Capitolo e8

Sistemi a più processori

Obiettivo dell'industria informatica è stato fin dall'inizio la ricerca di una potenza di calcolo sempre maggiore. L'ENIAC poteva eseguire 300 operazioni al secondo, oltre 1000 volte più veloce di qualsiasi calcolatore che lo aveva preceduto, eppure la gente non era soddisfatta. Oggi abbiamo macchine milioni di volte più veloci dell'ENIAC, eppure chiediamo ancora ulteriore potenza. Gli astronomi stanno provando a dare un senso all'universo, i biologi a capire le implicazioni del genoma umano e gli ingegneri aeronautici sono interessati alla costruzione di navicelle spaziali sempre più sicure ed efficienti e tutti vogliono più cicli di CPU. Per quanta sia la potenza di calcolo, non è mai sufficiente.

In passato, la soluzione era aumentare la velocità di clock; purtroppo, però, stiamo per raggiungere alcuni limiti fondamentali della velocità del clock. Secondo la speciale legge della relatività di Einstein, nessun segnale elettrico può propagarsi più velocemente della velocità della luce, che è di circa 30 cm/ns sotto vuoto e circa 20 cm/ns nei cavi di rame o nella fibra ottica. Ciò significa che in un computer con un clock da 10 GHz i segnali non possono spostarsi più di 2 cm in totale. Per un computer a 100 GHz la distanza totale percorribile è al massimo di 2 cm. Un computer da 1 THz (1000 GHz) dovrà avere dimensioni inferiori ai 100 micron solo per dare al segnale il tempo di procedere avanti e indietro una sola volta in un singolo ciclo di clock.

Realizzare computer così piccoli è possibile, facendo però sorgere un altro problema cruciale: la dissipazione di calore. Più un computer è veloce, più scalda, e più piccolo è il computer più è difficile eliminare il calore. Già sui sistemi Pentium di fascia alta la ventola di raffreddamento della CPU è più grande della stessa CPU.

Se il passaggio da 1 MHz a 1 GHz ha richiesto solo un miglioramento nell'ingegnerizzazione dei processi produttivi dei chip. Il passaggio da 1 GHz a 1 THz richiederà un approccio radicalmente diverso.

Un modo per ottenere una maggiore velocità è attraverso l'uso estremo di parallelismo in un computer.. Queste macchine sono composte da molte CPU, ciascuna delle quali funziona a velocità "normale" (quale che sia in un dato momento), ma che messe insieme danno una potenza di calcolo molto superiore rispetto a una singola CPU. Sono disponibili in commercio sistemi con più di mille CPU; nei laboratori sono già in fase di sperimentazione sistemi con un milione di CPU (Furber et al., 2013). Esistono anche altri approcci potenzialmente validi per ottenere un aumento della velocità, come i computer biologici, ma in questo capitolo ci concentreremo sui sistemi convenzionali con più CPU.

I computer che fanno uso di più CPU vengono frequentemente utilizzati per elaborazioni numeriche molto pesanti. Problemi come le previsioni del tempo, la modellazione del flusso aerodinamico sull'ala di un aeroplano, la simulazione dell'economia mondiale e la comprensione dei recettori della droga nel cervello sono tutti processi molto impegnativi dal punto di vista del calcolo. Le loro risoluzioni comportano lunghe elaborazioni su molte CPU. I sistemi multiprocessore di cui si parla in questo capitolo sono ampiamente usati, per questi e altri problemi simili in campo scientifico e ingegneristico.

Un altro sviluppo rilevante è la crescita incredibilmente veloce di Internet. Progettata come un prototipo per un sistema di controllo militare tollerante ai malfunzionamenti, è poi diventata popolare nella comunità accademica degli informatici e nel tempo ha acquisito molti nuovi utilizzi. Uno di questi è il collegamento di migliaia di computer in tutto il mondo per lavorare insieme su vaste problematiche scientifiche. In un certo senso, un sistema composto da mille computer distribuiti nel mondo non è diverso da uno di mille computer in una sola stanza, anche se il ritardo e altre caratteristiche tecniche sono differenti. In questo capitolo considereremo anche questi sistemi.

Mettere in una stanza un milione di computer non collegati è un'operazione semplice: basta disporre di abbastanza denaro e di una stanza sufficientemente grande. Distribuire un milione di computer non collegati è ancor più facile, dato che risolve il secondo problema. La questione si pone quando si vuole che comunichino l'uno con l'altro per lavorare insieme su un singolo compito; di conseguenza è stato svolto un grande lavoro sulla tecnologia d'interconnessione: diverse tecnologie d'interconnessione hanno portato a tipi di sistemi qualitativamente diversi e a diverse organizzazioni software.

Tutte le comunicazioni fra componenti elettronici (o ottici) hanno come obiettivo finale la spedizione di messaggi – stringhe ben definite di bit. Le differenze stanno nella scala del tempo, nella scala delle distanze e nell'organizzazione logica coinvolta. Ad un estremo vi sono i multiprocessori a memoria condivisa, in cui un certo numero di CPU comunicano tramite una memoria condivisa. In questo modello, ogni CPU ha uguale accesso all'intera memoria fisica e può leggere e scrivere singole parole usando le istruzioni `LOAD` e `STORE`. L'accesso a una parola di memoria generalmente richiede dai 2 ai 10 ns. Mentre questo modello, illustrato nella Figura 8.1, sembra semplice, la sua reale implementazione non lo è altrettanto e generalmente implica un considerevole passaggio di messaggi "nascosti", come spiegheremo a breve. Tuttavia questo passaggio di messaggi è invisibile ai programmatori.

Il passo successivo è il sistema della Figura 8.1(b), in cui un numero di coppie memoria-CPU viene collegato mediante un'interconnessione ad alta velocità. Questo tipo di sistema è detto multicomputer a scambio di messaggi. Ciascuna memoria è locale a una singola CPU e solo quella CPU può accedervi. Le CPU comunicano inviando messaggi multiparola sull'interconnessione. Con una buona interconnessione un breve messaggio può essere spedito in 10-50 μs, un tempo ancora troppo lontano da quello di accesso alla memoria della Figura 8.1(a). In questa configurazione non c'è una memoria globale condivisa. I multicomputer (cioè i sistemi a scambio di messaggi) sono molto più semplici da costruire dei multiprocessori (a memoria condivisa), ma sono più difficili da programmare. In questo modo si soddisfano i gusti di tutti.

Il terzo modello illustrato nella Figura 8.1(c) connette sistemi di computer completi su di un'area geografica estesa, come Internet, a formare dei **sistemi distribuiti**. Ciascuno di questi ha la propria memoria e comunica mediante lo scambio di messaggi. La sola reale

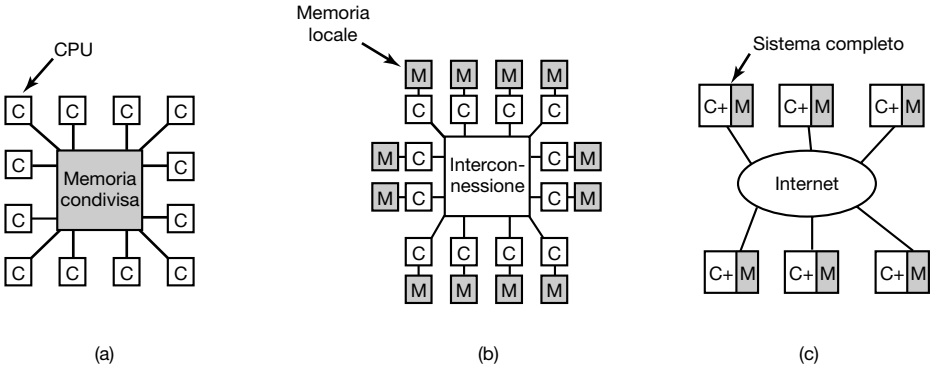

Figura 8.1 (a) Un multiprocessore a memoria condivisa. (b) Un multicomputer a scambio di messaggi. (c) Un sistema distribuito su un'area geografica estesa.

differenza fra la Figura 8.1(b) e 8.1(c) è che nell'ultima vengono utilizzati dei computer completi e i tempi di trasmissione dei messaggi di solito si aggirano tra i 10 e i 100 ms. Questo lungo ritardo obbliga a utilizzare questi **sistemi debolmente strutturati** (**loosely coupled**) in modi diversi dai **sistemi fortemente strutturati** (**tightly coupled**) della Figura 8.1(b). I ritardi di trasmissione dei tre tipi di sistemi differiscono di qualcosa come tre ordini di grandezza, ossia come la differenza fra un giorno e tre anni.

Questo capitolo è organizzato in tre sezioni principali, corrispondenti ai tre modelli della Figura 8.1. Per ciascun modello trattato nel corso del capitolo, si inizia con una breve introduzione all'hardware pertinente, passando quindi al software, specialmente al sistema operativo relativo a quel tipo di sistema. Come si vedrà, in ciascun caso si affrontano problemi diversi e servono pertanto approcci differenti.

8.1 Multiprocessori

Un **sistema multiprocessore a memoria condivisa** (o da qui in poi semplicemente *multiprocessore*) è un sistema di computer in cui due o più CPU condividono il pieno accesso a una RAM comune. Un programma eseguito in una qualsiasi delle CPU vede un normale spazio degli indirizzi virtuali (generalmente paginato). L'unica caratteristica inusuale che ha questo sistema è che la CPU può scrivere dei valori in una parola di memoria e poi rileggere la parola e trovarvi un valore diverso (perché un'altra CPU lo ha cambiato). Quando è organizzata correttamente, questa proprietà si trova alla base della comunicazione fra i processori: una CPU scrive determinati dati nella memoria, l'altra, leggendoli, li preleva.

I sistemi operativi multiprocessore sono per lo più normali sistemi operativi. Gestiscono chiamate di sistema, gestiscono la memoria, forniscono un file system e gestiscono i dispositivi di I/O. Oltre a ciò, vi sono aree in cui hanno caratteristiche uniche, fra le quali la sincronizzazione dei processi, la gestione delle risorse e lo scheduling. Nel seguito analizzeremo brevemente l'hardware dei multiprocessori, passando quindi ai sistemi operativi.

8.1.1 Hardware dei multiprocessori

Sebbene tutti i multiprocessori abbiano come caratteristica che tutte le CPU possono indirizzare tutta la memoria, alcuni multiprocessori hanno un'ulteriore caratteristica: ogni parola di memoria può essere letta alla stessa velocità di ogni altra parola di memoria. Queste macchine sono dette **multiprocessori UMA** (**uniform memory access – accesso alla memoria uniforme**). I **multiprocessori NUMA** (**nonuniform memory access**), invece, sono privi di questa caratteristica; il perché dell'esistenza di tale differenza si chiarirà più avanti. Esamineremo prima i multiprocessori UMA, per poi passare ai multiprocessori NUMA.

Multiprocessori UMA con architetture basate sui bus

I multiprocessori più semplici sono basati su un bus singolo, come illustrato nella Figura 8.2(a). Due o più CPU e uno o più moduli di memoria usano tutti lo stesso bus per le comunicazioni. Quando una CPU vuole leggere una parola della memoria, prima controlla se il bus è impegnato; se è inattivo, la CPU immette sul bus l'indirizzo della parola desiderata, dichiara un segnale di controllo e attende finché la memoria mette sul bus la parola prescelta.

Se il bus è impegnato quando la CPU vuole leggere o scrivere la memoria, la CPU aspetta che sia disponibile. Il problema di questa architettura sta proprio qui: con due o tre CPU le contese per il bus possono essere gestibili; con 32 o 64 sarebbero insostenibili. Il sistema sarebbe completamente limitato dalla banda del bus e la maggior parte delle CPU rimarrebbe inattiva per la maggior parte del tempo.

La soluzione a questo problema sta nell'aggiunta di una cache a ciascuna CPU, come descritto nella Figura 8.2(b). La cache può essere interna al chip della CPU, vicino al chip della CPU, sulla scheda del processore o in una qualche combinazione delle tre. Dato che ora possono essere soddisfatte molte letture direttamente dalla cache locale, ci sarà molto meno traffico sul bus e il sistema potrà supportare più CPU. In generale l'uso della cache non avviene sulla base di una singola parola, ma su blocchi di 32 o 64 byte. Quando si referenzia una parola, è il suo blocco per intero, chiamato **linea di cache**, a essere caricato nella cache della CPU che lo richiede.

Ogni blocco della cache è contrassegnato come in sola lettura (nel cui caso può essere presente in più cache contemporaneamente) o in lettura-scrittura (nel cui caso non può trovarsi in nessun'altra cache). Se una CPU prova a scrivere una parola che si trova in una

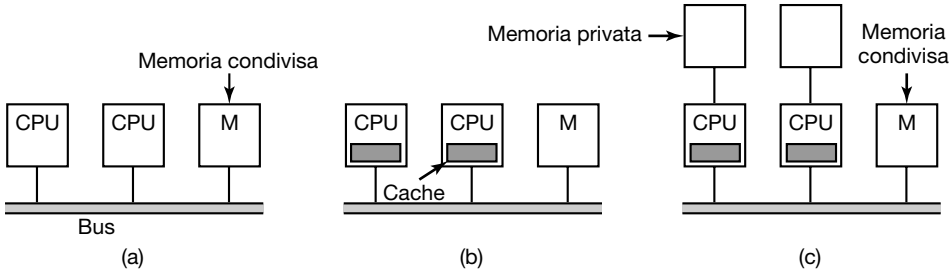

Figura 8.2 Tre multiprocessori basati su bus. (a) Senza uso della cache. (b) Con uso della cache. (c) Con uso della cache e memorie private.

o più cache remote, l'hardware del bus rileva la scrittura e segnala sul bus la scrittura a tutte le altre cache. Se le altre cache hanno una copia "pulita", ossia una copia esatta di quanto è in memoria, allora possono semplicemente eliminare la propria copia e lasciare che chi scrive prelevi il blocco di cache dalla memoria prima di modificarlo. Se qualche altra cache ha una copia "sporca" (cioè modificata), deve riscriverla nella memoria prima che la scrittura possa procedere o trasferirla direttamente a chi scrive sul bus. Questo insieme di regole è chiamato **protocollo di coerenza** della cache ed è uno dei tanti possibili.

Un'altra possibilità è data dall'architettura illustrata nella Figura 8.2(c), in cui ciascuna CPU non ha solo una cache, ma anche una memoria locale privata, alla quale accede attraverso un bus dedicato (e privato). Per utilizzare questa configurazione in modo ottimale, il compilatore deve posizionare tutto il testo dei programmi, le stringhe, le costanti e gli altri dati di sola lettura, gli stack e le variabili locali nelle memorie private. La memoria condivisa viene pertanto impiegata solo per le variabili condivise scrivibili. Nella maggior parte dei casi, questo attento posizionamento riduce enormemente il traffico del bus, ma richiede una collaborazione attiva da parte del compilatore.

Multiprocessori UMA con crossbar switch

Anche con le migliori tecniche di caching, l'uso di un singolo bus limita la dimensione del multiprocessore UMA a 16 o 32 CPU. Per superare questo limite è necessario un diverso tipo di rete d'interconnessione. Il circuito più semplice per la connessione di n CPU a k memorie è il **crossbar switch**, illustrato nella Figura 8.3. I crossbar switch sono stati usati per anni nei commutatori telefonici per collegare un gruppo di linee in ingresso a un insieme di linee in uscita in modo arbitrario.

A ciascuna intersezione di una linea orizzontale (in ingresso) con una verticale (in uscita) c'è un **crosspoint** (**punto di incrocio**). Un crosspoint è un piccolo interruttore che può essere aperto o chiuso elettricamente, a seconda che le linee orizzontali e verticali debbano essere collegate o meno. Nella Figura 8.3(a) vediamo tre crosspoint chiusi contemporaneamente, che consentono le connessioni fra le coppie (CPU, memoria) (010, 000), (101, 101) e (110, 010) nello stesso tempo. Sono possibili molte altre combinazioni; il numero di combinazioni è uguale al numero di diverse possibilità in cui si possono posizionare otto torri su una scacchiera.

Una delle caratteristiche più utili del crossbar switch è che si tratta di una **rete non bloccante** (**nonblocking network**), ossia che a nessuna CPU è mai negata la connessione di cui ha bisogno perché qualche crosspoint o linea sono già occupati (dando per scontato che il modulo di memoria stesso sia disponibile). Inoltre non serve alcuna pianificazione anticipata. Anche quando sono già impostate sette connessioni arbitrarie, è sempre possibile collegare la CPU rimanente alla memoria che resta.

Rimane comunque la possibilità di competizione per la memoria, qualora due CPU vogliano accedere allo stesso modulo nel medesimo istante. Tuttavia, partizionando la memoria in n unità, la competizione è ridotta di un fattore n rispetto al modello della Figura 8.2.

Uno dei difetti del crossbar switch è il fatto che il numero degli incroci cresce in progressione quadratica. Con mille CPU e relativi mille moduli di memoria occorre un milione di crosspoint e un crossbar switch di queste dimensioni non è fattibile. Per i sistemi di medie dimensioni si può invece optare per il crossbar switch.

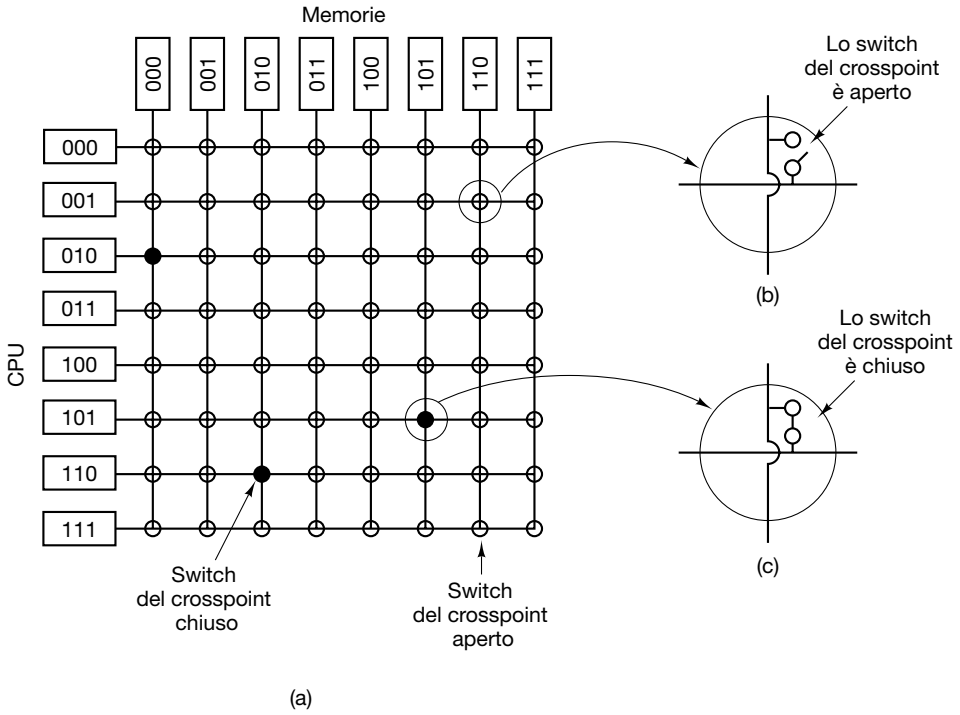

Figura 8.3 (a) Crossbar switch 8 × 8. (b) Un crosspoint aperto. (c) Un crosspoint chiuso.

Multiprocessori UMA con multistage switching network

Un'architettura a multiprocessori completamente diversa è basata sul semplice switch 2 × 2 mostrato nella Figura 8.4(a). Questo switch ha due input e due output. I messaggi che arrivano su entrambe le linee di input possono essere scambiati sulle due linee di output. Per i nostri scopi i messaggi conterranno fino a quattro parti, come mostrato nella Figura 8.4(b). Il campo *Module* indica quale memoria usare. *Address* specifica un indirizzo nel modulo. *Opcode* specifica l'operazione, come `READ` o `WRITE`. Infine, il campo opzionale *Value* può contenere un operando, come una parola a 32 bit da scrivere con una `WRITE`. Lo switch analizza il campo *Module* e lo usa per verificare se il messaggio debba essere inviato su *X* o su *Y*.

Questi switch 2 × 2 possono essere sistemati in molti modi per costruire una più vasta **multistage switching network** (Adams et al., 1987; Bhunyan et al., 1989 e Kumar eReddy, 1987). Una possibilità è la **rete omega**, economica ed essenziale, illustrata nella Figura 8.5. In questo caso ci sono otto CPU collegate a otto memorie mediante 12 switch. Più in generale, per n CPU e n memorie servono $\log_2 n$ stadi, con $n/2$ switch per stadio, per un totale di $(n/2)\log_2 n$ switch, con un ovvio vantaggio rispetto ai n^2 crosspoint, specie per valori di n elevati.

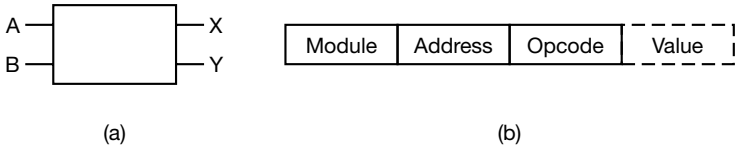

Figura 8.4 (a) Switch 2 × 2 con due linee di input, *A* e *B*, e due linee di output, *X* e *Y*. (b) Il formato di un messaggio.

Lo schema di connessione della rete omega è spesso chiamato **perfect shuffle** poiché il mix dei segnali a ciascuno stadio è simile a un mazzo di carte tagliato in due e poi mescolato carta per carta. Per vedere come funziona la rete omega, si supponga che la CPU 011 voglia leggere una parola dal modulo di memoria 110. La CPU invia un messaggio `READ` allo switch 1D, contenente il valore 110 nel campo *Module*. Lo switch prende il primo bit (cioè quello più a sinistra) di 110 e lo usa per eseguire l'instradamento. Uno 0 va all'output superiore e un 1 a quello inferiore; poiché questo bit vale 1, il messaggio viene instradato a 2D attraverso l'output inferiore.

