

Sul numero scorso, per i soliti problemi di spazio (tiranno), abbiamo pubblicato solo una versione ridotta dell'ADPmttb. Approfittando del fatto che ancora non pubblichiamo i listati di ADPnetwork, sfruttiamo un po' di spazio di questo articolo per «rilasciarvi» altre 5 routine dell'ADPmttb. Si tratta di funzioni per controllare lo stato delle porte create con la funzione NewPort mostrata lo scorso mese. Le prime due, PortWait (da non confondere con la WaitPort di Exec!!!) e MessageWait, permettono di sospendere il processo chiamante per un numero di secondi indicato o fino a quando non viene creata da qualche altro processo una porta, o arriva un messaggio atteso. In tutt'e due i casi il primo parametro è il nome della porta mttb e il secondo parametro è il tempo massimo da aspettare espresso in secondi. Indicando un tempo nullo il timeout è infinito (usare con cautela...).

La funzione CheckPort permette di stabilire se una determinata porta esiste e in caso affermativo quanti messaggi vi sono accodati. L'unico parametro da passare è il nome della porta.

La funzione WipePort serve invece per pulire una porta mttb da messaggi non più desiderati. Il primo parametro è, come al solito, la porta su cui agire, il secondo parametro indica la quantità di messaggi da togliere. Indicando 0 in questo campo la porta è ripulita da tutti i messaggi presenti. In risposta abbiamo il numero di messaggi tolti (o errore se qualcosa non è andato per il giusto verso).

Per finire, la funzione MessageLen permette di conoscere la dimensione del messaggio presente su una porta mttb. Ciò è molto utile quando non si conosce la quantità di memoria da riservare per contenere il messaggio arrivato. Il primo parametro è il modo di funzionamento (MODE_WAIT o MODE_NOWAIT), il secondo parametro è la porta. Indicando MODE_WAIT se la porta è vuota il processo chiamante è sospeso fino a quando non arriva il messaggio; indicando MODE_NOWAIT, nel caso la porta sia vuota, la funzione ritorna la costante predefinita EMPTY_PORT.

Sul prossimo numero cercheremo di mettere le rimanenti funzioni offerte dall'ADPmttb. Arrivederci...

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


int  PortWait(char *,int);
int  MessageWait(char *,int);
int  CheckPort(char *);
int  WipePort(char *, int);
int  MessageLen(UBYTE,char *);

extern struct ExecBase *SysBase;

/*************************************
*                                    *
*       P O R T   W A I T            *
*                                    *
*************************************/

PortWait(porta,sec)
char *porta;

{
if (sec == 0) sec--;
while ((struct MsgPort *)FindPort(porta) == NULL && sec)
      {
      Delay(SysBase->VBlankFrequency);
      sec--;
      }
return(iabs(sec));
}

/*************************************
*                                    *
*   M E S S A G E W A I T            *
*                                    *
*************************************/

MessageWait(porta,sec)
char *porta;

{

struct MsgPort *port;

if (sec == 0) sec--;
Forbid();
   if ((port = (struct MsgPort *)FindPort(porta)) == 0)
      {
      Permit();
      return(NO_PORT);
      }
Permit();

while (port->mp_MsgList.lh_Head->ln_Succ == NULL && sec)
      {
      Delay(SysBase->VBlankFrequency);
      sec--;
      }

return(iabs(sec));
}

/*************************************
*                                    *
*     C H E C K P O R T              *
*                                    *
*************************************/

CheckPort(porta)
char *porta;

{

int count=0;
struct MsgPort *port;
struct Node    *node;

Forbid();
   if ((port = (struct MsgPort *)FindPort(porta)) == 0)
      {
      Permit();
      return(NO_PORT);
      }
   for (node = port->mp_MsgList.lh_Head;
        node->ln_Succ;
        node = node->ln_Succ) count++;
Permit();
if (count) return(count);
else return(EMPTY_PORT);

}

/*************************************
*                                    *
*       W I P E   P O R T            *
*                                    *
*************************************/

WipePort(porta,nmsg)
char *porta;

{
struct adp_message *adpmsg;
struct MsgPort *port;
int i=0;

if (nmsg == 0) nmsg--;
Forbid();
   if ((port = (struct MsgPort *)FindPort(porta)) == 0)
      {
      Permit();
      return(NO_PORT);
      }
   while ((adpmsg = (struct adp_message *)GetMsg(port)) && nmsg--)
         {
         i++;
         if (adpmsg->mode & MODE_SYNC) ReplyMsg(adpmsg);
         else FreeMem( adpmsg, sizeof(struct adp_message) + adpmsg->len );
         }
Permit();
return(i);
}