Tutti gli switch del secondo stadio, incluso 2D, usano il secondo bit per l'instradamento. Anche questo vale 1; pertanto, il messaggio viene ora inoltrato a 3D attraverso l'output inferiore, dove viene analizzato il terzo bit, che risulta uguale a 0. Di conseguenza, il messaggio esce sull'output superiore e arriva alla memoria 110, come desiderato. Il percorso seguito da questo messaggio è contrassegnato dalla lettera *a* nella Figura 8.5.

Man mano che il messaggio si sposta lungo la rete di switch, i bit più a sinistra del numero del modulo non servono più e possono quindi essere utilizzati registrandovi il numero della linea d'ingresso, in modo che la risposta possa trovare la strada a ritroso. Per il percorso *a,* le linee in ingresso sono rispettivamente 0 (l'input superiore a 1D), 1 (l'input

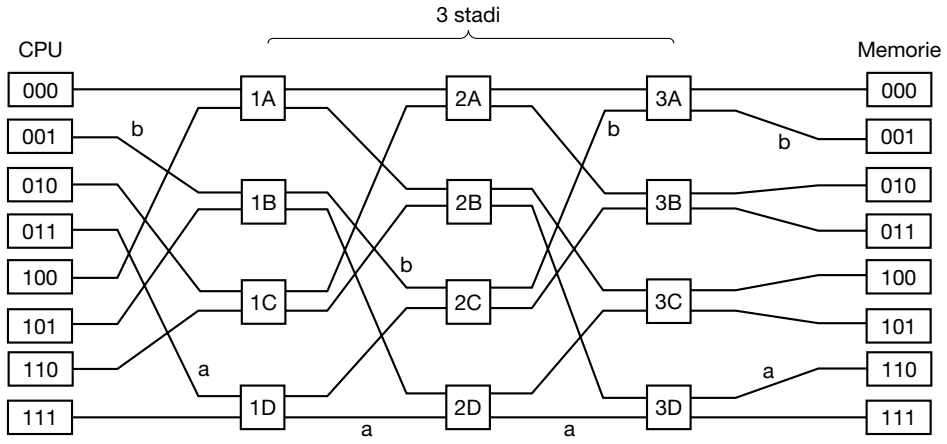

Figura 8.5 Rete omega a switch.

inferiore a 2D) e 1 (l'input inferiore a 3D). La risposta viene restituita utilizzando 011, leggendolo questa volta da destra a sinistra.

Durante queste operazioni, la CPU 001 chiede di scrivere una parola nel modulo di memoria 001. In questo caso si verifica un processo analogo con il messaggio rispettivamente instradato attraverso l'output superiore, quello superiore e quello inferiore, seguendo il percorso contrassegnato dalla lettera *b*. Quando arriva, il campo *Module* legge 001, a rappresentare il percorso intrapreso. Dal momento che queste due richieste non usano gli stessi switch, linee o moduli di memoria, possono procedere in parallelo.

Si consideri ora che cosa accadrebbe se la CPU 000 volesse accedere contemporaneamente al modulo di memoria 000. La sua richiesta andrebbe in conflitto con la richiesta della CPU 001 allo switch 3°: una di loro dovrebbe attendere. A differenza del crossbar switch, la rete omega è una **rete bloccante**, nella quale non tutti gli insiemi di richieste possono essere elaborati contemporaneamente. Possono avvenire conflitti nell'uso di una connessione o di uno switch, così come fra le richieste *verso* la memoria e le risposte *dalla* memoria.

È chiaramente auspicabile distribuire i riferimenti alla memoria in modo uniforme attraverso i moduli. Una tecnica comune consiste nell'utilizzare i bit secondari come numero del modulo. Si consideri per esempio uno spazio degli indirizzi byte-oriented per un computer che accede per lo più a parole intere a 32 bit. I 2 bit meno significativi sarebbero normalmente 00, ma i successivi 3 bit sarebbero distribuiti uniformemente. Usando questi 3 bit come modulo, le parole sarebbero in moduli consecutivi. Un sistema di memorie in cui le parole consecutive sono in moduli diversi è detto **interleaved**. Le memorie *interleaved* massimizzano il parallelismo, perché la maggior parte dei riferimenti alla memoria è in indirizzi consecutivi. È anche possibile progettare reti di switch che siano non bloccanti e offrire più percorsi da ciascuna CPU a ciascun modulo di memoria al fine di distribuire meglio il traffico.

Multiprocessori NUMA

I processori UMA a bus singolo sono generalmente limitati a non più di qualche decina di CPU e i multiprocessi con crossbar o con switch hanno bisogno di una gran quantità di hardware (costoso) e non possono raggiungere dimensioni molto grandi. Per arrivare a più di cento CPU bisogna rinunciare a qualcosa. Di solito ciò a cui si rinuncia è l'idea che tutti i moduli di memoria abbiano lo stesso tempo di accesso. È questa concessione che porta all'idea dei multiprocessori NUMA, come menzionato prima. Come i loro cugini UMA, forniscono uno spazio degli indirizzi singolo trasversale a tutte le CPU, ma, a differenza delle macchine UMA, accedere ai moduli di memoria locale è più veloce che a quelli remoti. Tutti i programmi UMA possono pertanto funzionare senza subire alcuna modifica anche sulle macchine NUMA, ma con prestazioni inferiori a una macchina UMA che abbia la medesima velocità di clock.

Le macchine NUMA hanno tre caratteristiche fondamentali che le distinguono dagli altri processori.

1. C'è un singolo spazio degli indirizzi visibile a tutte le CPU.
2. L'accesso alla memoria remota avviene tramite le istruzioni LOAD e STORE.
3. L'accesso alla memoria remota è più lento che alla memoria locale.

Quando il tempo di accesso alla memoria non è nascosto (perché non c'è *caching*) il sistema è chiamato **NC-NUMA** (**no cache NUMA**). Quando sono presenti delle cache coerenti, il sistema è chiamato **CC-NUMA** (**cache-coherent NUMA**).

L'approccio più comune per la costruzione di grandi multiprocessori CC-NUMA è attualmente il **multiprocessore directory-based**. L'idea è quella di mantenere un database che indichi dove si trova ciascuna linea e qual è il suo stato. Quando è referenziata una linea di cache, si interroga il database per trovare dove si trovi e se sia "pulita" o "sporca" (modificata). Poiché questa base di dati deve essere interrogata su ciascuna istruzione che referenzia la memoria, deve essere tenuta in un hardware specifico estremamente veloce che possa rispondere in una frazione di un ciclo del bus.

Per rendere un po' più concreta l'idea di un processore *directory-based*, consideriamo come (ipoteticamente) semplice l'esempio di un sistema a 256 nodi, ciascuno dei quali composto da una CPU e da 16 MB di RAM connessi alla CPU mediante un bus locale. La memoria totale è di 2^{32} byte, suddivisi in 2^{26} linee di cache di 64 byte l'una. La memoria è allocata staticamente fra i nodi, con 0-16M nel nodo 0, 16-32M nel nodo 1 e così via. I nodi sono collegati da una rete d'interconnessione, come illustrato nella Figura 8.6(a). Ciascun nodo contiene inoltre le voci delle directory per le 2^{18} linee di cache di 64 byte, inclusa la sua memoria di 2^{24} byte. Per ora si presuppone che una linea possa essere contenuta al massimo in una cache.

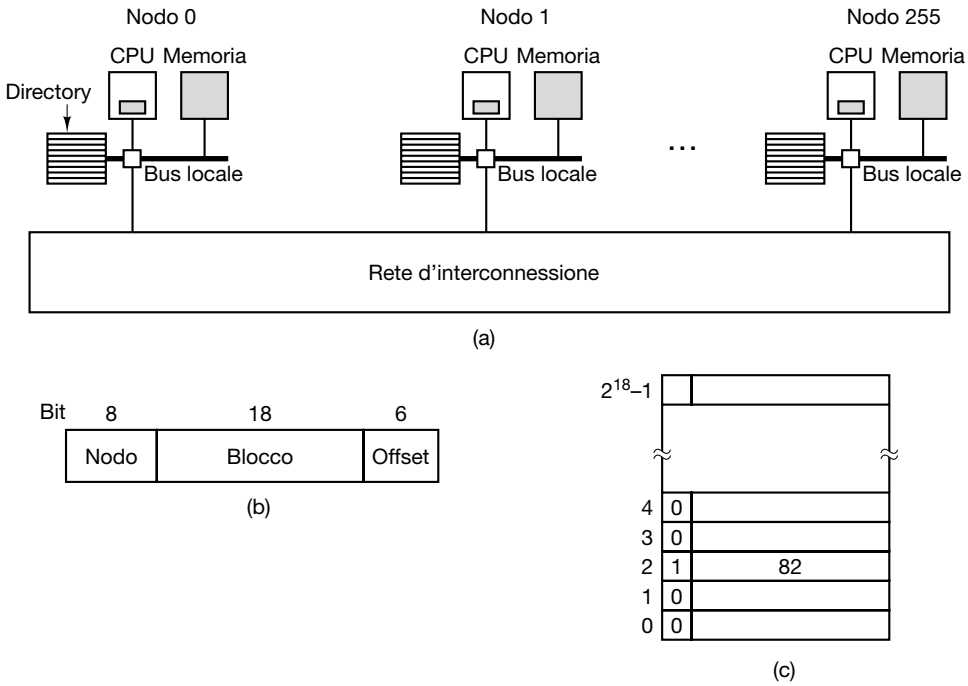

Figura 8.6 (a) Un multiprocessore directory-based a 256 nodi. (b) Divisione in campi di un indirizzo di memoria a 32 bit. (c) La directory al nodo 36.

Per vedere come funziona la directory, proveremo a seguire un'istruzione `LOAD` dalla CPU 20 che referenzia una linea di cache. Per prima cosa la CPU che invia l'istruzione la presenta all'MMU, che la traduce in un indirizzo fisico, per esempio 0x24000108. L'MMU divide questo indirizzo nelle tre parti illustrate nella Figura 8.6(b). In decimale, le tre parti sono il nodo 36, la linea 4 e l'offset 8. L'MMU vede che la parola in memoria referenziata proviene dal nodo 36, non dal 20, quindi invia un messaggio di richiesta attraverso la rete d'interconnessione al nodo iniziale della linea, 36, richiedendo se la sua linea 4 si trova nella cache e, nel caso, dove sia.

Quando attraverso la rete d'interconnessione la richiesta arriva al nodo 36, viene instradata all'hardware della directory. L'hardware indicizza la sua tabella di 2^{18} voci, una per ciascuna linea di cache, ed estrae la voce 4. Dalla Figura 8.6(c) vediamo che la linea non è nella cache, pertanto l'hardware preleva la linea 4 dalla RAM locale, la rimanda al nodo 20 e aggiorna la voce di directory 4 a indicare che la linea è ora in cache al nodo 20.

Si consideri ora una seconda richiesta, questa volta per la linea 2 del 36° nodo. Dalla Figura 8.6(c) si può vedere che questa linea è nella cache al nodo 82. A questo punto l'hardware potrebbe aggiornare la voce di directory 2 per indicare che la linea è ora al nodo 20 e poi inviare un messaggio al nodo 82 comunicandogli di passare la linea al nodo 20 e invalidare la cache. Notate che anche per un cosiddetto "multiprocessore a memoria condivisa" il passaggio di messaggi è intenso.

Facendo una rapida divagazione, si provi a calcolare rapidamente quanta memoria è occupata dalle directory. Ogni nodo ha 16 MB di RAM e 2^{18} voci da 9 bit per tener traccia di quella RAM. L'overhead della directory è pertanto pari a circa 9×2^{18} bit diviso 16 MB, ossia circa l'1,76%, un valore generalmente accettabile (anche se deve essere memoria ad alta velocità, il che ne aumenta naturalmente il costo). Anche con linee di cache di 32 byte l'overhead sarebbe solo del 4%; con linee di cache di 128 byte sarebbe sotto l'1%.

L'ovvia limitazione di questa architettura è che una linea può essere messa nella cache in un solo nodo. Per consentire l'inserimento delle linee nella cache in più nodi, occorre avere un sistema per poterle localizzare tutte, per esempio per invalidarle o aggiornarle dopo una scrittura. In molti multiprocessori, una voce di directory consiste pertanto di un vettore bit costituito da un singolo bit per CPU; un 1 indica che sulla CPU è presente la linea di cache, mentre uno 0 indica che non lo è. Inoltre, ciascuna voce di directory contiene solitamente qualche altro bit, il che significa che il consumo di memoria della directory aumenta sensibilmente.

Chip multicore

A mano a mano che la tecnologia di produzione dei chip migliora, i transistor diventano sempre più piccoli ed è possibile metterne sempre di più su un chip. Questa osservazione empirica, spesso chiamata **legge di Moore**, è stata formulata dal cofondatore di Intel, Gordon Moore. Nel 1974, il chip Intel 8080 conteneva poco più di duemila transistor, mentre oggi le CPU Xeon Nehalem-EX contengono più di due miliardi di transistor.

La domanda sorge spontanea: "Che fare di tutti questi transistor?" Come si è visto nel Paragrafo 1.3.1, una possibilità è aggiungere megabyte di cache al chip. Si tratta di una possibilità concreta e ormai sono molto comuni i chip con cache da 4 a 32 MB. A un certo punto, però, l'aumento della cache può solo incrementare la velocità dal 99% al 99,5%, senza migliorare un granché le prestazioni delle applicazioni. L'altra possibilità è mettere due

o più CPU intere, generalmente chiamate **core**, sullo stesso chip (tecnicamente sullo stesso **die**). I chip dual-core, quad-core e octa-core sono già diffusi; ed è già oggi possibile acquistare chip con centinaia di *core*. Per esempio, Intel Xeon 2651 ha dodici *core* fisici in hyperthreading, ossia 24 *core* virtuali. Ognuno dei dodici *core* fisici ha 32 KB di cache di primo livello per le istruzioni e 32 KB di cache di primo livello per i dati, nonché 256 KB di cache di secondo livello; infine, i dodici *core* condividono 30 MB di cache di terzo livello.

Mentre le CPU possono condividere le cache oppure no (si veda per esempio la Figura 1.8), condividono sempre la memoria principale, che risulta consistente, nel senso che vi è sempre un valore univoco per ciascuna parola di memoria. Circuiti hardware specializzati fanno in modo che, se una parola è presente in due o più cache e una delle CPU modifica la parola, detta parola venga automaticamente e atomicamente rimossa da tutte le cache al fine di mantenere la consistenza. Questo processo è conosciuto come **snooping**.

Il risultato di questa architettura è che i chip multicore sono semplicemente dei piccoli multiprocessori. In effetti i chip multicore sono spesso chiamati **CMP** (**chip-level multiprocessors**). Dal punto di vista del software, i CMP non sono molto diversi dai multiprocessori basati sul bus o dai multiprocessori che usano le switching network. Vi sono tuttavia alcune differenze. Per cominciare, su un multiprocessore basato su un bus, ciascuna CPU ha la propria cache, come nella Figura 8.2(b) e anche come nella struttura AMD della Figura 1.8(b). L'idea di cache condivisa della Figura 1.8(a), usata da Intel, non si presenta in altri multiprocessori. Una cache condivisa di secondo o terzo livello (L2 o L3) può influenzare le prestazioni. Se un *core* ha bisogno di molta memoria cache e gli altri no, questo schema permette a quello che monopolizza la cache di utilizzare tutta quella di cui ha bisogno. D'altra parte, una cache condivisa può anche far sì che un *core* ingordo danneggi le prestazioni degli altri.

Un'altra area in cui i CMP sono diversi dai loro cugini più grandi è la fault tolerance. Dal momento che le CPU sono strettamente connesse, un errore o un difetto nelle componenti condivise potrebbe bloccare più di una CPU nello stesso momento, un evento meno probabile nei multiprocessori tradizionali.

Oltre ai chip multicore simmetrici, dove tutti i *core* sono identici, un'altra categoria di chip multicore è quella dei **system on a chip** (**SoC**). Questi chip hanno una o più CPU principali e alcuni *core* specializzati, come decoder audio e video, criptoprocessori, interfacce di rete e altro, fino ad arrivare a un sistema di computer completo su di un chip.

Chip manycore

Multicore significa semplicemente "più di un *core*". Se però il numero di *core* aumenta, superando di gran lunga gli otto-dieci, si preferisce usare un altro termine. I **chip manycore** sono processori multicore che contengono decine, centinaia o addirittura migliaia di *core*. Per quanto non esista una soglia esatta oltre la quale un processore multicore può essere considerato manycore, una facile distinzione è che, se non è molto importante perdere un paio di *core*, probabilmente si ha a che fare con un sistema manycore.

Le schede acceleratrici come Intel Xeon Phi hanno oltre sessanta *core* x86; altri produttori hanno ormai superato la barriera dei 100 *core*, in cui i *core* sono di tipo diverso; è probabile che a breve verranno prodotti processori generici con un migliaio di *core*. Non è facile immaginare che cosa si possa fare con mille *core*, o in che modo li si possa programmare.

Un altro problema con un numero così elevato di *core* è che la circuiteria necessaria a mantenere la coerenza nelle cache diventa estremamente complessa e costosa. Molti ingegneri temono che la coerenza della cache non riesca a stare al passo con la presenza di diverse migliaia di *core* e alcuni ritengono che bisognerebbe lasciar perdere, temendo che il costo dei protocolli di coerenza hardware possa essere talmente alto che la presenza di così tanti *core* aggiuntivi finisca per non migliorare di molto le prestazioni, perché il processore è troppo impegnato a mantenere la coerenza delle cache. Peggio ancora, verrebbe consumata troppa memoria (veloce) per la directory. Questo fenomeno prende il nome di **muro della coerenza**.

Si consideri, per esempio, la soluzione di coerenza della cache directory-based di cui si è parlato in precedenza. Se ogni directory contiene un vettore bit per indicare quali *core* contengono una determinata linea di cache, la voce di directory per una CPU con 1024 *core* sarebbe lunga come minimo 128 byte. Poiché le linee di cache solo di rado superano i 128 byte, si arriverebbe alla imbarazzante situazione in cui la voce di directory è più grande della linea di cache referenziata; e non è certo questo che si cerca.

Alcuni ingegneri sostengono che l'unico modello di programmazione che abbia dimostrato di essere in grado di scalare a numeri molto elevati di processori è quello che utilizza lo scambio di messaggi e la memoria distribuita, ed è proprio quello che ci si dovrebbe aspettare nei futuri chip manycore. Vi sono processori sperimentali, come Intel SCC a 48 core, nei quali è già stata abbandonata la coerenza della cache, sostituita dal supporto hardware per il passaggio veloce dei messaggi. Esistono anche, peraltro, altri processori che continuano a garantire la coerenza anche con un numero di *core* molto elevato. È possibile anche pensare a modelli ibridi; per esempio, un chip a 1024 *core* potrebbe essere suddiviso in 64 isole, ciascuna delle quali dotata di 16 *core* con coerenza della cache, che però viene abbandonata fra le isole.

Del resto, la presenza di migliaia di *core* non è più nemmeno così speciale. I manycore oggi più diffusi, le unità di elaborazione grafica, si trovano praticamente in qualsiasi sistema informatico non embedded e provvisto di un monitor. Una **GPU** (**graphics processing unit**) è un processore dotato di una memoria dedicata e di migliaia di microscopici *core*. Rispetto ai processori generici, le GPU vengono realizzate investendo maggiormente in circuiti che eseguono calcoli e meno in cache e logica di controllo. Sono quindi l'ideale per eseguire molte piccole elaborazioni parallele, come il rendering di poligoni nelle applicazioni grafiche, mentre non sono altrettanto adatte all'esecuzione di compiti seriali; inoltre, sono difficili da programmare. Anche se si possono usare le GPU per i sistemi operativi (per esempio, per la codifica o l'elaborazione del traffico di rete), è assai improbabile che su una GPU giri una parte consistente del sistema operativo.

Vi sono peraltro altre elaborazioni che vengono effettivamente gestite dalla GPU, soprattutto quelle molto esigenti dal punto di vista computazionale, come quelle scientifiche. Il termine utilizzato per l'elaborazione generica su una GPU è **GPGPU** (**general purpose GPU**). Purtroppo, la programmazione efficiente delle GPU è molto complessa e richiede linguaggi specializzati come **OpenGL** o **CUDA**, linguaggio proprietario di NVIDIA. Una differenza importante fra la programmazione di una GPU e la programmazione di un processore generico è che le GPU sono fondamentalmente macchine "a istruzione singola e dati multipli", ossia che un gran numero di *core* esegue esattamente la stessa istruzione, ma su dati diversi; questo modello di programmazione è l'ideale per il parallelismo dei dati, ma non è sempre comodo da utilizzare in altri stili di programmazione, come il parallelismo dei task.

Multicore eterogenei

In alcuni chip sono integrati, sullo stesso die, una GPU e diversi *core* generici; allo stesso modo, in molti SoC sono contenuti dei *core* generici, accanto ai quali trovano posto uno o più processori specializzati. I sistemi che integrano sullo stesso chip processori di tipi diversi sono detti **multicore eterogenei**. Un esempio di processore multicore eterogeneo è la linea di processori di rete IXP, presentati per la prima volta da Intel nel 2000 e regolarmente aggiornati con le ultime tecnologie. I processori di rete contengono solitamente un solo *core* generico (per esempio, un processore ARM su cui è in esecuzione Linux) e molte decine di processori stream specializzati, idonei all'elaborazione di pacchetti di rete e a non molto altro. In generale sono utilizzati nelle apparecchiature di rete, come router e firewall. Per instradare pacchetti di rete non c'è la necessità di svolgere operazioni in virgola mobile; pertanto, nella maggior parte dei modelli i processori stream sono del tutto privi di unità in virgola mobile. D'altro canto, le operazioni di rete ad alta velocità dipendono pesantemente dalla velocità di accesso alla memoria (per leggere i pacchetti di dati); pertanto, i processori stream sono dotati di un hardware specializzato, proprio a questo scopo.

Negli esempi sopra riportati, i sistemi sono chiaramente eterogenei: i processori stream e i processori di controllo degli IXP sono unità totalmente diverse, con set di istruzioni differenti; lo stesso dicasi per la GPU e i *core* generici. È però possibile creare eterogeneità anche mantenendo il medesimo set di istruzioni. Per esempio, una GPU potrebbe avere pochi *core* "grossi", con pipeline profonde e magari velocità di clock elevate, e una quantità maggiore di *core* "piccoli", più semplici, meno potenti e che magari funzionano a frequenze inferiori. I *core* potenti servono a eseguire il codice che richiede un'elaborazione sequenziale veloce, mentre quelli meno potenti servono per i task che vengono eseguiti in modo più efficiente in parallelo. Un esempio di architettura eterogenea che segue questo schema è la famiglia di processori ARM big.LITTLE.

Programmazione con processori multicore

Come accaduto spesso in passato, l'hardware è molto più avanti del software. Mentre da un lato i chip multicore sono presenti e attuali, non lo è la capacità di scrivere applicazioni che li sfruttino. I linguaggi di programmazione attuali sono poco adatti alla scrittura di programmi altamente parallelizzati e le risorse in termini di buoni compilatori e di strumenti di debugging sono scarse. Pochi programmatori hanno esperienza di programmazione parallela e la maggior parte sa poco della suddivisione del lavoro in più task che possono funzionare in parallelo. La sincronizzazione, l'eliminazione delle race condition e l'elusione dei deadlock sono un problema serio, e sfortunatamente se non sono gestiti in modo appropriato possono avere un impatto disastroso sulle prestazioni. I semafori non sono la risposta giusta.

E al di là di questi problemi iniziali, non è per niente ovvio quali siano le applicazioni che necessitino davvero di centinaia, se non di migliaia, di *core*, soprattutto nell'uso privato. Nelle grosse server farm, d'altro canto, spesso c'è parecchio lavoro da svolgere per un numero elevato di *core*. Per esempio, un server molto popolare potrebbe migliorare la propria efficienza utilizzando un *core* diverso per ciascuna richiesta da parte di un client; i provider di cloud di cui si è parlato nel capitolo precedente potrebbero riunire insieme i *core* per fornire un gran numero di macchine virtuali da noleggiare ai clienti in cerca di potenza di elaborazione su richiesta.

8.1.2 Tipi di sistemi operativi multiprocessore

Passiamo ora dall'hardware dei multiprocessori al software dei multiprocessori, in particolare ai sistemi operativi multiprocessore. Sono possibili vari approcci e di seguito ne sono illustrati tre. Tutti e tre sono utilizzabili sia nei sistemi multicore sia nelle CPU discrete.

Ogni CPU con il proprio sistema operativo

Il metodo più semplice per organizzare un sistema operativo multiprocessore è la divisione statica della memoria in un numero di partizioni uguale al numero di CPU e l'assegnazione a ciascuna della propria memoria privata e della propria copia privata del sistema operativo. In effetti poi le *n* CPU funzionano come *n* computer indipendenti. Un'ovvia ottimizzazione è consentire a tutte le CPU di condividere il codice del sistema operativo e fare copie private delle sole strutture dati del sistema operativo, come illustrato nella Figura 8.7.

Questo modello è comunque migliore rispetto ad avere *n* computer separati, dal momento che consente alle macchine di condividere un insieme di dischi e altri dispositivi di I/O, consentendo al contempo anche la condivisione flessibile della memoria. Per esempio, anche con l'allocazione statica della memoria è possibile assegnare a una CPU una parte aggiuntiva di memoria, così da gestire programmi più grandi in maniera efficiente. Inoltre, i processi possono comunicare l'uno con l'altro con efficienza, permettendo a un produttore di scrivere i dati direttamente nella memoria e al consumatore di prelevarli da dove il produttore li ha scritti. Tuttavia, dal punto di vista dei sistemi operativi, il fatto che ogni CPU esegua il proprio sistema operativo personale è una soluzione abbastanza primitiva.

Vale la pena notare quattro aspetti di questo schema che potrebbero non essere ovvi. Primo, quando un processo fa una chiamata di sistema, questa viene presa e gestita dalla sua CPU utilizzando le strutture dati nelle tabelle di quel sistema operativo.

Secondo, dato che ciascun sistema operativo ha le proprie tabelle, ha anche il proprio insieme di processi da lui stesso schedulati: non c'è condivisione dei processi. Se un utente si connette alla CPU 1, tutti i suoi processi sono eseguiti sulla CPU 1. Di conseguenza può accadere che la CPU 1 sia inattiva, mentre la CPU 2 sia carica di lavoro.

Terzo, non c'è condivisione delle pagine. Può accadere che la CPU 1 abbia delle pagine libere, mentre la CPU 2 sia paginata in continuazione. Non c'è modo per la CPU 2 di prendere a prestito delle pagine dalla CPU 1, dato che l'allocazione della memoria è fissa.

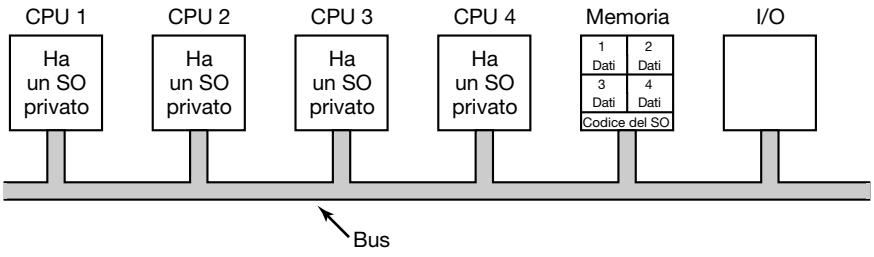

Figura 8.7 Partizionamento della memoria del multiprocessore fra quattro CPU, con la condivisione di una sola copia del codice del sistema operativo. I riquadri contrassegnati come Dati sono i dati privati del sistema operativo di ciascuna CPU.

Quarto, e peggiore, se il sistema operativo mantiene una cache buffer dei blocchi del disco utilizzati di recente, ciascun sistema operativo lo fa indipendentemente dagli altri. In questo modo può accadere che un determinato blocco del disco sia presente come "sporco" in più cache buffer, con una conseguente incoerenza dei dati. L'unico modo per evitare questo problema è l'eliminazione delle cache buffer. Farlo non è difficile, ma le prestazioni ne risentono pesantemente.

Sebbene impiegato nel primo periodo dei multiprocessori, quando l'obiettivo era migrare i sistemi operativi esistenti a un qualche sistema multiprocessore il più in fretta possibile, un simile sistema è usato ormai raramente.

Multiprocessori master-slave

Un secondo modello è illustrato nella Figura 8.8. In questo caso, una copia del sistema operativo e delle sue tabelle è presente sulla CPU 1 e non sulle altre; tutte le chiamate di sistema sono ridirette alla CPU 1, dove vengono elaborate. La CPU 1 può anche eseguire dei processi utente, se le avanza tempo. Questo modello è detto master-slave, poiché la CPU 1 è il *master* (ossia quella che comanda) e le altre sono le slave (sottomesse al master).

Il modello master-slave risolve la maggior parte dei problemi del primo modello. C'è una struttura dati singola (per esempio una lista o un insieme di liste a priorità) che tiene traccia dei processi pronti. Quando una CPU diventa inattiva, richiede al sistema operativo sulla CPU 1 un processo da eseguire e gliene viene assegnato uno. Non può così accadere che una CPU sia inattiva, mentre un'altra è sovraccarica di lavoro. Similmente anche le pagine possono essere allocate fra tutti i processi dinamicamente e c'è una sola buffer cache, così da evitare le inconsistenze.

Il problema di questo modello è che, nel caso di molte CPU, il master diventa un collo di bottiglia: alla fine deve gestire tutte le chiamate di sistema da tutte le CPU. Ipotizzando il 10% del tempo impiegato per le chiamate di sistema, bastano 10 CPU per saturare praticamente il master, 20 CPU per sovraccaricarlo completamente. Perciò questo modello è semplice e funzionale per piccoli multiprocessori, ma inadatto per quelli grandi.

Multiprocessori simmetrici

Il terzo modello, l'**SMP** (**symmetric multiprocessor**), elimina questa asimmetria. C'è una sola copia del sistema operativo in memoria, che può essere eseguita da qualsiasi CPU.

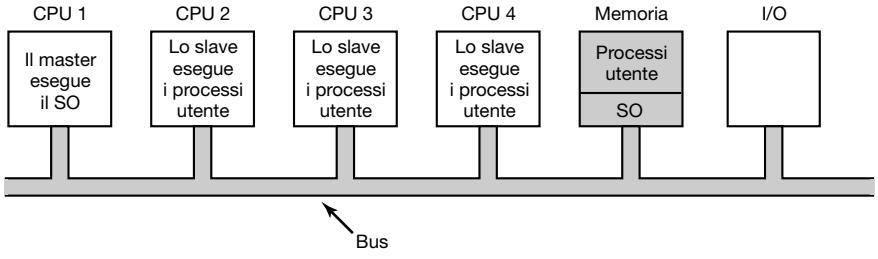

Figura 8.8 Modello di multiprocessore master-slave.

Quando si fa una chiamata di sistema, la CPU su cui è stata eseguita la chiamata di sistema esegue un trap nel kernel ed elabora la chiamata di sistema. Il modello SMP è illustrato nella Figura 8.9.

Questo schema bilancia dinamicamente i processi e la memoria, dal momento che vi è un solo gruppo di tabelle del sistema operativo; inoltre, elimina anche il collo di bottiglia del master, poiché non c'è master; tuttavia, presenta i suoi problemi. In particolare, se una o più CPU stanno eseguendo il codice del sistema operativo contemporaneamente, il risultato può essere disastroso. Immaginate due CPU che prendano simultaneamente lo stesso processo da eseguire o reclamino la stessa pagina di memoria libera. Il modo più semplice per aggirare questo problema è associare un mutex (cioè un lock) al sistema operativo, rendendo l'intero sistema una grande zona critica. Quando una CPU vuole eseguire del codice del sistema operativo, deve prima acquisire il mutex. Se il mutex è in stato *locked* (bloccato), deve semplicemente mettersi in attesa. In questo modo qualunque CPU può eseguire il sistema operativo, ma solo una per volta.

Questo modello funziona, ma è negativo quasi quanto il modello master-slave. Supponete ancora che il 10% del tempo di esecuzione sia impiegato all'interno del sistema operativo: con 20 CPU, ci saranno lunghe code di CPU in attesa del proprio turno. Per fortuna non è difficile da migliorare. Molte parti del sistema operativo sono indipendenti l'una dall'altra: per esempio, non c'è alcun problema se una CPU esegue lo scheduler mentre un'altra gestisce una chiamata al file system e una terza elabora un errore di pagina.

Questa osservazione porta alla suddivisione del sistema operativo in più zone critiche indipendenti che non interagiscono l'una con l'altra. Ciascuna zona critica è protetta dal proprio mutex, così che solo una CPU alla volta possa eseguirla; in questo modo si ottiene qualcosa di molto simile al parallelismo. Può tuttavia accadere che alcune tabelle, come le tabelle dei processi, siano usate da più di una zona critica. Per esempio, la tabella dei processi serve per lo scheduling, ma anche per la chiamata di sistema `fork` nonché per la gestione dei segnali. Ciascuna tabella che può essere usata da molteplici zone critiche necessita del suo mutex. In questo modo, ogni zona critica può essere eseguita da una sola CPU per volta e ciascuna tabella critica può dare accesso a una CPU alla volta.

La maggior parte dei multiprocessori moderni usa questa modalità. La difficoltà nello scrivere un sistema operativo per una macchina di questo tipo non è dovuta al fatto che il codice sia così diverso da quello di un sistema operativo normale: non lo è. La parte difficile

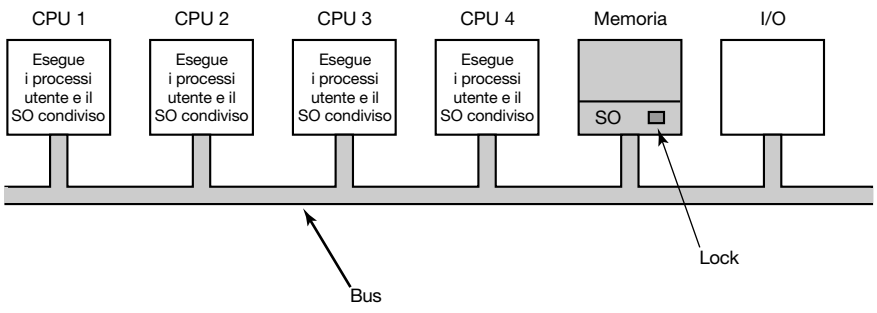

Figura 8.9 Modello di multiprocessore SMP.

sta nella suddivisione in zone critiche che possano essere eseguite dalle diverse CPU in maniera concorrente senza che interferiscano fra loro, nemmeno in modo lieve o indiretto. Inoltre, ogni tabella usata da due o più zone critiche deve essere protetta separatamente da un mutex e tutto il codice che usa la tabella deve usare il mutex nel modo appropriato.

Inoltre, occorre prestare grande attenzione nell'evitare i deadlock. Se due zone critiche necessitano entrambe della tabella A e della tabella B e una di loro richiama A per prima e l'altra richiama B, prima o poi si verificherà un deadlock senza che nessuno sappia il perché. In teoria, a tutte le tabelle possono essere assegnati valori interi e può essere richiesto che le zone critiche acquisiscano le tabelle in ordine crescente. Questa strategia evita i deadlock, ma richiede che il programmatore pensi attentamente alle tabelle di cui necessita ciascuna zona critica e faccia le richieste nell'ordine giusto.

Man mano che il codice avanza, una zona critica può aver bisogno di una nuova tabella di cui non aveva bisogno prima. Se il programmatore è alle prime armi e non ha ben presente l'intera logica del sistema, la tentazione sarebbe quella di prendere il mutex nel punto in cui si rende necessario per poi rilasciarlo nel momento in cui non lo è più. Per quanto possa apparire ragionevole, questo potrebbe causare dei deadlock, percepiti dall'utente come blocchi (*freeze*) del sistema. Prendere la direzione giusta non è semplice e mantenerla nel corso degli anni, nonostante i programmatori cambino, è molto difficile.

8.1.3 Sincronizzazione dei multiprocessori

In un multiprocessore le CPU hanno spesso bisogno di sincronizzarsi. Si è appena visto il caso in cui le zone critiche del kernel e le tabelle devono essere protette dai mutex; ora si darà uno sguardo a come funziona realmente la sincronizzazione in un multiprocessore. Come si vedrà fra poco, non è assolutamente banale.

Innanzitutto, sono assolutamente necessarie delle primitive di sincronizzazione adatte. Se un processo su una macchina monoprocessore (una singola CPU) esegue una chiamata di sistema che richiede l'accesso a una tabella critica del kernel, il codice del kernel può semplicemente disabilitare gli interrupt prima di toccare la tabella. In seguito può svolgere il suo lavoro, sapendo che sarà in grado di portarlo a termine senza che nessun altro processo si insinui e tocchi la tabella prima che abbia terminato. Su un multiprocessore, la disabilitazione degli interrupt interessa solo la CPU che esegue la disabilitazione; le altre CPU continuano a funzionare e possono sempre toccare la tabella critica. Di conseguenza, un appropriato protocollo per l'uso di mutex deve essere usato e rispettato da tutte le CPU per garantire che la mutua esclusione funzioni correttamente. Il cuore di ogni protocollo di mutex pratico è un'istruzione speciale che permette di controllare il valore di una parola di memoria e di modificarlo in un'operazione atomica. Nella Figura 2.22 si è visto come l'istruzione TSL (Test and Set Lock) sia stata usata per implementare le zone critiche. Come già trattato in precedenza, ciò che questa istruzione fa è leggere una parola di memoria e caricarla in un registro. Simultaneamente scrive un 1 (o un valore comunque diverso da zero) nella parola della memoria. Per eseguire la lettura e la scrittura della memoria impiega naturalmente due cicli del bus. Su un monoprocessore, dato che l'istruzione non può essere spezzata in due parti, la TSL funziona sempre come ci si aspetta.

Si pensi ora a che cosa accadrebbe in un multiprocessore. Nella Figura 8.10 vediamo la tempistica del caso peggiore, in cui la parola di memoria 1000, usata come lock, è inizializzata a 0. Nel passaggio 1, la CPU 1 legge la parola e prende lo 0. Nel passaggio 2, prima

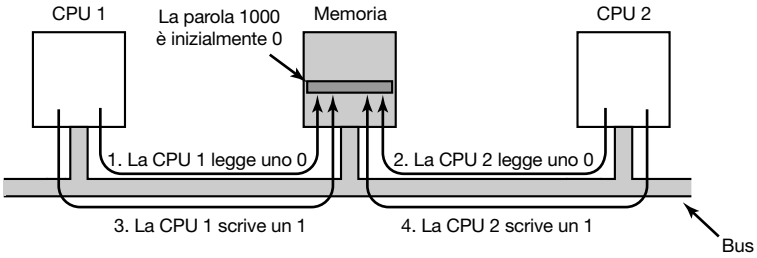

Figura 8.10 L'istruzione TSL può fallire se il bus non può essere messo in *lock*. Questi quattro passaggi mostrano una sequenza di eventi che dimostra il fallimento.

che la CPU 1 abbia la possibilità di riscrivere la parola a 1, la CPU 2 entra e legge la parola come 0. Nel passaggio 3 la CPU 1 scrive un 1 nella parola. Nel passaggio 4, anche la CPU 2 scrive un 1 nella parola. Entrambe le CPU hanno ora accesso alla zona critica e la mutua esclusione fallisce.

Per evitare questo problema, l'istruzione TSL deve prima eseguire il lock del bus, evitando che qualsiasi altra CPU vi acceda, quindi eseguire entrambi gli accessi alla memoria, infine sbloccare il bus. Solitamente, il blocco del bus avviene richiedendo il bus mediante il solito protocollo di richiesta, quindi dichiarando (ossia impostando su un 1 logico) una linea speciale di bus finché *entrambi* i cicli non siano stati completati. Appena dichiarata questa linea speciale, a nessun'altra CPU sarà garantito l'accesso al bus. Questa istruzione può essere implementata solo su di un bus che abbia le necessarie linee e il protocollo (hardware) per usarle.

I bus moderni hanno queste funzionalità, ma sui primi, che non le avevano, non si poteva implementare correttamente la TSL. Questo è il motivo per cui fu inventato il protocollo di Peterson: per eseguire la sincronizzazione completamente via software (Peterson, 1981).

Se la TSL è implementata e usata correttamente, garantisce il funzionamento della mutua esclusione. Tuttavia questo metodo di mutua esclusione usa uno **spin lock**, poiché la CPU che esegue la richiesta si pone proprio in un ciclo in cui verifica il lock il più velocemente possibile. Nel farlo, non solo spreca il tempo della (o delle) CPU che esegue (o eseguono) la richiesta, ma mette anche un carico di lavoro molto pesante sul bus o sulla memoria, rallentando seriamente tutte le altre CPU che tentano di svolgere il loro normale lavoro.

A prima vista può sembrare che l'uso della cache possa eliminare il problema della contesa del bus, ma non è così. In teoria, una volta che la CPU che ha fatto la richiesta ha letto la parola di lock, dovrebbe averne una copia nella sua cache. Finché nessun'altra CPU cerca di usare il lock, la CPU richiedente dovrebbe essere in grado di fare a meno della sua cache. Quando la CPU che possiede il lock vi scrive un 1 per rilasciarlo, il protocollo della cache invalida automaticamente tutte le copie di esso nelle cache remote, richiedendo che sia nuovamente prelevato il valore corretto.

Il problema è che le cache operano in blocchi di 32 o 64 byte. Solitamente, le parole che si trovano nelle vicinanze del lock sono necessarie alla CPU che ha il lock. Poiché l'istruzione TSL è una scrittura (dato che modifica il lock), ha bisogno di accesso esclusivo al

blocco della cache contenente il lock; pertanto, ogni TSL invalida il blocco nella cache di chi detiene il lock e ne preleva una copia privata, esclusiva per la CPU che lo ha richiesto. Appena chi detiene il lock tocca una parola in prossimità del lock, il blocco della cache viene spostato sulla sua macchina. Di conseguenza, l'intero blocco della cache contenente il lock viene costantemente spostato avanti e indietro fra il proprietario del lock e chi lo richiede (il lock), generando sul bus ancor più traffico di quanto farebbero le singole letture della parola del lock.

Se ci si potesse liberare di tutte le scritture indotte dalla TSL dal lato del richiedente, sarebbe possibile ridurre sensibilmente il carico di lavoro della cache. Questo obiettivo può essere ottenuto facendo sì che la CPU richiedente esegua prima una lettura per verificare se il lock è libero; solo se il lock appare libero esegue quindi una TSL per acquisirlo effettivamente. Il risultato di questo piccolo cambiamento è che la maggior parte delle interrogazioni avviene ora in lettura invece che in scrittura. Se la CPU che possiede il lock sta solo leggendo le variabili nel medesimo blocco di cache, ciascuna può avere una copia del blocco della cache in modalità condivisa di sola lettura, eliminando tutti i trasferimenti dei blocchi della cache.