/*************************************
*                                    *
*    M E S S A G E   L E N           *
*                                    *
*************************************/

MessageLen(mode,porta)
UBYTE mode;
char *porta;

{
struct MsgPort *port;
struct adp_message *temp;
ULONG  succ;
if ((mode & MODE_WAIT || mode & MODE_NOWAIT)==0) return(OP_FAIL);
Forbid();
   if ((port = (struct MsgPort *)FindPort(porta)) == 0)
      {
      Permit();
      return(NO_PORT);
      }
Permit();
if (mode & MODE_WAIT) WaitPort(port);
temp = (struct adp_message *)(port->mp_MsgList.lh_Head);
succ = (ULONG)(((struct Node *)temp)->ln_Succ);
if (succ) return(temp->len);
else return(EMPTY_PORT);
}
```

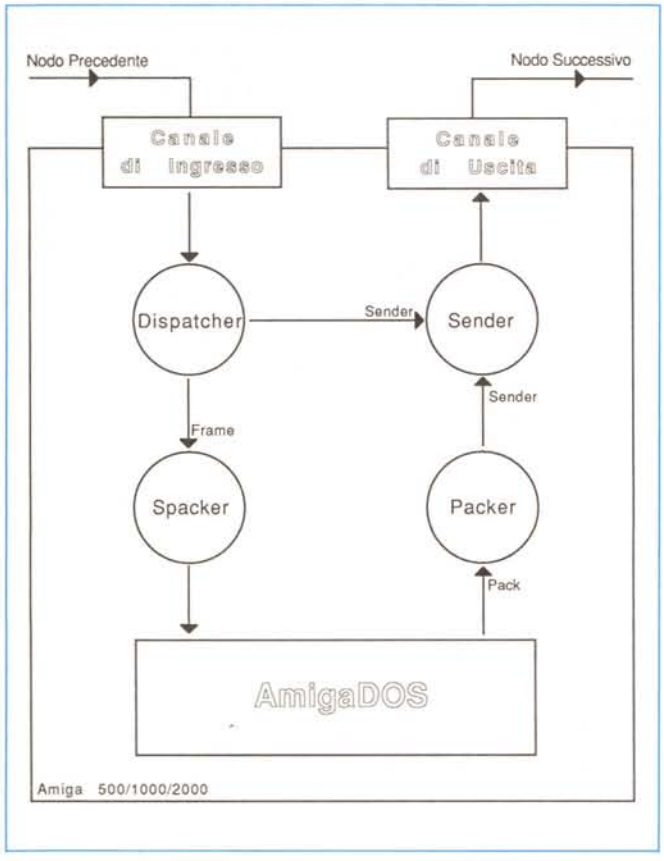


Figura 5
I quattro processi di base di ADPnetwork.

CRC (Cyclical Redundance Code) controlla invece possibili errori di trasmissione sulla vera e propria porzione di messaggio trasmessa. Gli ultimi tre campi servono per ricostruire il messaggio originario man mano che arrivano i frame. I primi due (costanti su ogni frame di uno stesso messaggio) indicano l'effettiva lunghezza del messaggio originario prima della frammentazione e l'identificatore di messaggio che viene incrementato, localmente ad ogni macchina, ogni nuova richiesta di spedizione su rete. L'ultimo campo, Offset, indica a partire da quale byte il corpo di questo frame va posizionato: nessuna assunzione è fatta circa la sequenzialità dei frame di un messaggio. Infatti frame arrivati male verranno rispediti solo dopo la prima spedizione dell'ultimo frame e quindi è sempre necessario sapere ciò che arriva a quale porzione del messaggio originario corrisponde.

Per finire, tutti i rimanenti byte del frame contengono la parte di messaggio spedita.

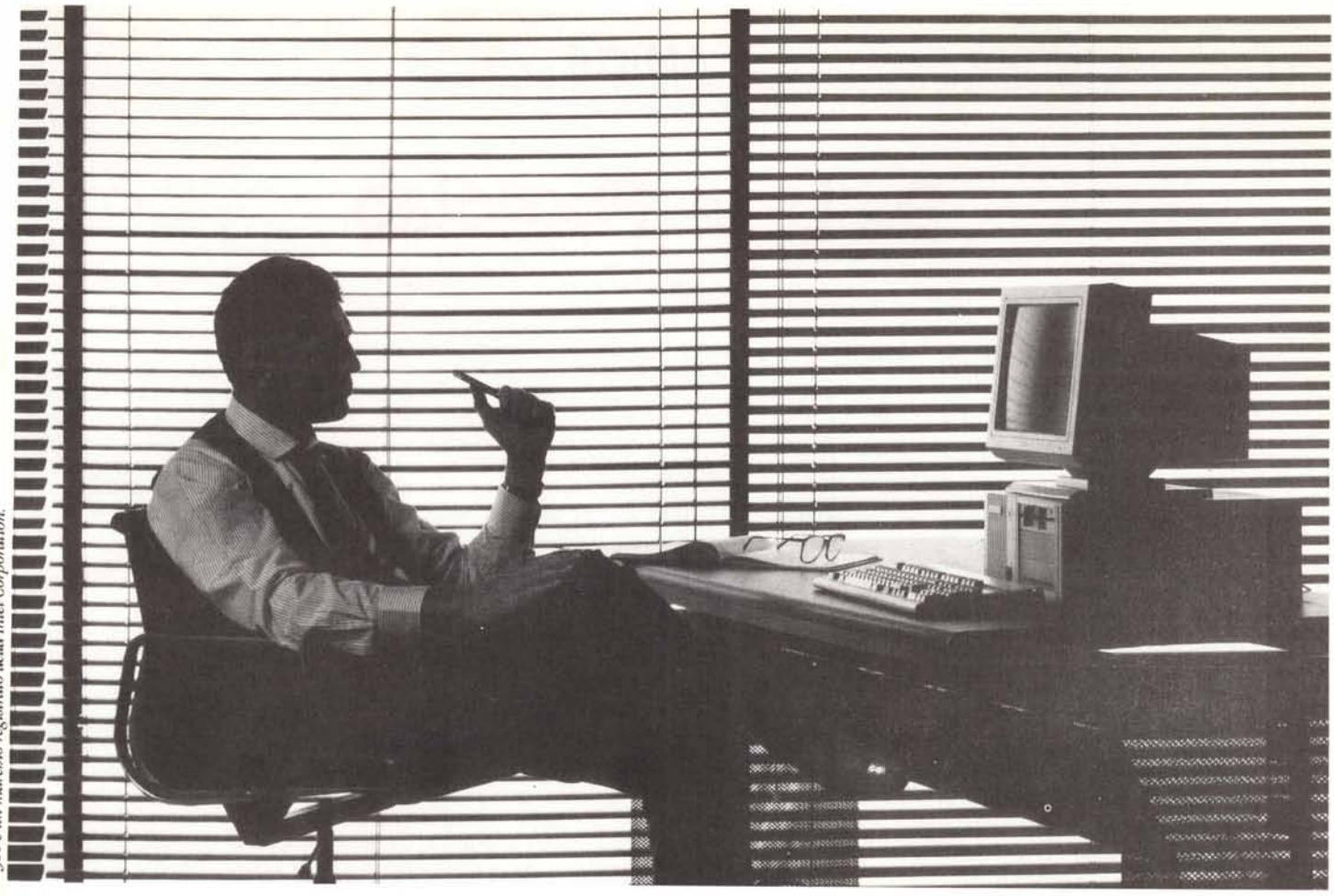
Il Software di Rete

Per Software di Rete (SDR) intendiamo i processi lanciati in background su ogni macchina atti a permettere la comunicazione tra due qualsiasi nodi. Utilizzando l'SDR è possibile a due processi qualunque lanciati su due macchine distinte di scambiare agevolmente messaggi. È su questo SDR che Marco Ciuchini e Andrea Suatoni hanno installato i loro Net-Handler e NetServer per realizzare una sorta di file system distribuito permettendo tanto l'utilizzo sotto WB della rete quanto l'accesso a file remoti da shell