Quando alla fine il lock viene liberato, il proprietario esegue una scrittura, che richiede un accesso esclusivo, invalidando così tutte le altre copie nelle cache remote. Alla lettura successiva da parte della CPU richiedente, il blocco della cache sarà ricaricato. Si noti che, se due o più CPU si contendono lo stesso lock, può accadere che entrambe vedano simultaneamente che si è liberato e che eseguano simultaneamente una TSL per acquisirlo. Solo una vi riuscirà, senza quindi che si verifichino condizioni di competizione, poiché l'acquisizione reale viene eseguita da parte dell'istruzione TSL, che è un'istruzione atomica. Vedere che il lock è libero e provare ad acquisirlo immediatamente con una TSL non garantisce di riuscirci. Se la pura lettura ha successo significa solo che potrebbe essere la volta buona per acquisire il lock, ma non dà alcuna garanzia sul successo dell'acquisizione.

Un altro modo per ridurre il traffico del bus è usare il ben noto algoritmo *Ethernet binary exponential backoff* (Anderson, 1990). Al posto di una interrogazione (polling) continua, come nella Figura 2.22, si può inserire un ciclo di ritardo fra le interrogazioni. Inizialmente il ritardo è un'istruzione. Se il lock è ancora occupato, si raddoppia e diventa di due istruzioni, quindi di quattro istruzioni e così via, fino a un certo massimo. Un valore massimo non elevato dà una risposta rapida quando il lock è rilasciato, ma spreca più cicli del bus nel thrashing della cache; un valore massimo elevato riduce il thrashing della cache, ma si rischia di non accorgersi in tempi brevi che il lock è libero. Il *binary exponential backoff* può essere usato con e senza le letture pure precedenti l'istruzione TSL.

Un'idea ancor migliore è assegnare a ciascuna CPU che vuole acquisire un mutex la propria personale variabile di lock da verificare, come mostrato nella Figura 8.11 (Mellor-Crummey e Scott, 1991). Per evitare conflitti la variabile dovrebbe risiedere in un altro blocco di cache inutilizzato. L'algoritmo funziona facendo in modo che una CPU che fallisca nell'acquisizione del lock allochi una variabile di lock e aggiunga se stessa alla fine dell'elenco delle CPU in attesa del lock. Quando l'attuale utilizzatore del lock esce dalla zona critica, libera il lock privato che la prima CPU della lista sta controllando (nella sua cache). Questa CPU entra quindi nella zona critica. Quando ha terminato, libera il lock in uso dal suo successore e così via. Sebbene il protocollo sia in un certo senso complicato (per evitare che due CPU si aggiungano contemporaneamente alla fine dell'elenco), è efficiente ed esente da starvation.

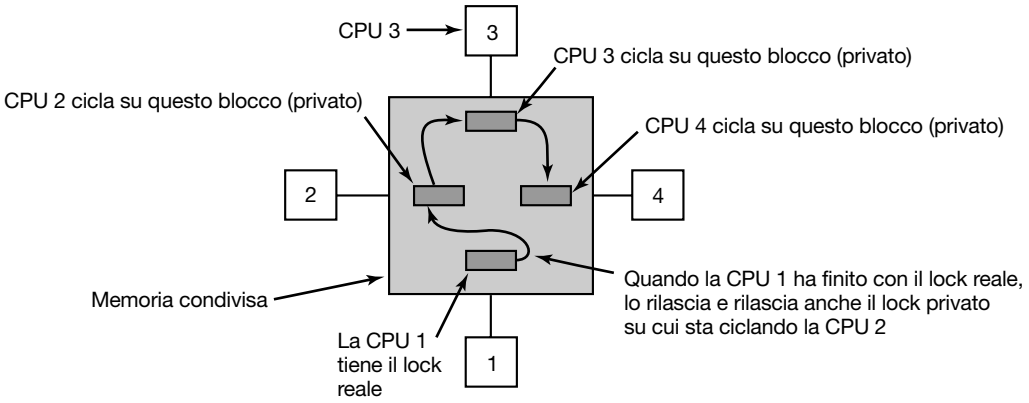

Figura 8.11 Utilizzo di molteplici lock per evitare il thrashing della cache.

Spinning e switching a confronto

Nella sezione precedente si è presupposto che una CPU che ha bisogno di un mutex in stato locked debba solo restarne in attesa, verificando continuamente, a intermittenza o mettendosi in una lista di attesa di CPU. Talvolta non c'è alternativa all'attesa per la CPU richiedente. Per esempio, si supponga che una determinata CPU sia inattiva e debba accedere all'elenco condiviso dei processi disponibili per prelevarne uno da eseguire. Se l'elenco dei processi disponibili è in stato locked, la CPU non può decidere di sospendere quanto sta facendo ed eseguire un altro processo, poiché farlo richiederebbe la lettura dell'elenco. Deve attendere sino a poterlo acquisire.

Tuttavia, in altri casi, c'è una possibilità di scelta. Per esempio, se un thread su una CPU ha bisogno di accedere alla buffer cache del file system ed è attualmente locked, invece di attendere la CPU può decidere di passare a un altro thread. La questione se ciclare (spin) o fare un scambio (switch) di thread è stata oggetto di molta ricerca, parte della quale è trattata nei successivi paragrafi. La questione non si pone su un sistema monoprocessore, dal momento che ciclare non ha senso nel caso in cui non vi sia un'altra CPU che rilascia il lock. Se un thread prova ad acquisire un lock e fallisce, è sempre bloccato per dare al proprietario del lock la possibilità di essere eseguito e rilasciare il lock.

Supponendo che lo spinning e lo scambio di thread siano entrambi opzioni percorribili, il compromesso sta in quanto segue. Lo spinning spreca direttamente dei cicli di CPU; verificare continuamente un lock non è un'attività produttiva. Tuttavia anche lo switch spreca cicli della CPU, dal momento che occorre salvare lo stato del thread attuale, acquisire il lock sull'elenco dei processi pronti, selezionare un thread, caricarne lo stato quindi avviarlo. Inoltre la cache della CPU conterrà tutti blocchi sbagliati e si verificheranno di conseguenza molti costosi fallimenti di cache quando partirà il nuovo thread. Anche gli errori di TLB sono probabili. Alla fine, deve aver luogo uno switch al thread originale, con altre cache miss. I cicli spesi nel realizzare questi due scambi di contesto più tutte le cache miss sarebbero sprecati.

Se si sa che i mutex sono tenuti generalmente per, diciamo, 50 μs e che ci vuole 1 ms per fare lo switch dall'attuale thread e altrettanto per tornare indietro, è più efficiente semplicemente ciclare sul mutex. D'altra parte, se il mutex viene tenuto mediamente per 10 ms, vale la pena fare due scambi di contesto. Il problema è che le zone critiche possono essere molto diverse in termini di durata, per cui qual è l'approccio migliore?

Una possibilità è fare sempre spinning; una seconda prevede di fare sempre lo switch. Una terza possibilità è prendere una decisione diversa ogni volta che si incontra un mutex. Al momento di prendere la decisione, non si sa se sia meglio ciclare o fare lo switch, ma per ogni dato sistema è possibile tener traccia di tutte le attività e analizzarle in seguito. Potrà essere detto a posteriori quale decisione sarebbe stata la più efficiente e quanto tempo sarebbe stato sprecato nella migliore delle ipotesi. Questo algoritmo a posteriori diventa quindi un riferimento con cui misurare gli algoritmi fattibili.

Questo problema è stato studiato dai ricercatori (Karlin et al., 1989; Karlin et al., 1991 e Ousterhout, 1982). La maggior parte dei lavori usa un modello in cui un thread che fallisce nell'acquisizione di un mutex cicla per un certo periodo di tempo; se si supera questo limite, allora passa allo switch. In alcuni casi il limite è fisso, tipicamente l'overhead per lo scambio a un altro thread e il suo ritorno; in altri casi è dinamico, a seconda della storia del mutex in attesa.

I migliori risultati sono ottenuti quando il sistema tiene traccia degli ultimi tempi di spinning osservati e assume che questo sia comunque simile ai precedenti. Per esempio, assumendo ancora un tempo per lo scambio di contesto di 1 ms, un thread dovrebbe ciclare per un massimo di 2 ms, ma occorre osservare per quanto tempo cicla in realtà. Se fallisce nell'acquisizione di un lock e vede che nelle precedenti esecuzioni ha aspettato in media 200 μs, dovrebbe ciclare per 2 ms prima dello switch; tuttavia, se vede che ha ciclato per tutti i 2 ms su ciascuno dei tentativi precedenti, dovrebbe eseguire immediatamente uno swith senza fare alcuno spinning. Ulteriori dettagli possono essere trovati in Karlin et al. (1991).

8.1.4 Scheduling dei multiprocessori

Prima di analizzare come eseguire lo scheduling sui multiprocessori è importante determinare *che cosa* dev'essere schedulato. In passato, quando i processi erano tutti a singolo thread, occorreva schedulare solamente i processi. Tutti i sistemi operativi moderni supportano ora processi multithread, il che rende lo scheduling più complicato.

È importante sapere se i thread sono del kernel o thread utente. Se l'utilizzo dei thread avviene attraverso una libreria dello spazio utente e il kernel non sa nulla riguardo ai thread, allora lo scheduling avviene sulla base del processo, come si è sempre fatto: se il kernel non sa nulla riguardo ai thread difficilmente potrà schedularli.

Con i thread del kernel cambia tutto. In questo caso il kernel è a conoscenza dei thread e può scegliere fra i thread appartenenti a un processo. In questi sistemi si tende a far sì che il kernel prenda il thread da eseguire, mentre il processo a cui il thread appartiene gioca un ruolo piccolo (se non addirittura nullo) nell'algoritmo di selezione del thread. Nel seguito verrà illustrato lo scheduling dei thread, ma in un sistema con processi a singolo thread o con thread implementati nello spazio utente sono naturalmente i processi a essere schedulati.

La questione dello scheduling dei processi in opposizione ai thread non è l'unico problema. Lo scheduling su un monoprocessore è monodimensionale. La sola domanda cui (ripetutamente) rispondere è: "Qual è il prossimo thread da eseguire?". Su un multiprocessore,

lo scheduling ha due dimensioni: lo scheduler deve decidere che thread eseguire e su quale CPU. Questa ulteriore dimensione complica enormemente lo scheduling sui multiprocessori.

Un altro fattore di complicazione è che in alcuni sistemi tutti i thread non sono correlati, mentre in altri sono raccolti in gruppi, tutti appartenenti alla stessa applicazione e che lavorano insieme. Un esempio della prima situazione è un sistema, in cui utenti indipendenti avviano processi indipendenti. I thread di processi differenti non sono correlati e ciascuno può essere schedulato senza considerare gli altri.

Un esempio della seconda situazione si verifica regolarmente negli ambienti di sviluppo dei programmi. I grandi sistemi spesso sono composti da un certo numero di file d'intestazione contenenti macro, definizioni dei tipi e dichiarazioni delle variabili usate dai file sorgenti reali. Quando si modifica un file d'intestazione, tutti i file sorgenti che lo includono devono essere ricompilati. Solitamente, per gestire lo sviluppo si utilizza il programma *make*. Quando si richiama *make*, questo avvia la compilazione dei soli file sorgenti che devono essere ricompilati a causa di modifiche apportate ai file d'intestazione o di codice, mentre i file oggetto ancora validi non vengono ricompilati.

La versione originale di *make* eseguiva il lavoro sequenzialmente, mentre le nuove versioni progettate per i multiprocessori possono avviare tutte le compilazioni in una volta. Se occorrono dieci compilazioni, non ha senso schedularne nove da eseguire immediatamente e lasciare l'ultima in seguito, dato che l'utente percepirà il lavoro come non completato finché non sarà terminata l'ultima compilazione. In questo caso ha più senso considerare i thread che fanno la compilazione come un gruppo e tenerne conto quando li si schedula.

Time sharing

Consideriamo per primo il caso dello scheduling di thread indipendenti; più avanti considereremo lo scheduling di thread correlati. L'algoritmo di scheduling più semplice per gestire thread non correlati è avere una sola struttura dati a livello dell'intero sistema per i thread pronti, presumibilmente solo un elenco, ma più probabilmente un insieme di elenchi per thread con diverse priorità, come rappresentato nella Figura 8.12(a). In questo caso le 16 CPU sono attualmente impegnate e vi è un insieme con priorità di 14 thread in attesa. La prima CPU a terminare il lavoro che sta eseguendo (o il cui thread viene bloccato) è la CPU 4, che poi esegue un lock sulle code di scheduling e seleziona il thread con la priorità più alta, *A*, come mostrato nella Figura 8.12(b). Subito dopo diventa inattiva la CPU 12, che sceglie il thread *B*, come illustrato nella Figura 8.12(c). Finché i thread non sono in relazione gli uni con gli altri, la scelta di eseguire lo scheduling in questo modo è ragionevole ed è facile implementarla in modo efficiente.

Il fatto di avere una sola struttura dati di scheduling usata da tutte le CPU ripartisce il tempo tra le CPU come se fossero in un sistema monoprocessore e fornisce altresì un sistema automatico di bilanciamento del carico, perché non accade mai che una CPU sia inattiva mentre le altre sono sovraccariche. I due svantaggi di questo approccio sono le potenziali dispute per la struttura dati di scheduling al crescere del numero delle CPU e l'usuale overhead nel fare lo scambio di contesto quando un thread si blocca per l'I/O.

Si può verificare uno scambio di contesto anche quando scade il quantum di un thread. Su un multiprocessore che ha caratteristiche che non esistono su un monoprocessore. Si supponga che un thread detenga uno spin lock quando termina il proprio quantum. Le altre CPU in attesa dello spin lock sprecano tempo ciclando finché il thread non viene

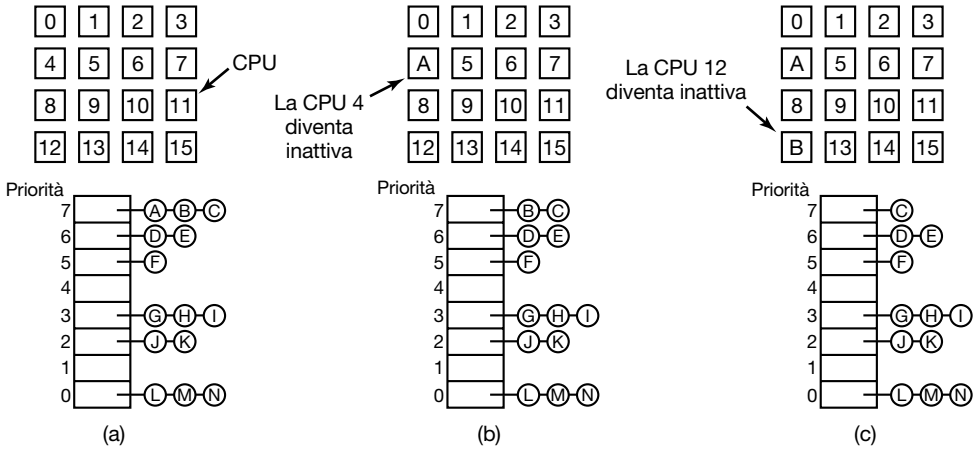

Figura 8.12 Utilizzo di una sola struttura dati per lo scheduling di un multiprocessore.

nuovamente schedulato e rilascia il lock. Su un monoprocessore gli spin lock sono usati raramente, quindi, se un processo è sospeso mentre trattiene un mutex e parte un altro thread che prova ad acquisire il mutex, sarà immediatamente bloccato, così da sprecare poco tempo.

Per aggirare tale anomalia, alcuni sistemi usano lo **smart scheduling**, in cui un thread che acquisisce uno spin lock imposta un flag a livello di processo per indicare che ha attualmente uno spin lock (Zahorjan et al., 1991). Quando rilascia il lock, azzera l'indicatore. Lo scheduler non ferma un thread che ha uno spin lock, ma gli dà invece un altro po' di tempo per completare la sua zona critica e rilasciare il lock.

Un'altra questione che gioca un ruolo importante nello scheduling è il fatto che, se è pur vero che tutte le CPU sono uguali, alcune lo sono di più. In particolare, quando il thread *A* è stato eseguito a lungo sulla CPU *k*, la cache della CPU *k* sarà piena di blocchi di *A.* Se *A* viene nuovamente eseguito a breve, potrebbe essere preferibile che sia eseguito di nuovo sulla CPU *k*, poiché la cache contiene dei blocchi di *A*: avere dei blocchi precaricati migliorerà le prestazioni della cache e quindi la velocità del thread. Inoltre il TLB potrebbe anche contenere la pagine giuste, riducendo gli errori del TLB.

Alcuni multiprocessori tengono conto di questo effetto e usano quello che è chiamato **scheduling per affinità** (Vaswani e Zahorjan, 1991). In questo caso, l'idea di base è di fare ogni sforzo possibile affinché un thread sia eseguito sulla stessa CPU su cui è stato eseguito l'ultima volta. Un modo per creare questa affinità è l'uso di un **algoritmo di scheduling a due livelli**. Quando il thread viene creato, è assegnato a una CPU, selezionando per esempio quella con il minor carico in quel momento. Questa assegnazione di thread alle CPU è il livello superiore (top-level) dell'algoritmo. In conseguenza a questa policy ogni CPU acquisisce il proprio insieme di thread.

Lo scheduling reale dei thread è il livello inferiore (bottom-level) dell'algoritmo. È fatto separatamente da ogni CPU, usando le priorità o altri sistemi. Se si cerca di mantenere un thread sempre sulla stessa CPU si massimizza l'affinità della cache; se tuttavia una CPU non ha thread da eseguire, piuttosto che diventare inattiva ne prende uno da un'altra CPU.

Lo scheduling a due livelli presenta tre vantaggi. Primo, distribuisce il carico abbastanza equamente fra le CPU disponibili. Secondo, dove possibile si usufruisce dell'affinità della cache. Terzo, dando a ciascuna CPU la sua lista, poiché i tentativi di utilizzo della lista di thread pronti di un'altra CPU sono relativamente poco frequenti, le dispute per le liste di thread pronti sono ridotte al minimo.

Space sharing

L'altro approccio generale allo scheduling dei multiprocessori può essere usato quando i thread sono in qualche modo in relazione tra loro. In precedenza abbiamo menzionato l'esempio del *make* parallelo come uno di questi casi. Accade anche spesso che un singolo processo abbia più thread che lavorano insieme. Per esempio, se i thread di un processo comunicano molto fra loro, è utile che siano in esecuzione nello stesso momento. Lo scheduling di più thread nello stesso tempo su più CPU è detto **space sharing**.

L'algoritmo più semplice di space sharing funziona in questo modo. Si supponga che venga creato un intero gruppo di thread in relazione fra loro. Nel momento in cui è creato, lo scheduler verifica se vi sono tante CPU libere quanti sono i thread. Se sì, ogni thread è assegnato alla sua CPU dedicata (cioè non multiprogrammata) quindi viene eseguito. Se non ci sono abbastanza CPU, non parte alcun thread finché non ci sono CPU a sufficienza. Ciascun thread resta nella propria CPU finché non termina, al che la CPU viene inserita all'interno dell'insieme delle CPU disponibili. Se un thread si ferma su un I/O, continua a tenere la CPU, che è semplicemente inattiva fino al risveglio del thread. Con il blocco di thread successivo si applica lo stesso algoritmo.

In un qualunque momento, l'insieme delle CPU viene suddiviso staticamente in un determinato numero di partizioni, ciascuna delle quali esegue i thread di un processo. Nella Figura 8.13 sono presenti partizioni con dimensioni di 4, 6, 8 e 12 CPU, con 2 CPU non assegnate, per esempio. Col passar del tempo, il numero e la dimensione delle partizioni cambiano, a mano a mano che vengono creati nuovi thread e quelli vecchi terminano.

Occorre prendere periodicamente decisioni riguardanti lo scheduling. Nei sistemi monoprocessore si usa spesso l'algoritmo *shortest job first* (prima il lavoro più breve) per il batch scheduling. L'algoritmo analogo nei sistemi multiprocessore consiste nello scegliere il processo che ha bisogno del minor numero di cicli di CPU, ossia il thread in cui cicli di CPU × il tempo di esecuzione siano i più bassi fra i candidati. In realtà, solo di rado questa informa-

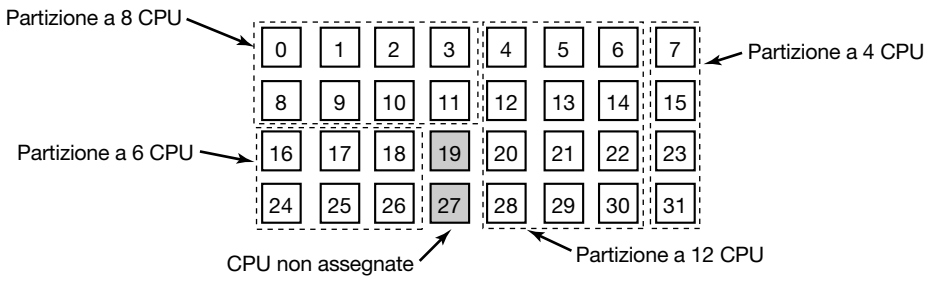

Figura 8.13 Un insieme di 32 CPU diviso in quattro partizioni, con 2 CPU disponibili.

zione è disponibile; pertanto l'algoritmo non è sempre implementabile. Studi hanno in effetti dimostrato che nella pratica è difficile superare le prestazioni di un algoritmo *first-come, first-served* (Krueger et al., 1994).

In questo semplice schema di partizionamento un thread non fa altro che richiedere un certo numero di CPU finché non le ottiene, oppure aspetta che si rendano disponibili. Diverso è l'approccio per i thread che gestiscono attivamente il livello di parallelismo. Un metodo per gestire il parallelismo è avere un server centrale che tiene traccia di tutti i thread in esecuzione e che vogliono essere eseguiti e di quali siano i loro requisiti massimi e minimi di CPU (Tucker e Gupta, 1994). Ogni applicazione interroga periodicamente il server centrale per sapere quante CPU potrebbe usare. A seconda di quante ne ha a disposizione, il numero di thread aumenta o diminuisce. Per esempio un Web server può avere 5, 10, 15, 20 o qualunque altro numero di thread eseguiti in parallelo. Se attualmente ha 10 thread e vi è un'improvvisa richiesta di CPU ed è avvisato di rilasciarne 5, i primi cinque che finiranno il loro attuale lavoro, invece di vedersene assegnato uno nuovo, saranno avvisati di terminare l'esecuzione. Questo schema permette alle dimensioni delle partizioni di variare dinamicamente per adattarsi meglio all'attuale carico di lavoro, piuttosto che nel caso del sistema statico della Figura 8.13.

Gang scheduling

Un chiaro vantaggio dello space sharing è l'eliminazione della multiprogrammazione, che elimina l'overhead dovuto allo scambio di contesto. Uno svantaggio altrettanto chiaro è il tempo sprecato quando si blocca una CPU e non c'è nulla da fare finché non è nuovamente pronta. Di conseguenza si sono cercati algoritmi che tentassero lo scheduling di tempo e spazio insieme, in particolare per i thread che creano molteplici thread, che solitamente hanno necessità di comunicare con un altro thread.

Per vedere il tipo di problema che si può verificare quando i thread di un processo sono schedulati indipendentemente considerate un sistema con i thread A_0 e A_1 appartenenti al processo A e i thread B_0 e B_1 appartenenti al processo *B*. I thread A_0 e B_0 sono in time sharing sulla CPU 0; i thread A_1 e B_1 sono in time sharing sulla CPU 1. I thread A_0 e A_1 hanno bisogno di comunicare spesso. Lo schema di collegamento è che A_0 invia un messaggio ad A_1 e A_1 rimanda indietro la risposta ad A_0, seguito da un'altra sequenza. Supponete che per fortuna A_0 e B_1 partano per primi, come illustrato nella Figura 8.14.

Al momento 0, A_0 invia una richiesta ad A_1, ma A_1 non la riceve finché non è eseguito, nel momento 1, che comincia a 100 ms. Invia la risposta immediatamente, ma A_0 non la riceverà finché non sarà rieseguito a 200 ms. Il risultato netto è una sequenza domanda-risposta ogni 200 ms. Quindi con prestazioni non molto buone.

La soluzione a questo problema è il **gang scheduling** (*scheduling di squadra*), estensione del co-scheduling (Ousterhout, 1982). Il gang scheduling ha tre parti.

1. I gruppi di thread correlati sono schedulati come un'unità, una squadra (gang).
2. Tutti i membri di una squadra sono eseguiti contemporaneamente, su CPU differenti, in condizioni di timesharing.
3. Tutti i membri della squadra entrano ed escono dai loro intervalli di tempo assegnati insieme.

Il trucco che fa funzionare il gang scheduling è che tutte le CPU siano schedulate in sincronia. Ciò significa che il tempo è suddiviso in quantum distinti come abbiamo visto nella Figura 8.14. All'inizio di ciascun nuovo quantum tutte le CPU sono rischedulate, con un nuovo thread che parte su ognuna ogni CPU. All'inizio del quantum successivo avviene un altro evento di scheduling. Nel mezzo, niente scheduling. Se un thread si blocca, la sua CPU rimane inattiva sino alla fine del quantum.

Un esempio di come funzioni il gang scheduling è dato nella Figura 8.15. In questo caso abbiamo un multiprocessore con 6 CPU usate da 5 processi, da A a E, per un totale di 24 thread pronti. Durante il periodo di tempo 0 sono schedulati ed eseguiti i thread da A_0 a A_6. Durante il periodo di tempo 1 sono schedulati ed eseguiti i thread B_0, B_1, B_2, C_0, C_1 e C_2. Durante il periodo di tempo 2 è la volta dell'esecuzione dei 5 thread D e di E_0. I restanti 6 thread appartenenti a E sono eseguiti nell'intervallo 3. A questo punto il ciclo si ripete, con il periodo 4 uguale al periodo 0 e così via.

L'idea del gang scheduling è far sì che tutti i thread di un thread siano eseguiti insieme, in modo che se uno di loro invia una richiesta a un altro, riceverà il messaggio quasi immediatamente e potrà rispondergli quasi immediatamente. Nella Figura 8.15, dato che tutti i thread A sono eseguiti insieme, durante un quantum, possono spedire e ricevere una gran quantità di messaggi, eliminando così i problemi della Figura 8.14.

8.2 Multicomputer

I multiprocessori sono apprezzati e attrattivi perché offrono un modello di comunicazione semplice: tutte le CPU condividono una memoria comune. I processi possono scrivere messaggi nella memoria leggibili da altri processi. La sincronizzazione può essere fatta usando mutex, semafori, monitor o altre tecniche ben definite. Il solo difetto che rovina il tutto è che i grandi multiprocessori sono difficili da costruire e perciò sono costosi.

Per aggirare questi problemi è stata fatta molta ricerca sui **multicomputer**, con CPU fortemente accoppiate che non condividono la memoria. Ciascun computer ha una propria CPU, come mostrato nella Figura 8.1(b). Questi sistemi sono anche conosciuti sotto una varietà di altri nomi, incluso **computer cluster** e **COWS** (**clusters of workstations**).

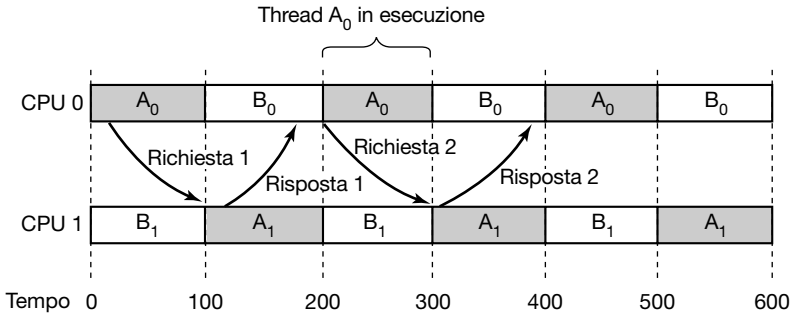

Figura 8.14 Comunicazioni tra due thread appartenenti al thread A in esecuzione fuori fase.

Periodo di tempo	CPU 0	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

Figura 8.15 Gang scheduling (scheduling di squadra).

I multicomputer sono semplici da costruire perché la componente base è un computer spoglio con l'aggiunta di una scheda di comunicazione di rete. Naturalmente il segreto nell'ottenere alte prestazioni sta nel progettare l'interconnessione di rete e le schede d'interfaccia in modo intelligente. Questo problema è del tutto analogo alla costruzione della memoria condivisa in un multiprocessore. Tuttavia l'obiettivo è l'invio di messaggi in un tempo dell'ordine dei microsecondi, piuttosto che l'accesso alla memoria in un tempo dell'ordine dei nanosecondi, quindi più semplice, più economico e semplice da realizzare.

Nei paragrafi seguenti analizzeremo in principio l'hardware dei multicomputer, specialmente le interconnessioni hardware. Passeremo poi al software, cominciando con il software di comunicazione di basso livello, poi con quello di alto livello. Analizzeremo anche il modo in cui si può ottenere la memoria condivisa su sistemi che non ne hanno. Alla fine esamineremo lo scheduling e la ripartizione del carico di lavoro (load balancing).

8.2.1 Hardware dei multicomputer

Il nodo base di un multicomputer è composto da una CPU, una memoria, un'interfaccia di rete e talvolta un disco fisso. Il nodo può essere assemblato come un PC standard, ma adattatore grafico, monitor, tastiera e mouse sono quasi sempre assenti. In alcuni casi il PC contiene una scheda multiprocessore a 2 o 4 vie, presumibilmente ciascuna con un chip dual-core o quad-core, invece di una singola CPU; per semplificare, considereremo tuttavia che ogni nodo abbia una CPU. Spesso, centinaia o migliaia di nodi sono collegati a formare un multicomputer. Nel seguito parleremo di com'è organizzato questo hardware.

Tecnologia d'interconnessione

Ogni nodo ha una scheda d'interfaccia di rete con uno o due cavi (o fibre) che fuoriescono. Questi cavi si connettono o ad altri nodi o a degli switch. In un sistema piccolo potrebbe esserci uno switch cui tutti i nodi si collegano nella topologia a stella della Figura 8.16(a). I moderni switch Ethernet usano questa topologia.

In alternativa alla progettazione con un singolo switch, i nodi possono formare un anello, con due cavi che escono dalla scheda d'interfaccia di rete, uno verso il nodo di destra e uno verso il nodo a sinistra, come illustrato nella Figura 8.16(b). In questa topologia non è necessario alcuno switch.

La **griglia** o **maglia** della Figura 8.16(c) è uno schema bidimensionale usato in molti sistemi commerciali. È estremamente regolare e facile da scalare a grandi dimensioni. Ha un **diametro**, che è il percorso più lungo fra due nodi e che aumenta solo come la radice quadrata del numero dei nodi. Una variante alla griglia è il **doppio toro** della Figura 8.16(d), una griglia con le estremità che si congiungono. Non solo è più tollerante ai malfunzionamenti della griglia, ma anche il diametro è minore, visto che gli angoli opposti sono congiunti in soli due salti.

Il **cubo** della Figura 8.16(e) è una topologia tridimensionale regolare. Abbiamo illustrato un cubo di $2 \times 2 \times 2$, ma nel caso più generale possibile potrebbe essere un cubo $k \times k \times k$. Nella Figura 8.16(f) abbiamo un cubo a quattro dimensioni costruito con due cubi tridimensionali con i nodi corrispondenti collegati. Potremmo fare un cubo a cinque dimensioni clonando la struttura della Figura 8.16(f) e connettendo i nodi corrispondenti a formare un blocco di cinque cubi. Per ottenere le sei dimensioni potremmo replicare il blocco di cinque cubi e connettere i nodi corrispondenti e così via. Un cubo n-dimensionale costruito in questo modo è denominato **ipercubo**. Molti computer paralleli usano questa topologia perché il diametro cresce linearmente con la dimensionalità. In altre parole, il diametro è il logaritmo in base 2 del numero dei nodi, così per esempio un ipercubo a 10

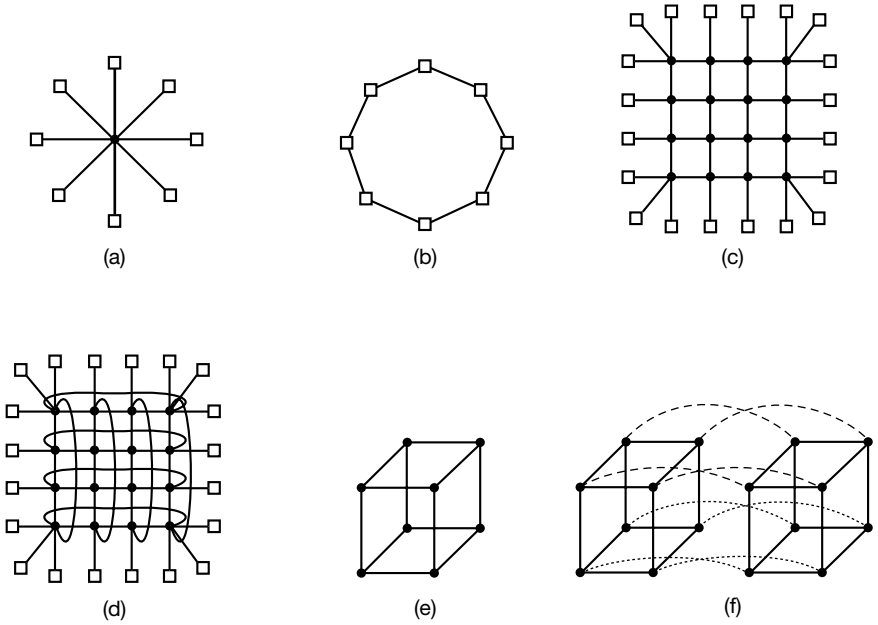

Figura 8.16 Varie topologie d'interconnessione. (a) Un singolo switch. (b) Un anello. (c) Una griglia. (d) Un doppio toro. (e) Un cubo. (f) Un ipercubo a 4D.

dimensioni ha 1024 nodi, ma un diametro solamente di 10, con eccellenti proprietà dal punto di vista del ritardo. Notate che, diversamente, 1024 nodi configurati come una griglia di 32 × 32 hanno un diametro di 62, più di sei volte peggio dell'ipercubo. Il prezzo pagato per il diametro ridotto è che il fanout e quindi il numero di collegamenti (e il costo) è molto più grande per l'ipercubo.

Nei multicomputer sono usati due tipi di schemi di switching. Nel primo ogni messaggio è prima spezzato (o dal software utente o dall'interfaccia di rete) in un elemento di una certa lunghezza massima, chiamato **pacchetto**. Nello schema di switching chiamato **smistamento di pacchetti store-and-forward**, la scheda d'interfaccia di rete del nodo sorgente inserisce il pacchetto nel primo switch, come da Figura 8.17(b). Quando il pacchetto arriva allo switch connesso al nodo di destinazione, come nella Figura 8.17(c), il pacchetto è copiato nella scheda d'interfaccia di rete del nodo destinatario ed eventualmente nella sua RAM.

Mentre da una parte lo smistamento di pacchetti store-and-forward è flessibile ed efficiente, dall'altra ha il problema dell'incremento dei tempi di latenza (il ritardo) lungo la rete d'interconnessione. Supponete che il tempo per spostare un pacchetto di un salto sia di T ns. Dato che il pacchetto per andare dalla CPU 1 alla CPU 2 dev'essere copiato quattro volte (verso A, verso C, verso D e alla CPU di destinazione) e che nessuna copia può iniziare prima che sia terminata la precedente, la latenza lungo la rete d'interconnessione è di $4T$. Una via d'uscita è di progettare una rete in cui un pacchetto possa essere diviso in unità più piccole. Appena arriva la prima unità allo switch, può essere inoltrata, anche prima che sia arrivata la coda. Concettualmente l'unità potrebbe essere piccola quanto un bit.

Nell'altro regime di switching, il **circuit switching**, il primo switch anzitutto stabilisce un percorso attraverso tutti gli switch, sino allo switch finale. Una volta che il percorso è stato definito, i bit sono immessi lungo tutta la strada dal sorgente al destinatario senza interruzione e più velocemente possibile. Non c'è alcuna azione di buffer intermedia negli

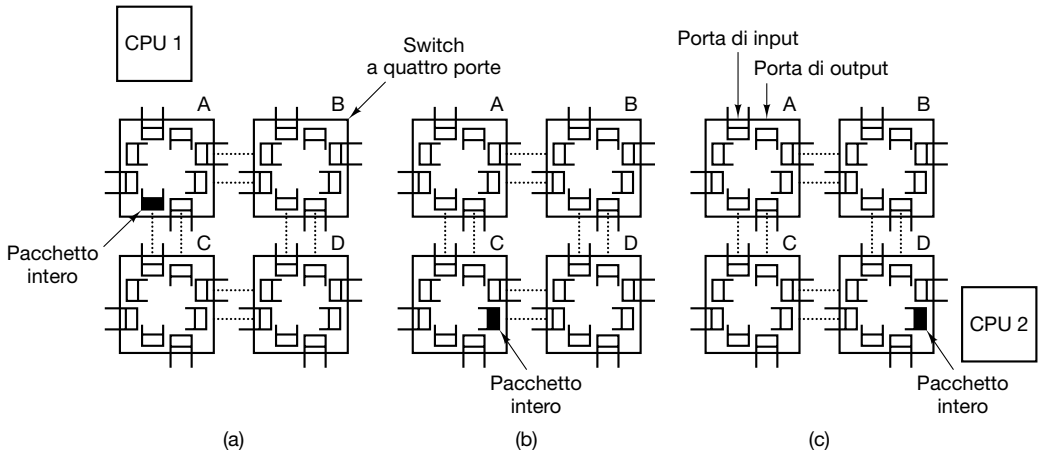

Figura 8.17 Smistamento di pacchetti store-and-forward.

switch che intervengono. Il circuit switching richiede una fase preparatoria che richiede del tempo, ma una volta che questa è terminata è più veloce. Dopo che il pacchetto è stato spedito, il percorso deve essere nuovamente staccato. Una variazione al circuit switching, detta **wormhole routing** spezza ciascun pacchetto in sottopacchetti e consente al primo sottopacchetto di iniziare l'invio prima che l'intero percorso sia stato costruito.

Interfacce di rete

In un multicomputer tutti i nodi hanno installata una scheda contenente la connessione del nodo alla rete d'interconnessione che tiene insieme i computer. Il modo in cui queste schede sono costruite e come si connettono alla CPU principale e alla RAM ha implicazioni considerevoli per il sistema operativo. Esamineremo adesso brevemente alcune delle questioni. Il materiale è in parte basato su Bhoedjang (2000).

Praticamente in tutti i multicomputer la scheda d'interfaccia contiene della RAM in quantità sufficiente per contenere i pacchetti in uscita e in entrata. Solitamente, un pacchetto in uscita dev'essere copiato nella RAM della scheda d'interfaccia prima della trasmissione al primo switch. La ragione per questa progettazione è che molte reti d'interconnessione sono sincrone, così una volta che il pacchetto è partito, i bit devono continuare a fluire a velocità costante. Se il pacchetto è nella RAM principale il continuo flusso d'uscita sulla rete non può essere garantito a causa del resto del traffico sul bus della memoria. Usando una RAM dedicata sulla scheda d'interfaccia si elimina questo problema. Questa progettazione è illustrata nella Figura 8.18.

Il medesimo problema si presenta con i pacchetti in ingresso. I bit arrivano dalla rete a velocità costante e spesso a velocità estremamente alta. Se la scheda d'interfaccia di rete non può memorizzarli in tempo reale quando arrivano, i dati andranno persi. Anche in questo caso, tentare di passare attraverso il bus di sistema (per esempio, il bus PCI) per andare alla RAM principale è troppo rischioso. Dato che la scheda di rete è normalmente inserita sul bus PCI, questa è la sola connessione che ha con la RAM principale, perciò la

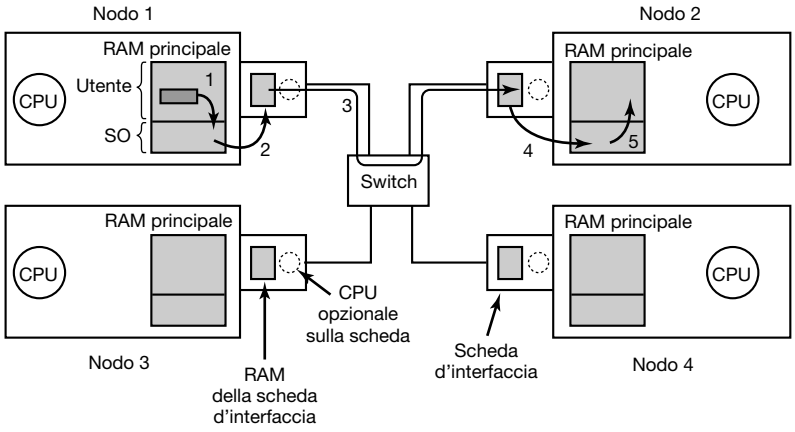

Figura 8.18 Posizione delle schede d'interfaccia di rete in un multicomputer.

competizione per questo bus con il disco e tutti gli altri dispositivi di I/O è inevitabile. È più sicuro memorizzare i pacchetti entranti nella RAM privata della scheda d'interfaccia e poi copiarli nella RAM principale in seguito.

La scheda d'interfaccia può avere uno o più canali DMA o anche una CPU completa (o anche più di una CPU) sulla scheda. I canali DMA possono copiare i pacchetti fra la scheda d'interfaccia e la RAM principale ad alta velocità, richiedendo i trasferimenti dei blocchi sul bus di sistema, trasferendo così molte parole senza dover richiedere il bus separatamente per ciascuna parola. Tuttavia è esattamente questo tipo di trasferimento dei blocchi, che occupa il bus di sistema per molti cicli del bus, che rende la RAM sulla scheda d'interfaccia necessaria in primo luogo.

Molte schede d'interfaccia hanno una CPU completa su di loro, presumibilmente in aggiunta a uno o più canali DMA. Sono chiamate **processori di rete** e stanno diventando sempre più potenti. Questo schema significa che la CPU principale può scaricare del lavoro sulla schede di rete, come la gestione di una trasmissione affidabile (nel caso l'hardware perdesse dei pacchetti), il multicasting (l'invio di un pacchetto a più di un destinatario), la compressione e la decompressione, la crittazione e la descrittazione e il prendersi cura della protezione in un sistema con molteplici processi. Tuttavia avere due CPU significa che debbono sincronizzarsi per evitare le condizioni competitive, il che aggiunge ulteriore overhead e significa più lavoro per il sistema operativo.

8.2.2 Software di comunicazione di basso livello

Il nemico delle comunicazioni ad alte prestazioni nei sistemi di multicomputer è l'eccessiva copia dei pacchetti. Nel migliore dei casi ci sarà una copia dalla RAM alla scheda d'interfaccia nel nodo sorgente, una copia dalla scheda d'interfaccia sorgente alla scheda d'interfaccia destinataria (se non avvengono memorizzazioni e inoltri lungo il percorso) e una copia da lì alla RAM destinataria, per un totale di tre copie. In alcuni sistemi è tuttavia ancora peggio. In particolare se la scheda d'interfaccia è mappata nello spazio degli indirizzi virtuali del kernel e non nello spazio degli indirizzi virtuali dell'utente, un processo utente può solo spedire un pacchetto inviando una chiamata di sistema che esegua un trap nel kernel. I kernel potrebbero dover copiare i pacchetti nella loro memoria sia in output sia in input, per esempio, per evitare errori di pagine durante la trasmissione sulla rete. Inoltre il kernel ricevente probabilmente non sa dove mettere i pacchetti in ingresso finché non ha la possibilità di analizzarli. Questi cinque passaggi di copia sono illustrati nella Figura 8.18.