e da ogni programma commerciale purché questo sia stato realizzato all'origine secondo canoni di programmazione «puliti». Tanto per non fare nomi, il nostro fiore all'occhiello italiano, C1-Text della Cloanto Italia, caricato su una macchina collegata in rete, nei requester «Aprire documento» o «Memorizzare documento» mostra un nuovo button «NET» tramite il quale possiamo leggere e salvare file su altre macchine.

L'SDR è attualmente formato da diversi processi cooperanti (scritti tutti, ovviamente, in ADPmttb 2.0) e per il momento vi mostreremo il funzionamento dei quattro modulil più importanti: Packer, Sender, Dispatcher e Spacker (beh, in effetti un nome più corretto per questo processo sarebbe UnPacker...). Il loro schema di cooperazione è mostrato in figura 5.

Poniamoci allora dal punto di vista di una macchina che deve spedire un messaggio ad un processo in esecuzione su un'altra macchina e vediamo cosa succede. L'ipotesi di collegamento è sempre quella di figura 3, che poi è quella che ci ha accompagnato in tutti questi mesi di sperimentazione e realizzazione qui in redazione. Allora, diciamo che Socrate vuole spedire un messaggio lungo 23 K a Pitagora. Ciò che deve fare è mandare un messaggio al processo Packer in esecuzione sulla sua stessa macchina, contenente il nome del destinatario («Pitagora») il tipo del messaggio (per il momento ignoreremo questo campo) la lunghezza del messaggio, 23552 pari cioè a 23×1024 byte, e il messaggio vero e proprio cioè i 23552 byte da spedire. Ricevuto il messaggio, il Packer provvede a spezzare i 23552 byte in 5 porzioni, quattro delle quali lunghe circa 5 K e l'ultima circa 3 K. Il circa è dovuto al fatto che ogni frame è lungo un multiplo di 128 byte, compreso però l'header che contiene le informazioni trattate nel precedente paragrafo. Ogni frame costruito dal Packer è passato al processo Sender che provvede alla effettiva spedizione sul canale di uscita. Lo stesso Sender riceve sulla medesima porta anche frame provenienti dal Dispatcher che, ricevendoli dal canale di ingresso, ma non riconoscendoli come destinati a quel nodo, li inoltra alla macchina successiva. Questa sarà l'operazione svolta da Platone che, come mostrato in figura 3, si trova tra Socrate e Pitagora. All'interno di quest'ultima, man mano che giungono i frame al suo Dispatcher, riconosciuti come propri, i frame sono inviati al processo Spacker che provvede a «riincollare» le 5 porzioni di messaggio arrivate in sequenza. Non appena il processo Spacker si accorge di aver ricomposto interamente il messaggio originario (e se ne accorge sapendo da ogni frame la lunghezza totale e la posizione relativa dei vari pezzi arrivati) provvede a spedirlo al processo opportuno, funzione del «tipo messaggio» che non abbiamo ancora trattato. Comunque per il momento è tutto. Mentre qui in redazione continuiamo a fare gli ultimi ritocchi «ottimizzatori» al SDR il sottoscritto e all'Handler di rete l'accoppiata vincente Ciuchini-Suatoni, vi diamo appuntamento al prossimo numero per un più approfondito commento al funzionamento dei processi e ai meccanismi utilizzati per implementare la tolleranza ai guasti.

MC



LEO BURNETT

* 386 è un marchio registrato della Intel Corporation.

Prima di dare il posto a un computer leggete il suo curriculum.

1975. Quando Jugi Tandon arriva in California, a Silicon Valley, è il boom. La sua azienda comincia dalle testine di lettura e scrittura per i disk drive. In soli due anni è leader con l'80% del mercato.

1979. Dalla Tandon escono i primi drive completi per floppy disk. Già dopo un anno è leader nel nuovo settore.

1985. Nasce la linea di PC Tandon. Caratteristiche chiave, la compatibilità e l'ottimo rapporto prezzo/prestazioni.

1986. In soli tre paesi europei, Tandon vende 55.000 unità in un anno. Un record mai raggiunto.

1987. Il Personal Data Pac Tandon è il primo hard disk estraibile. Per il computer significa memoria illimitata, portatilità e sicurezza dati. In 4 paesi europei il PAC 286 Tandon è il "Personal Computer of the Year".

1988. Tandon è ormai fra i primi 5 produttori di PC compatibili standard in Europa, con 137.701 sistemi venduti. Mentre si avvia un centro di produzione europeo in Austria, apre la filiale italiana.

1989. 8 marzo: al CEBIT di Hannover, Tandon annuncia il primo personal 386*/33 Mhz, il più potente al mondo.

5/9 ottobre: è allo SMAU con una gamma completa di PC professionali, dall'XT ai 386, e novità ricche di interesse. Per le aziende che, come noi in questi 15 anni, hanno un solo obiettivo: essere i più competitivi.

Tandon

USA TECHNOLOGY MADE IN EUROPE

TANDON COMPUTER S.p.A. - Via E. Fermi 20 - 20094 Assago (MI) - Tel. 02/4883352