Se le copie verso e dalla RAM sono colli di bottiglia, le copie extra verso e dal kernel possono raddoppiare il ritardo da un capo all'altro e dimezzare l'efficienza della trasmissione. Per evitare questo problema prestazionale, tanti multicomputer mappano la scheda d'interfaccia direttamente nello spazio utente e consentono al processo utente di mettere direttamente i pacchetti sulla scheda, senza coinvolgimento del kernel. Mentre da un lato questo approccio aiuta sicuramente le prestazioni, dall'altro introduce due problemi.

Primo, che cosa accade se molti processi sono in esecuzione sul nodo e c'è bisogno di accesso alla rete per spedire i pacchetti? Quale di loro si prende la scheda d'interfaccia nel proprio spazio degli indirizzi? Il fatto di avere una chiamata di sistema che mappa la scheda in ingresso e in uscita da uno spazio degli indirizzi virtuali è costoso, ma se un solo processo prende la scheda, come fanno gli altri a mandare i pacchetti? E che cosa accade se la scheda

è mappata nello spazio degli indirizzi virtuali del processo *A* e arriva un pacchetto per il processo *B*, specialmente se *A* e *B* hanno proprietari diversi, nessuno dei quali vuole fare lo sforzo di aiutare l'altro?

Una soluzione è quella di mappare la scheda d'interfaccia in tutti i processi che ne hanno bisogno, ma si rende poi necessario un meccanismo che eviti le condizioni di contesa. Per esempio, se *A* richiede un buffer sulla scheda d'interfaccia e poi, a causa dello scadere del suo intervallo di tempo, *B* esegue e richiede lo stesso buffer, il risultato è un disastro. Serve un meccanismo di sincronizzazione, ma questi meccanismi, come i mutex, funzionano solo quando si presuppone che i processi siano cooperativi. In un ambiente di *time sharing* con molteplici utenti tutti frettolosi di avere il loro lavoro concluso, un utente potrebbe semplicemente mettere in lock il mutex associato alla scheda e non rilasciarlo più. La conclusione in questo caso è che mappare la scheda d'interfaccia all'interno dello spazio utente funziona bene solo quando c'è un solo processo utente in esecuzione su ciascun nodo, a meno che non siano prese speciali precauzioni (per esempio che processi differenti mappino nei loro spazi degli indirizzi porzioni diverse della RAM dell'interfaccia).

Il secondo problema è che anche il kernel stesso potrebbe aver bisogno di accedere alla rete d'interconnessione stessa, per esempio per accedere al file system su un nodo remoto. Che il kernel condivida la scheda d'interfaccia con qualsiasi utente non è una buona idea, anche sulla base del time sharing. Supponete che mentre la scheda sia mappata nello spazio utente, arrivi un pacchetto del kernel. O supponete che un processo utente spedisca un pacchetto a una macchina remota fingendo di essere il kernel. La conclusione è che la progettazione più semplice è quella che prevede due schede d'interfaccia di rete, una mappata nello spazio utente per il traffico applicativo e l'altra mappata nello spazio del kernel per l'uso tramite il sistema operativo. Molti multicomputer fanno esattamente questo.

Comunicazione tra nodi e interfaccia di rete

Un'altra questione è come portare i pacchetti nelle schede d'interfaccia. Il modo più veloce è quello di usare il chip DMA sulla scheda per copiarli semplicemente dalla RAM. Il problema di un simile approccio è che il DMA usa indirizzi fisici piuttosto che virtuali ed è eseguito indipendentemente dalla CPU. Innanzitutto, sebbene un processo utente conosca certamente l'indirizzo virtuale di ogni pacchetto che voglia spedire, generalmente non conosce l'indirizzo fisico. Effettuare una chiamata di sistema per fare un mappaggio da virtuale a fisico non è auspicabile, dato che il fatto di mettere la scheda d'interfaccia nello spazio utente è per evitare di aver una chiamata di sistema per ciascun pacchetto inviato.

Inoltre, se il sistema operativo decide di sostituire una pagina mentre il chip DMA sta copiando un pacchetto da essa, saranno trasmessi i dati sbagliati. Ancor peggio, se il sistema operativo sostituisce una pagina mentre il chip DMA sta copiando su di essa un pacchetto in ingresso, non solo sarà perso il pacchetto in ingresso, ma si rovinerà anche un'innocente pagina di memoria.

Questi problemi sono evitabili con delle chiamate di sistema che annotano e spuntano le pagine in memoria, contrassegnandole come temporaneamente "non paginabili". Tuttavia, il fatto di avere una chiamata di sistema che contrassegni la pagina contenente ogni pacchetto in uscita e poi avere una chiamata di sistema per l'operazione inversa è oneroso. Se i pacchetti sono piccoli, diciamo 64 byte o meno, l'overhead per contrassegnare e poi spuntare ogni buffer è proibitivo. Per pacchetti grandi, diciamo da 1 KB e oltre, può essere tollerabile. Per

le dimensioni a metà finisce per dipendere dai dettagli dell'hardware. Oltre a introdurre un problema prestazionale, l'operazione di aggiungere e togliere questo contrassegno alle pagine complica ulteriormente il software.

DMA remoto

In alcuni campi, elevate latenze di rete non sono assolutamente accettabili; per alcune applicazioni ad elevate prestazioni il tempo di elaborazione dipende pesantemente dalla latenza di rete. Il trading ad alta frequenza, per esempio, richiede che i computer eseguano le transazioni (compravendita di azioni) a velocità estremamente alte; anche i microsecondi sono importanti. In questa sede non ci interessa giudicare se sia saggio oppure no far sì che dei programmi possano scambiare, nel giro di pochi millisecondi, milioni di dollari di azioni, visto soprattutto che, in generale, il software tende ad avere dei bug; la questione riguarda più i filosofi, giusto quando, durante un pasto, non sono impegnati a tenere in mano la forchetta. Il punto è invece riuscire a mantenere bassa la latenza.

In questi casi è utile mantenere più bassa possibile la quantità di copie; per questo motivo, alcune interfacce di rete supportano l'**RDMA (Remote Direct Memory Access)**, una tecnica che consente di eseguire un accesso diretto alla memoria da un computer a un altro. L'RDMA non comporta un intervento del sistema operativo; i dati vengono passati direttamente dalla memoria dell'applicazione, o scritti su di essa.

L'RDMA è una tecnica eccellente, che tuttavia presenta alcuni svantaggi. Esattamente come il DMA, il sistema operativo sui nodi in comunicazione deve contrassegnare le pagine coinvolte nello scambio di dati; inoltre, inserire i dati nella memoria di un computer remoto non riduce molto la latenza, se l'altro programma non se ne accorge. Un RDMA efficiente non implica necessariamente una notifica esplicita. Una possibile soluzione è che il ricevente interroghi un byte nella memoria; non appena il trasferimento è completato, il mittente modifica il byte per segnalare al ricevente che ci sono nuovi dati. La soluzione funziona, ma non è l'ideale e spreca cicli di CPU.

Per il trading ad altissima frequenza, le schede di rete vengono realizzate per questo scopo utilizzando dei *field programmable gate array*. Hanno una latenza determinata dal mezzo fisico: dalla ricezione dei bit sulla scheda di rete alla trasmissione di un messaggio che comunica di acquistare qualche milione di azioni passa meno di un microsecondo. Acquistare un milione in azioni in 1 μsec rappresenta una prestazione pari a 1 teradollaro/sec, il che va benissimo se si riescono a prendere per il verso giusto i guadagni e le perdite, ma non è un'attività adatta a chi soffre di cuore. I sistemi operativi non hanno un peso determinante in ambienti di elaborazione così estremi.

8.2.3 Software di comunicazione a livello utente

I processi su CPU diverse di un multicomputer comunicano inviandosi l'un l'altro dei messaggi. Nella forma più semplice, lo scambio di messaggi è esposto ai processi utente. In altre parole, il sistema operativo fornisce un modo per inviare e ricevere messaggi e le procedure delle librerie rendono queste chiamate sottostanti disponibili ai processi utente. In una forma più sofisticata, il reale passaggio dei messaggi è nascosto agli utenti facendo sì che le comunicazioni remote appaiano come delle chiamate di procedure. Studieremo nel seguito entrambi questi metodi.

Invio e ricezione

Come minimo, i servizi di comunicazione forniti possono essere ridotti a due chiamate (di librerie), una per l'invio dei messaggi e una per la ricezione. La chiamata per l'invio potrebbe essere

```
send(dest, &mptr);
```

e quella per la ricezione potrebbe essere

```
receive(addr, &mptr);
```

La prima invia il messaggio puntato da *mptr* a un processo identificato da *dest* e causa il blocco del chiamante finché il messaggio non è stato inviato. La seconda causa il blocco del chiamante finché il messaggio non arriva. Quando ne arriva uno, il messaggio è copiato nel buffer puntato da *mptr* e il chiamante è sbloccato. Il parametro *addr* specifica l'indirizzo a cui il ricevente è in ascolto. Sono possibili molte varianti a queste due procedure e ai loro parametri.

Un problema è come è fatto l'indirizzamento. Poiché tanti multicomputer sono statici, con un numero fisso di CPU, il modo più semplice per gestire l'indirizzamento è quello di rendere *addr* un indirizzo di due parti costituite dal numero della CPU e dal numero del processo o della porta sulla CPU indirizzata. Così facendo ogni CPU può gestire i suoi indirizzi senza potenziali conflitti.

Chiamate bloccanti e chiamate non bloccanti a confronto

Le chiamate descritte in precedenza sono **chiamate bloccanti** (**blocking calls**), talvolta chiamate anche **chiamate sincrone** (**synchronous calls**). Quando un processo chiama una *send*, specifica una destinazione e un buffer da spedire a quella destinazione. Durante la spedizione del messaggio, il processo che invia è bloccato (cioè, sospeso). L'istruzione che segue la chiamata a *send* non è eseguita finché il messaggio non è stato spedito completamente, come mostrato nella Figura 8.19(a). Allo stesso modo, una chiamata a una *receive* non restituisce il controllo finché il messaggio non è stato effettivamente ricevuto e messo nel buffer dei messaggi puntato dal parametro. Il processo resterebbe sospeso in *receive* fino all'arrivo del messaggio, anche se impiegasse ore. In alcuni sistemi il ricevente può specificare da chi vorrebbe ricevere il messaggio, nel cui caso resta bloccato finché non lo riceve dal destinatario specificato.

Un'alternativa alle chiamate bloccanti è l'utilizzo delle **chiamate non bloccanti** (**nonblocking calls**), altrimenti dette **chiamate asincrone** (**asynchronous calls**). Se la *send* non è bloccante, prima dell'invio del messaggio restituisce immediatamente il controllo al chiamante. Il vantaggio di questa progettazione è che il processo di spedizione può continuare l'elaborazione in parallelo con la trasmissione del messaggio, invece di lasciare la CPU inattiva (sempre che non vi sia alcun altro processo eseguibile). La scelta fra le primitive bloccanti e quelle non bloccanti è normalmente presa dai progettisti del sistema (quindi o si utilizzano le une o le altre), sebbene in qualche sistema siano disponibili entrambe e l'utente possa scegliere quella che preferisce.

Il vantaggio prestazionale offerto dalle primitive non bloccanti è tuttavia controbilanciato da un serio svantaggio: il mittente non può modificare il buffer del messaggio finché il messaggio non è stato spedito. Le conseguenze dovute alla possibilità che il processo possa

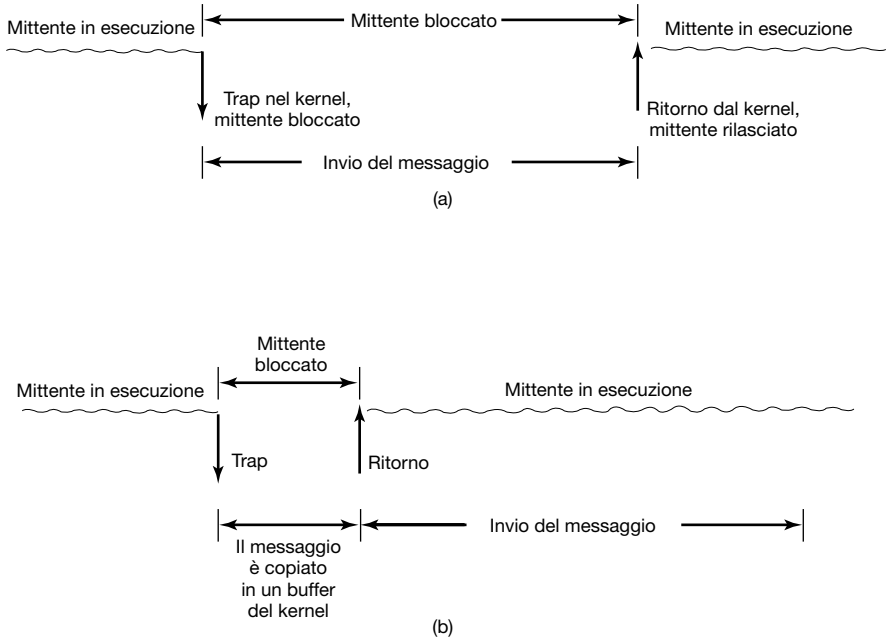

Figura 8.19 (a) Una chiamata send bloccante. (b) Una chiamata send non bloccante.

modificare il messaggio durante la trasmissione sono troppo gravi da poter essere acccettabili. Ancor peggio, il processo mittente non ha idea di quando la trasmissione sarà terminata, quindi non sa quando sarà sicuro riusare il buffer. E difficilmente potrà evitare di non usarlo mai più.

Le possibilità di uscirne sono tre. La prima soluzione è far sì che il kernel copi il messaggio in un buffer interno e poi permetta al processo di continuare, come mostrato nella Figura 8.19(b). Dal punto di vista del mittente, questo schema è il medesimo di una chiamata bloccante: appena restituito il controllo, è libero di riusare il buffer. Naturalmente il messaggio non è stato ancora spedito, ma il mittente non è intralciato da questo fatto. Lo svantaggio di questo metodo è che ogni messaggio in uscita dev'essere copiato dallo spazio utente allo spazio del kernel. Con molte interfacce di rete, il messaggio dovrà essere copiato comunque in seguito a un buffer di trasmissioni hardware, per cui la prima copia è essenzialmente sprecata. La copia aggiuntiva può ridurre considerevolmente le prestazioni del sistema.

La seconda soluzione è di inviare un interrupt al mittente quando il messaggio è stato completamente inviato per informare che il buffer è ancora una volta disponibile. In questo caso non è richiesta alcuna copia, il che risparmia tempo, ma gli interrupt a livello utente rendono la programmazione pesante, difficile e soggetta alle condizioni competitive, rendendo impossibile riprodurli e farne il debug.

La terza soluzione è fare la copia del buffer sulla scrittura, cioè contrassegnarlo come read-only finché il messaggio non sia stato inviato. Se il buffer è riutilizzato prima che il messaggio sia stato spedito, ne viene fatta una copia. Il problema di questa soluzione è che,

a meno che il buffer sia isolato nella sua pagina, anche le scritture a quasi tutte le variabili forzeranno una copia. Inoltre serve un'amministrazione ulteriore, perché adesso l'atto di spedire un messaggio implicitamente influisce sullo stato di lettura/scrittura della pagina. Alla fine, prima o poi, la pagina probabilmente sarà scritta ancora, innescando una copia che non sarebbe necessaria.

Lato mittente, queste sono quindi le possibilità.

1. Spedizione bloccante (con la CPU inattiva durante la trasmissione).
2. Spedizione non bloccante con copia (con del tempo della CPU sprecato per la copia).
3. Spedizione non bloccante con interrupt (con difficoltà di programmazione).
4. Copia su scrittura (con copia ulteriore plausibilmente alla fine necessaria).

In condizioni normali la prima scelta è la migliore, specialmente con molteplici thread a disposizione, nel cui caso mentre un thread è bloccato nello spedire, gli altri possono continuare a lavorare. Per essere gestita essa non richiede inoltre alcun buffer del kernel. Inoltre, come si vede dal confronto fra la Figura 8.19(a) e la Figura 8.19(b), il messaggio esce di solito più velocemente se non serve la copia.

Per completezza, dobbiamo puntualizzare che alcuni autori usano un criterio diverso per distinguere fra primitive sincrone e asincrone. Da un punto vista alternativo, una chiamata è sincrona se il mittente è bloccato finché il messaggio è stato ricevuto ed è stata restituita una conferma (Andrews, 1991). Nel mondo delle comunicazioni real-time, sincrono ha un altro significato, che sfortunatamente può creare confusione.

Così come una *send* può essere bloccante o non bloccante, così accade anche per una *receive*. Una chiamata bloccante sospende il chiamante finché il messaggio non è arrivato. Se sono disponibili più thread, questo è un approccio semplice. In alternativa, una *receive* non bloccante indica semplicemente al kernel dove è il buffer e restituisce il controllo quasi immediatamente. Per segnalare l'arrivo del messaggio si può usare un interrupt. Tuttavia gli interrupt sono difficili da programmare e anche abbastanza lenti, così sarebbe preferibile per il destinatario controllare i messaggi in entrata usando una procedura, *poll*, che avvisa se vi è qualche messaggio in attesa. Se è così, il chiamante può richiamare *get_message*, che restituisce il primo messaggio in arrivo. In alcuni sistemi il compilatore può inserire nel codice delle chiamate *poll* in punti appropriati, sebbene sia complesso sapere quanto spesso fare *polling*.

Un'altra opzione è uno schema in cui l'arrivo di un messaggio causa la creazione spontanea di un nuovo thread nello spazio degli indirizzi del processo ricevente. Questo thread è chiamato **thread pop-up**. Esegue una procedura specificata in anticipo e il cui parametro è un puntatore al messaggio entrante. Dopo aver processato il messaggio, semplicemente esce e viene automaticamente distrutto.

Una variante di questa idea è il fatto di eseguire il codice del ricevente direttamente nel gestore dell'interrupt, senza avere il problema della creazione del thread pop-up. Per velocizzare ulteriormente questo schema, il messaggio stesso contiene l'indirizzo del gestore, così quando arriva un messaggio, il gestore può essere richiamato in poche istruzioni. In questo caso il grande vantaggio è che non è necessaria alcuna copia. Il gestore prende il messaggio dalla scheda d'interfaccia e lo processa direttamente. Questo schema è chiamato **messaggi attivi** (**active messages** – Von Eicken et al., 1992). Poiché ciascun messaggio contiene

l'indirizzo del gestore, i messaggi attivi lavorano solo quando chi invia e chi riceve si conoscono l'un l'altro reciprocamente.

8.2.4 Chiamata di procedura remota

Sebbene il modello a scambio di messaggi fornisca un sistema conveniente per strutturare un sistema operativo per multicomputer, soffre di una pecca fondamentale: il paradigma base attorno al quale sono costruite tutte le comunicazioni è l'input/output. Le procedure *send* e *receive* sono impegnate nel fare I/O; molte persone credono che l'I/O sia un modello di programmazione errato.

Questo problema è conosciuto da molto tempo, ma è stato fatto poco, fino a che una pubblicazione di Birrell e Nelson (1984) ha introdotto un modo completamente diverso di affrontare il problema. Sebbene l'idea sia sorprendentemente semplice (dopo averci pensato), le implicazioni sono spesso sottili. In questo paragrafo ne esamineremo il concetto, le sue implicazioni, i suoi punti di forza e di debolezza.

In parole povere, quanto suggerito da Birrell e Nelson fu di consentire ai programmi di chiamare procedure che si trovavano su altre CPU. Quando un processo sulla macchina 1 chiamava una procedura sulla macchina 2, il processo chiamante sulla macchina 1 era sospeso e l'esecuzione della procedura chiamata aveva luogo sulla macchina 2. Le informazioni potevano essere trasportate dal chiamante al chiamato nei parametri e potevano tornare indietro come risultato della procedura. Nessun passaggio di messaggi o I/O era visibile al programmatore.

Questa tecnica è conosciuta come **RPC** (**remote procedure call – chiamata di procedura remota**) ed è alla base di gran parte del software per multicomputer. Tradizionalmente la procedura chiamante è conosciuta come client e quella chiamata come server e anche noi useremo questa terminologia.

L'idea alla base delle RPC è che inoltrare una chiamata di procedura remota è altrettanto possibile che una chiamata locale. Nella forma più semplice, per chiamare una procedura remota, il programma del client dev'essere legato a una piccola procedura di libreria chiamata **client stub**, che rappresenta la procedura del server nello spazio degli indirizzi del client. Analogamente, il server è legato a una procedura chiamata **server stub**. Queste procedure nascondono il fatto che la chiamata di procedura dal client al server non è locale.

I passaggi effettivi per fare una RPC sono illustrati nella Figura 8.20. Nel passo 1 il client chiama il *client stub*. Questa chiamata è una chiamata di procedura locale, con i parametri inseriti nello stack nel solito modo. Nel passo 2 il *client stub* confeziona i parametri in un messaggio ed esegue una chiamata di sistema per spedirlo. Il confezionamento di questi parametri è detto **marshaling**. Nel passo 3 il kernel spedisce il messaggio dalla macchina client alla macchina server. Nel passo 4 il kernel passa il pacchetto in ingresso al server stub (che normalmente dovrebbe aver in precedenza chiamato *receive*). Alla fine, nel passo 5 il server stub chiama la procedura del server. La risposta ripercorre lo stesso cammino in direzione opposta.

In questo caso l'elemento chiave da notare è che la procedura del client, scritta dall'utente, esegue solo una normale (cioè locale) chiamata di procedura al client stub, che ha lo stesso nome della procedura del server. Poiché la procedura del client e il client stub sono nel medesimo spazio degli indirizzi, i parametri sono trasmessi nel modo solito. Allo stesso modo, la procedura del server è chiamata tramite una procedura nel suo spazio degli indirizzi

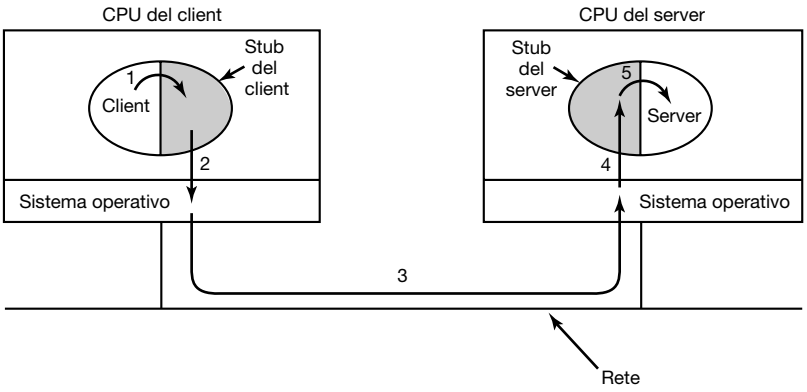

Figura 8.20 Passaggi nella preparazione di una chiamata di procedura remota. Gli stub sono evidenziati in grigio.

con i parametri che si aspetta. Per la procedura del server, non vi è nulla di anomalo. In questo modo, invece di fare l'I/O usando *send* e *receive*, la comunicazione remota è fatta simulando una normale chiamata di procedura.

Questioni di implementazione

Malgrado la loro eleganza concettuale, le RPC presentano alcuni trabocchetti. Uno importante è l'uso dei puntatori come parametri. Normalmente, il passaggio di un puntatore a una procedura non è un problema. La procedura chiamata può usare il puntatore allo stesso modo del chiamante, dato che le due procedure risiedono nel medesimo spazio degli indirizzi virtuali. Con le RPC il passaggio dei puntatori è impossibile, dato che il client e il server sono in spazi di indirizzi diversi.

In alcuni casi, possono essere usati dei trucchi per rendere possibile il passaggio dei puntatori. Supponete che il primo parametro sia un puntatore a un intero, *k*. Il *client stub* può preparare *k* e inoltrarlo al server. Il *server stub* crea poi un puntatore a *k*, lo passa alla procedura del server, come previsto. Quando la procedura del server restituisce il controllo al server stub, quest'ultimo rimanda *k* indietro al client, dove il nuovo *k* è copiato sul vecchio, nel caso il server lo avesse cambiato. In effetti la sequenza di chiamata standard di "chiamata per riferimento" è stata sostituita da "copia-ripristina". Sfortunatamente questo trucco, per esempio, non sempre funziona qualora il puntatore sia relativo a un grafico o ad un'altra struttura dati complessa. È per questa ragione che devono essere messe alcune restrizioni sui parametri delle procedure chiamate remotamente.

Un secondo problema è che nei linguaggi detti "debolmente tipizzati", come il C, è perfettamente logico scrivere una procedura che elabori il prodotto interno di due vettori (array), senza specificare quanto siano grandi l'uno o l'altro. Ciascuno potrebbe essere terminato da un valore speciale conosciuto solo dalla procedura chiamante e chiamata. A queste condizioni è praticamente impossibile per il client stub preparare i parametri: non ha possibilità di determinare quanto siano grandi.

Un terzo problema è che non sempre è possibile dedurre i tipi dei parametri, nemmeno da una normale specifica o dal codice stesso. Un esempio è *printf*, che può avere qualunque numero di parametri (almeno uno) e che può essere un misto di interi, numeri in singola precisione o in doppia precisione, caratteri, stringhe, numeri a virgola mobile di svariata lunghezza o altri tipi. Dato che il C è troppo permissivo, provare a chiamare *printf* come una procedura remota sarebbe praticamente impossibile. Tuttavia, una regola gradita che indica che le RPC possono essere usate a condizione di non programmare in C (o C++) sarebbe troppo restrittiva.

Un quarto problema è relativo all'uso delle variabili globali. Normalmente, le procedure chiamanti e chiamate possono comunicare usando variabili globali, oltre a comunicare tramite i parametri. Se ora la procedura chiamata è spostata su una macchina remota, il codice genererà un errore, dato che le variabili non sono più condivise.

Questi problemi non vogliono suggerire che le RPC siano senza speranza. Sono usate infatti abbastanza diffusamente, ma sono necessarie alcune restrizioni e attenzioni affinché all'atto pratico funzionino correttamente.

8.2.5 Memoria condivisa distribuita

Sebbene le RPC abbiano il loro fascino, molti programmatori continuano a preferire un modello di memoria condivisa e gli piacerebbe utilizzarlo, anche su un multicomputer. Stranamente, è possibile preservare l'illusione della memoria condivisa ragionevolmente bene, anche quando non esiste, usando una tecnica chiamata **DSM** (**distributed shared memory**) (Li, 1986 e Li e Hudak, 1989). Con la DSM, ciascuna pagina è posizionata in una delle memorie della Figura 8.1. Ciascuna macchina ha la sua memoria virtuale e le sue personali tabelle delle pagine. Quando una CPU effettua una LOAD o una STORE su una pagina che non ha, accade una *trap* nel sistema operativo. Il sistema operativo localizza poi la pagina e chiede alla CPU che la detiene attualmente di eseguirne l'*unmap* e di spedirla lungo la rete d'interconnessione. Quando arriva, la pagina viene mappata e l'istruzione che ha generato l'errore riavviata. In effetti, il sistema operativo semplicemente gestisce degli errori di pagina remotamente invece che dal disco locale. Per l'utente, la macchina è come se avesse la memoria condivisa.

La differenza fra la memoria condivisa reale e la DSM è illustrata nella Figura 8.21. Nella Figura 8.21(a) vediamo un vero multiprocessore con memoria condivisa fisica implementato tramite l'hardware. Nella Figura 8.21(b) vediamo la DSM, implementata tramite il sistema operativo. Nella Figura 8.21(c) vediamo un'altra forma ancora di memoria condivisa, implementata da livelli ulteriormente superiori del software. Ritorneremo più avanti nel capitolo su questa terza opzione, ma al momento ci concentreremo sulla DSM.

Analizziamo adesso in dettaglio come funziona la DSM. In un sistema DSM, lo spazio degli indirizzi è suddiviso in pagine, con le pagine distribuite su tutti i nodi del sistema. Quando la CPU referenzia un indirizzo che non è locale, avviene una *trap* e il software della DSM preleva la pagina contenente l'indirizzo e riavvia l'istruzione che ha generato l'errore, che ora si completa con successo. Questo concetto è illustrato nella Figura 8.22(a) per uno spazio degli indirizzi con 16 pagine e quattro nodi, ognuno capace di contenere sei pagine.

In questo esempio, se la CPU 0 referenzia delle istruzioni o dei dati nelle pagine 0, 2, 5 o 9, i riferimenti sono gestiti localmente. I riferimenti ad altre pagine causano delle *trap* Per esempio, un riferimento a un indirizzo nella pagina 10 provoca una *trap* nel software della DSM, che poi muove la pagina 10 dal nodo 1 al nodo 0, come mostrato nella Figura 8.22(b).

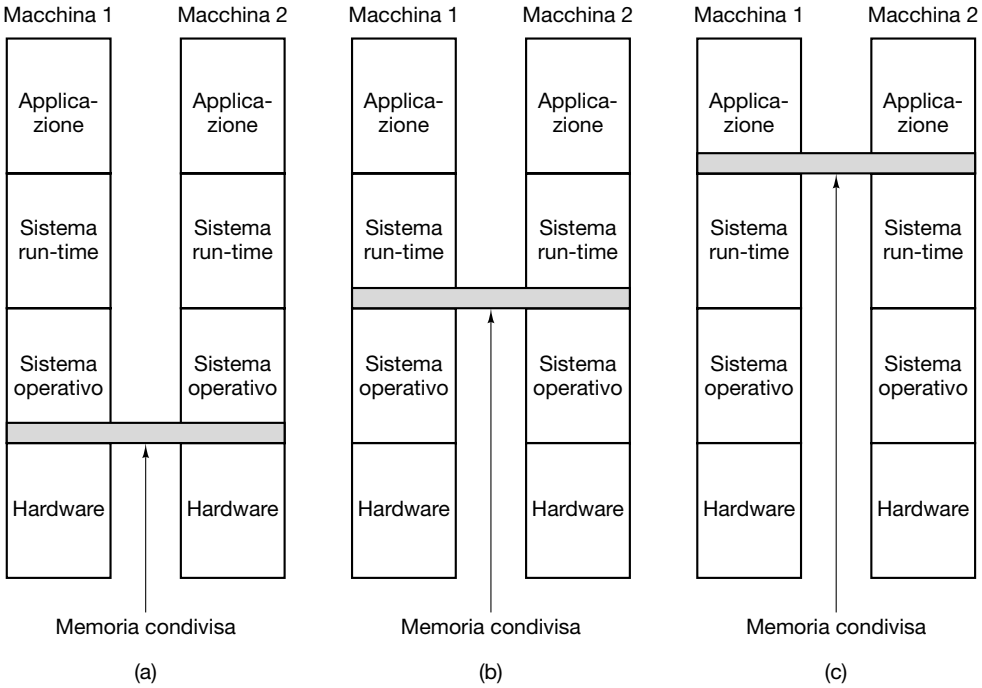

Figura 8.21 Vari livelli a cui si può implementare la memoria condivisa. (a) L'hardware. (b) Il sistema operativo. (c) Il software a livello utente.

Replica

Un miglioramento al sistema di base che può portare a considerevoli salti di qualità in termini di prestazioni è la replica delle pagine che sono di sola lettura, per esempio il testo dei programmi, le costanti di sola lettura o altre strutture dati di sola lettura. Per esempio, se la pagina 10 nella Figura 8.22 è un paragrafo del testo del programma, il suo utilizzo da parte della CPU 0 può comportare il fatto che una sua copia venga inviata alla CPU 0 senza che l'originale nella memoria della CPU 1 venga intaccato, come illustrato nella Figura 8.22(c). In questo modo le CPU 0 e 1 possono entrambe referenziare la pagina 10 ogni qualvolta sia necessario senza provocare trap per prelevare memoria mancante. Un'altra possibilità è quella di replicare non solo le pagine di sola lettura, ma anche tutte le pagine. Finché si fanno delle letture, non c'è effettivamente diversità nel replicare pagine in sola lettura o pagine in lettura-scrittura. Tuttavia, se una pagina replicata è improvvisamente modificata, dev'essere intrapresa un'azione speciale che eviti il fatto di avere molteplici copie inconsistenti. Come prevenire l'inconsistenza si vedrà nei paragrafi successivi.

Falsa condivisione

I sistemi DSM sono simili ai multiprocessori in alcuni punti chiave. In entrambi i sistemi, quando si fa riferimento a una parola di memoria non locale, un pezzo di memoria conte-

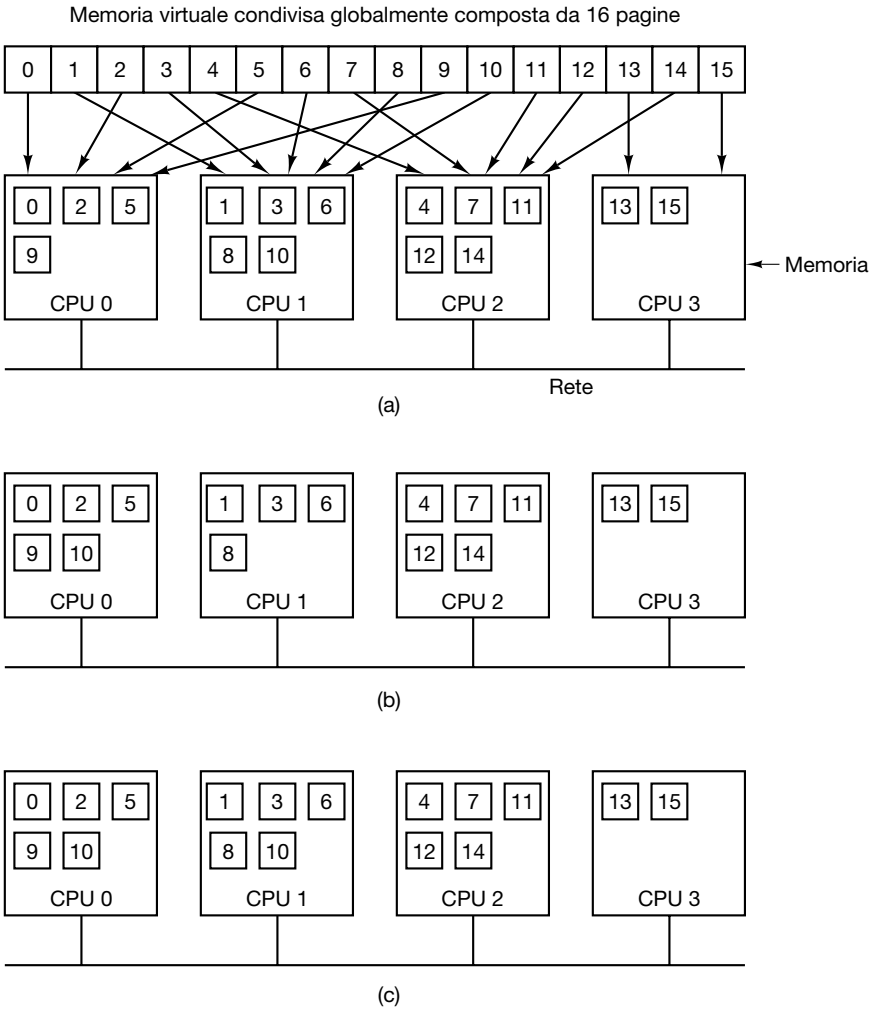

Figura 8.22 (a) Pagine dello spazio degli indirizzi distribuite fra quattro macchine. (b) Situazione dopo che la CPU 1 referenzia la pagina 10 e che la pagina è spostata lì. (c) Situazione se la pagina è in sola lettura e si utilizza la replica.

nente la parola è prelevato dalla sua posizione attuale e messo sulla macchina che esegue il riferimento (nella memoria o nella cache, rispettivamente). Un'importante questione in fase di progettazione è: quanto dev'essere grande questo pezzo? Nei multiprocessori la dimensione del blocco della cache è solitamente di 32 o 64 byte, per evitare di stressare il bus con trasferimenti troppo lunghi. Nei sistemi DMS l'unità dev'essere un multiplo della dimensione della pagina (poiché l'MMU funziona con le pagine), ma può essere di 1, 2, 4 o più pagine. In effetti, facendo così si simula una dimensione di pagina maggiore.

Avere una dimensione di pagina maggiore per la DSM ha i suoi pro e i suoi contro. Il vantaggio più grande è che, poiché il tempo di avvio per un trasferimento di rete è abbastanza lungo, non impiega in realtà molto di più a trasferire 4096 byte piuttosto che 1024 byte. Trasferendo i dati in grandi unità, quando dev'essere spostato un pezzo grande di uno spazio degli indirizzi, il numero di trasferimenti può essere spesso ridotto. Questa caratteristica è particolarmente importante perché molti programmi fanno riferimenti localizzati, che significa che se un programma ha referenziato una parola in una pagina, probabilmente referenzierà nel futuro immediato altre parole nella medesima pagina.

Per contro, la rete sarebbe occupata più a lungo con un trasferimento più grande, bloccando altri errori provocati da altri processi. Inoltre, una dimensione di pagina effettiva troppo grande introduce un nuovo problema, chiamato **falsa condivisione** (**false sharing**), illustrato nella Figura 8.23. In questa situazione abbiamo un pagina contenente due variabili condivise non relazionate, *A* e *B.* Il processore 1 fa un uso pesante di *A*, leggendola e scrivendola. Il processo 2 allo stesso modo usa *B* frequentemente. In questa situazione la pagina contenente entrambe le variabili sarà costantemente rimpallata avanti e indietro fra le due macchine.

In questo caso il problema è che, sebbene le variabili non siano relazionate, per caso sono sulla stessa pagina, per cui quando un processo ne usa una, preleva anche l'altra. Più grande è la dimensione effettiva della pagina, più spesso capita la falsa condivisione; viceversa più piccola è la dimensione della pagina, meno frequentemente accade. Nei normali sistemi a memoria virtuale non vi è nulla di analogo a questo fenomeno.

I compilatori intelligenti che capiscono il problema e mettono le variabili nello spazio degli indirizzi di conseguenza possono aiutare a ridurre la falsa condivisione e a migliorare le prestazioni. L'operazione è semplice a dirsi ma non a farsi. Inoltre, se la falsa condivisione è dovuta al fatto che il nodo 1 usa un elemento di un array e il nodo 2 usa un diverso elemento dello stesso array, anche un compilatore intelligente può fare ben poco per eliminare il problema.

Ottenere la consistenza sequenziale

Se pagine scrivibili non sono replicate, ottenere la consistenza non è un problema. C'è esattamente una sola copia di ciascuna pagina scrivibile ed è spostata avanti e indietro dinami-

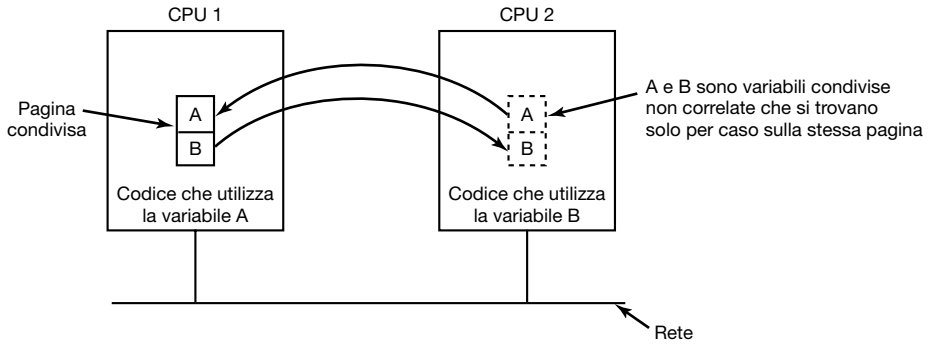

Figura 8.23 Falsa condivisione di una pagina contenente due variabili non correlate.

camente, a seconda delle necessità. Dato che non sempre è possibile sapere in anticipo quali pagine siano scrivibili, in molti sistemi DSM, quando un processo tenta di leggere una pagina remota, si effettua una copia in locale: sia le copie locali sia quelle remote sono impostate nelle rispettive MMU come read-only. Finché tutti i riferimenti sono letture, è tutto a posto.

Se tuttavia qualche processo provasse a scrivere su una pagina replicata, sorgerebbe un potenziale problema di consistenza, poiché cambiare una copia e lasciare sole le altre non è accettabile. Questa situazione è analoga a ciò che accade in un multiprocessore quando una CPU prova a modificare una parola presente in più cache. In questo caso la soluzione è che la CPU che sta per fare la scrittura per prima cosa metta un segnale sul bus che indichi a tutte le altre CPU di scaricare la loro copia del blocco della cache. I sistemi DSM tipicamente funzionano allo stesso modo. Prima che una pagina condivisa possa essere scritta, è inviato un messaggio a tutte le altre CPU che tengono una copia della pagina, avvisandole di eseguirne l'unmap e di scaricarla. Dopo che tutte hanno avvisato che l'operazione di unmap è terminata, la CPU originale può fare la scrittura.

In determinate circostanze ben identificate è anche possibile tollerare più di una copia di pagine scrivibili. Un modo è consentire a un processo di acquisire un *lock* su una parte di uno spazio degli indirizzi virtuali e poi eseguire molteplici operazioni di lettura e scrittura sulla memoria in stato *locked*. Nel momento in cui il *lock* è rilasciato, i cambiamenti possono essere propagati alle altre copie. Finché solo una CPU può fare il *lock* di una pagina a un determinato momento, questo schema preserva la consistenza.

In alternativa, quando una pagina potenzialmente scrivibile è effettivamente scritta per la prima volta, è effettuata una copia pulita e salvata sulla CPU che fa la scrittura. I *lock* sulla pagina possono poi essere acquisiti, la pagina aggiornata e i *lock* rilasciati. In seguito, quando un processo su una macchina remota prova ad acquisire un *lock* su quella pagina, la CPU che l'ha scritta prima confronta lo stato attuale della pagina con la copia pulita e costruisce un messaggio che elenca tutte le parole cambiate. L'elenco è poi spedito alla CPU che la acquisisce per aggiornare la sua copia, invece che invalidarla (Keleher et al., 1994).

8.2.6 Scheduling nei multicomputer

Su un multiprocessore tutti i processi risiedono nella stessa memoria. Quando una CPU termina la sua attività attuale, preleva un processo e lo esegue. In linea di principio, tutti i processi sono potenzialmente candidati all'esecuzione. Nel caso di un multicomputer la situazione è abbastanza differente. Ciascun nodo ha la sua personale memoria e il suo personale insieme di processi. La CPU 1 non può improvvisamente decidere di eseguire un processo situato sul nodo 4 senza prima svolgere una discreta quantità di lavoro per prenderlo. Questa differenza significa che lo scheduling nei multicomputer è più semplice, ma l'allocazione dei processi ai nodi è più importante.

Lo scheduling nei multicomputer è in qualche modo simile allo scheduling nei multiprocessori, ma non tutti gli algoritmi del primo sono applicabili a quest'ultimo. L'algoritmo più semplice dei multiprocessori – tenere un elenco centrale unico di processi pronti – comunque non funziona, dato che ogni processo può essere eseguito solo sulla CPU su cui è correntemente posizionato. Tuttavia, quando è creato un nuovo processo, dev'essere fatta una scelta su dove situarlo, per esempio per bilanciare il carico di lavoro.

Dato che ciascun nodo ha i suoi processi, può essere usato qualsiasi algoritmo di scheduling locale. È tuttavia anche possibile usare il *gang scheduling* dei multiprocessori, dato

che richiede semplicemente un accordo iniziale su quale processo eseguire in quale intervallo di tempo e un qualche metodo per coordinare la partenza degli intervalli di tempo.

8.2.7 Bilanciamento del carico

C'è relativamente poco da dire sullo scheduling nei multicomputer perché, una volta che un processo è stato assegnato a un nodo, funzionerà qualsiasi algoritmo di scheduling locale, a meno che non sia usato il *gang scheduling*. Tuttavia proprio perché c'è così poco controllo una volta che un processo è stato assegnato a un nodo, la decisione di quale processo dovrebbe andare su quale nodo è decisamente importante, diversamente dai sistemi di multiprocessori, in cui tutti i processi convivono nella medesima memoria e possono essere schedulati a piacere su qualsiasi CPU. Di conseguenza vale la pena analizzare come i processi possono essere assegnati in modo efficace. Gli algoritmi e le euristiche per questa assegnazione sono conosciuti come **algoritmi di allocazione dei processori**.

Nel corso degli anni è stato proposto un gran numero di algoritmi di allocazione dei processori (cioè dei nodi). Differiscono in ciò che presuppongono si sappia e nell'obiettivo.

Le proprietà che dovrebbero essere conosciute riguardo a un processo includono i requisiti della CPU, l'utilizzo della memoria e la quantità di comunicazione con tutti gli altri processi. Gli obiettivi possibili sono la minimizzazione dei cicli di CPU sprecati dovuti alla mancanza di lavoro locale, la minimizzazione della banda delle comunicazioni locali e l'assicurare l'imparzialità a utenti e processi.

Un algoritmo deterministico basato sulla teoria dei grafi

Una classe di algoritmi ampiamente studiata è indirizzata ai sistemi i cui processi hanno requisiti di CPU e di memoria conosciuti e per cui si conosca la matrice che fornisce la quantità media di traffico fra ciascuna coppia di processi. Se il numero dei processi è maggiore del numero delle CPU, k, a ciascuna CPU devono essere assegnati molti processi. L'idea è di effettuare questo assegnamento cercando di minimizzare il traffico della rete.

Il sistema può essere rappresentato come un grafo pesato, con ciascun vertice che rappresenta un processo e ciascun arco che rappresenta il flusso dei messaggi fra due processi. Dal punto di vista matematico, il problema si riduce nel trovare un modo per partizionare (cioè tagliare) il grafico in k sottografici disgiunti, soggetti a determinati vincoli (per esempio i requisiti totali di CPU e di memoria sotto certi limiti per ciascun sottografico). Per ciascuna soluzione che rispetti i vincoli, gli archi che stanno del tutto all'interno di un singolo sottografico rappresentano le comunicazioni all'interno della stessa macchina e possono essere ignorati. Gli archi che vanno da un sottografico a un altro rappresentano il traffico di rete. L'obiettivo è quindi quello di trovare la suddivisione che minimizzi il traffico di rete rispettando nel contempo i vincoli. Come esempio, la Figura 8.24 mostra un sistema di nove processi, da A a I, con ciascun arco etichettato con il carico medio di comunicazione fra quei due processi (in Mbps).

Nella Figura 8.24(a) abbiamo partizionato il grafico con i processi A, E e G sul nodo 1, i processi B, F e H sul nodo 2 e i processi C, D e I sul nodo 3. Il traffico di rete totale è dato dalla somma degli archi intersecati dai tagli (le linee tratteggiate), cioè 30 unità. Nella Figura 8.24(b) abbiamo un differente partizionamento che ha solo 28 unità di traffico di rete. Supponendo che rispetti tutti i vincoli di memoria e CPU, questa è la soluzione migliore poiché richiede meno comunicazione.

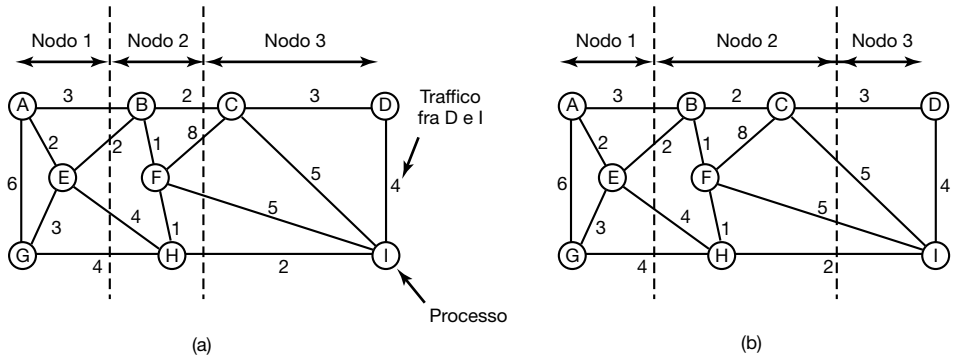

Figura 8.24 Due modi di assegnare nove processi a tre nodi.

Intuitivamente, quello che stiamo facendo è cercare dei cluster che siano fortemente accoppiati (con un alto flusso di traffico all'interno del cluster) ma che interagiscano poco con gli altri cluster (con un basso flusso di traffico fra cluster). Alcune delle prime pubblicazioni a questo riguardo sono Chow e Abraham (1982); Lo, (1984); Stone e Bokhari (1978).

Un algoritmo euristico distribuito avviato dal mittente

Analizziamo adesso alcuni algoritmi distribuiti. Un algoritmo indica che quando è creato un processo, questi è eseguito sul nodo che l'ha creato a meno che quel nodo non sia sovraccarico. La valutazione del sovraccarico potrebbe riguardare i troppi processi, un working set totale troppo grande o qualche altra considerazione. Qualora sia sovraccarico, il nodo seleziona un altro nodo a caso e ne chiede lo stato di carico di lavoro (usando gli stessi parametri di valutazione). Se il nodo esaminato è al di sotto del limite, gli viene spedito il processo (Eager et al., 1986). In caso contrario è scelta un'altra macchina da esaminare. La valutazione non prosegue all'infinito. Se non è rintracciato un nodo nel giro di N valutazioni, l'algoritmo termina e il processo è eseguito sulla macchina che lo ha originato. L'idea è, per i nodi con un carico di lavoro pesante, di liberarsi del lavoro in eccesso, come illustrato nella Figura 8.25(a), che descrive il bilanciamento del carico avviato dal mittente del processo.

Eager et al. (1986) costruirono un modello analitico a code di questo algoritmo. Usando questo modello si stabilì che l'algoritmo ha un buon comportamento ed è stabile all'interno di un ampio campo di parametri, inclusi diversi valori limite, costi di trasferimento e limiti di esame.

Un algoritmo euristico distribuito avviato dal destinatario

Un algoritmo complementare a quello appena indicato, avviato da un mittente sovraccarico, è quello avviato da un destinatario in situazione di sottocarico, come mostrato nella Figura 8.25(b). Con questo algoritmo, ogni volta che finisce un processo, il sistema verifica se ha lavoro abbastanza. In caso negativo, sceglie una macchina a caso e le chiede del lavoro.

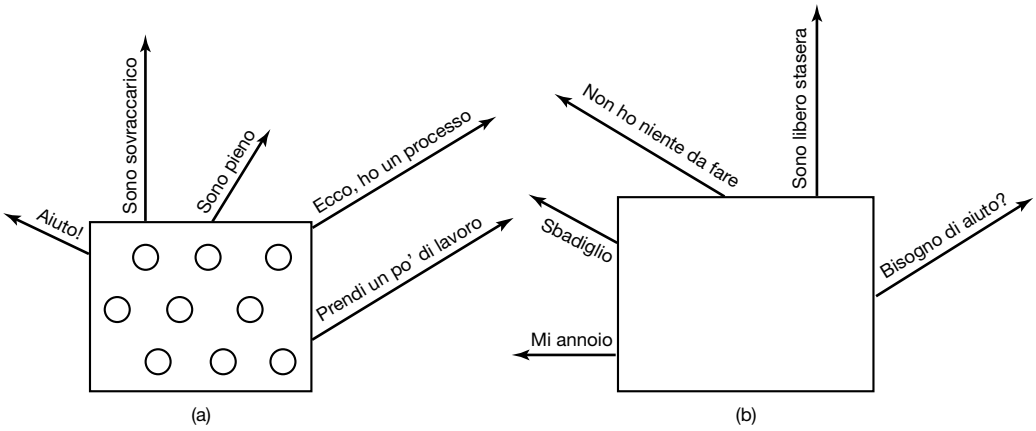

Figura 8.25 (a) Un nodo sovraccarico che cerca un nodo meno carico a cui passare i processi. (b) Un nodo vuoto che cerca lavoro da svolgere.

8.3 Sistemi distribuiti

Dopo aver completato il nostro studio su multiprocessori, multicomputer e macchine virtuali, è il momento di rivolgere la nostra attenzione all'ultima tipologia di sistemi multiprocessore, i **sistemi distribuiti**. Questi sistemi sono simili ai multicomputer, nel senso che ogni nodo ha la sua memoria privata, senza memoria fisica condivisa nel sistema. Tuttavia, i sistemi distribuiti sono ancora più debolmente strutturati dei multicomputer.

Innanzitutto, i nodi di un multicomputer hanno generalmente una CPU, della RAM, un'interfaccia di rete e forse un hard disk per la paginazione. Al contrario, ogni nodo in un sistema distribuito è un computer completo, con un buon numero di periferiche. Inoltre, i nodi di un multicomputer si trovano normalmente in un unico ambiente, così sono in grado di comunicare tramite una rete dedicata ad alta velocità, mentre i nodi di un sistema distribuito possono essere distribuiti in tutto il mondo. Infine, tutti i nodi di un multicomputer eseguono lo stesso sistema operativo, condividono un unico file system e sono gestiti dal medesimo amministratore, mentre i nodi di un sistema distribuito possono ciascuno funzionare con un diverso sistema operativo, ognuno dei quali ha il suo personale file system ed è gestito da un differente amministratore. Un esempio tipico di multicomputer è a 512 nodi, in una singola stanza di un'azienda o università e che lavora, per esempio, su modelli farmaceutici, mentre un tipico sistema distribuito consiste di migliaia di macchine debolmente cooperanti su Internet. Nella Figura 8.26 si paragonano multiprocessori, multicomputer e sistemi distribuiti sui punti sopra menzionati.

Se utilizziamo queste metriche, i multicomputer sono chiaramente nel mezzo. Una domanda interessante è: "I multicomputer sono più simili ai multiprocessori o ai sistemi distribuiti?" La risposta dipende fortemente dalle vostre prospettive. Da un punto di vista tecnico, i multiprocessori hanno una memoria condivisa e gli altri due no. Questa differenza genera diversi modelli di programmazione e un atteggiamento mentale differente. Tuttavia,

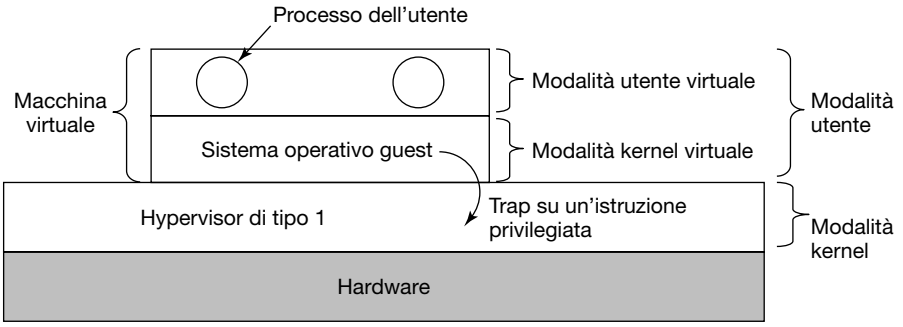

Figura 8.26 Confronto tra tre tipologie di sistemi con più di una CPU.

per quanto riguarda la prospettiva delle applicazioni, i multiprocessori e i multicomputer sono solo grandi scaffalature di attrezzature in una sala macchine. Entrambi sono utilizzati per risolvere problemi intensivi dal punto di vista elaborativo, mentre un sistema distribuito che collega computer su Internet è in genere maggiormente coinvolto in comunicazioni più che elaborazioni e si utilizza in modo diverso.

Per certi versi, la debole struttura dei computer in un sistema distribuito è sia un punto di forza che di debolezza. È un punto di forza perché i computer sono utilizzabili per una grande varietà di applicazioni, ma è anche un punto di debolezza, perché la programmazione di tali applicazioni è difficoltosa per la mancanza di un modello sottostante comune.

Le applicazioni tipiche di Internet comprendono l'accesso a computer remoti (utilizzando *telnet*, *ssh* e *rlogin*), l'accesso a informazioni remote (utilizzando il World Wide Web e l'FTP, il File Transfer Protocol), la comunicazione persona-a-persona (utilizzando programmi di e-mail e chat) e molte altre applicazioni emergenti (come il commercio elettronico, la telemedicina e i corsi on-line a distanza). Il problema di tutte queste applicazioni è che ognuna deve ricominciare da capo. Per esempio, e-mail, FTP e World Wide Web fondamentalmente spostano file da un punto *A* a un punto *B*, ma ognuno ha il suo modo di farlo, completo delle sue regole di naming, protocolli di trasferimento, tecniche di replica e tutto il resto. Sebbene molti browser web nascondano queste differenze all'utente medio, i meccanismi basilari sono completamente differenti. Nasconderli a livello d'interfaccia utente è come ordinare un viaggio da New York a San Francisco presso un sito web full-service di un'agenzia di viaggi e scoprire solo alla fine se si è acquistato un biglietto aereo, ferroviario o per un autobus.

Quello che i sistemi distribuiti aggiungono alla rete sottostante è un paradigma comune (modello) che fornisce un modo uniforme di guardare all'intero sistema. Lo scopo dei sistemi distribuiti è quello di trasformare un gruppo di macchine connesse debolmente in un sistema coerente basato su un unico concetto. A volte il paradigma è semplice e a volte è più elaborato, ma l'idea è sempre quella di fornire qualcosa che unifichi il sistema.

Un esempio semplice di paradigma unificante in un contesto leggermente diverso si ritrova in UNIX, dove tutti i dispositivi di I/O vengono fatti assomigliare a dei file. Disporre di tastiere, stampanti e linee seriali che operano tutte nello stesso modo, con le stesse primitive, ne semplifica l'impiego rispetto all'averle tutte concettualmente differenti.

Un modo in cui un sistema distribuito può raggiungere una certa misura di uniformità a fronte di hardware e sistemi operativi sottostanti differenti è avere un livello di software al di sopra del sistema operativo. Questo livello, chiamato **middleware**, è illustrato nella Figura 8.27 e fornisce determinate strutture di dati e operazioni che permettono ai processi e agli utenti di macchine remote di interagire in modo consistente.

In un certo senso, il middleware è come il sistema operativo di un sistema distribuito; per questo motivo si tratta questo argomento in un libro sui sistemi operativi. Di contro, *non* è realmente un sistema operativo, per cui l'analisi non entrerà troppo nel dettaglio. Per un maggiore approfondimento fate riferimento al volume *Sistemi distribuiti* (traduzione italiana, Tanenbaum e Van Steen, 2006).

Nel resto del capitolo analizzeremo velocemente l'hardware utilizzato in un sistema distribuito (cioè la rete di computer sottostante), poi il suo software di comunicazione (i protocolli di rete). In seguito considereremo una gamma di paradigmi utilizzati in questi sistemi.

8.3.1 Hardware di rete

I sistemi distribuiti sono costruiti al di sopra delle reti di computer, per cui una breve introduzione a questo argomento è d'obbligo. Esistono due importanti tipologie di reti, le **LAN** (**local area networks**), che servono un edificio o un campus e le **WAN** (**wide area networks**), che possono essere estese quanto una città, uno stato o anche su scala mondiale. Il tipo più importante di LAN è Ethernet, quindi la esamineremo come esempio di LAN. Come esempio di WAN esamineremo Internet, anche se dal punto di vista tecnico Internet non è una rete unica ma una federazione di migliaia di reti separate. Tuttavia, per i nostri scopi, è sufficiente pensare a essa come a una WAN.

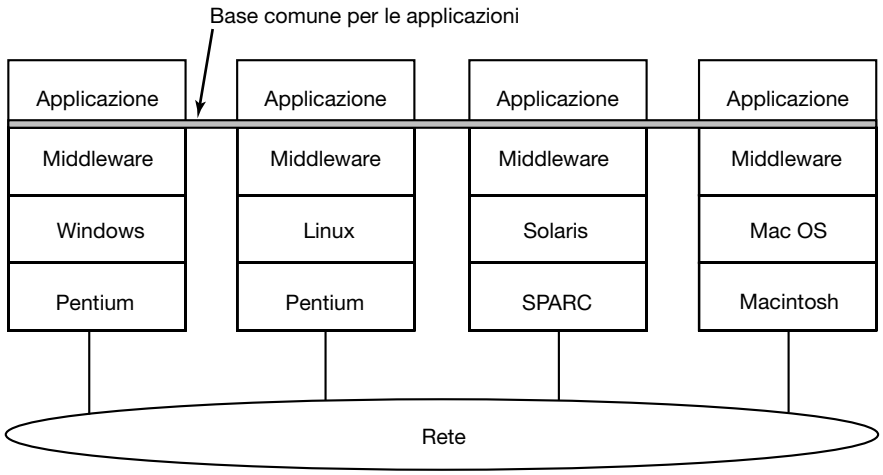

Figura 8.27 Posizionamento del middleware in un sistema distribuito.

Ethernet

La classica Ethernet, descritta nello Standard IEEE 802.3, consiste di un cavo coassiale al quale sono collegati un certo numero di computer. Il cavo è chiamato Ethernet, in riferimento all'*etere luminifero* (*luminiferous ether*) attraverso il quale un tempo si pensava si propagassero le onde elettromagnetiche. (Quando il fisico inglese del XIX secolo James Clerk Maxwell scoprì che le onde elettromagnetiche potevano essere descritte da un'equazione d'onda, gli scienziati ritennero che lo spazio dovesse essere riempito con una qualche sostanza eterea nella quale le radiazioni potessero propagarsi. Solo nel 1887 dopo il famoso esperimento MichelsonMorley, che fallì nell'identificare l'etere, i fisici realizzarono che le radiazioni potevano propagarsi nel vuoto.)

Nella primissima versione di Ethernet, un computer era collegato al cavo attraverso un buco in mezzo al cavo stesso e avvitandovi un filo che portava al computer. Fu chiamato **vampire tap**, schematicamente mostrato nella Figura 8.28(a). Dato che era difficile far funzionare correttamente i tap, essi furono sostituiti da connettori appropriati. Tuttavia, dal punto di vista elettrico, tutti i computer erano connessi come se i cavi sulle loro schede d'interfaccia di rete fossero saldati insieme.

Per spedire un pacchetto su una Ethernet, un computer verifica prima sul cavo per vedere se altri computer stanno trasmettendo in quel momento. Altrimenti, inizia semplicemente a trasmettere un pacchetto, che consiste di una breve intestazione seguita da un carico utile da 0 a 1500 byte. Se il cavo è in uso, il computer attende la fine della trasmissione in atto, poi inizia il suo invio.

Se due computer iniziano a trasmettere simultaneamente, ne risulta una collisione rilevata da entrambi. Entrambi rispondono terminando le loro trasmissioni, aspettando un tempo casuale tra 0 e T μs e poi ricominciando di nuovo. Se si verifica un'altra collisione, tutti i computer che si scontrano randomizzano l'attesa nell'intervallo da 0 a $2T$ μs e poi provano nuovamente. Per ogni ulteriore collisione, l'intervallo massimo di attesa è raddoppiato, riducendo la possibilità di altre collisioni. Questo algoritmo è conosciuto come **binary exponential backoff**, visto precedentemente per ridurre l'overhead del polling sui lock.

Una Ethernet ha una lunghezza massima del cavo e un numero massimo di computer che si possono collegare. Per superare ciascuno di questi limiti, grandi edifici o campus possono essere cablati con più Ethernet, a loro volta collegate da dispositivi chiamati **bridge**.

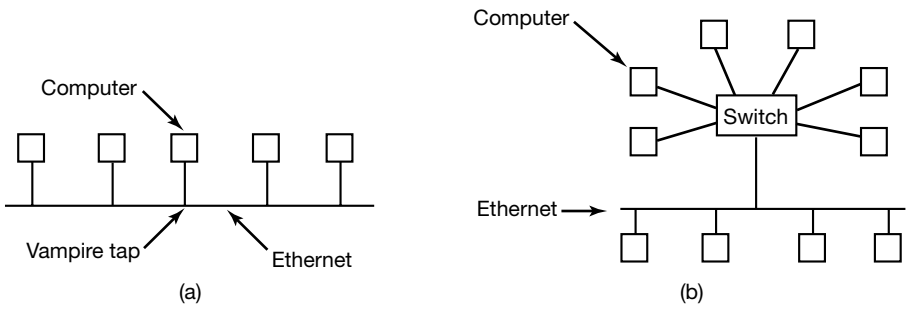

Figura 8.28 (a) Ethernet classica. (b) Ethernet con switch.

Un bridge permette che il traffico passi da una Ethernet all'altra, quando la sorgente è da un lato e il destinatario dall'altro. Per evitare il problema delle collisioni, le Ethernet moderne utilizzano degli *switch*, come mostrato nella Figura 8.28(b). Ogni switch ha un certo numero di porte, alle quali possono essere connessi un computer, una Ethernet o un altro switch. Quando un pacchetto evita con successo tutte le collisioni e raggiunge lo switch, è messo nel buffer ed è inviato sulla porta dove si trova la macchina di destinazione. Dotando ogni computer della sua porta esclusiva, possono essere eliminate tutte le collisioni, a costo di switch più grandi. Sono anche possibili dei compromessi, con solo pochi computer per porta. Nella Figura 8.28(b) una Ethernet classica con più computer collegati a un cavo da un *vampire tap* è a sua volta connessa a una delle porte dello switch.

Internet

Internet si è evoluta a partire da ARPANET, una rete sperimentale a commutazione di pacchetto fondata dall'Agenzia dei Progetti di Ricerca Avanzata del Dipartimento della Difesa degli Stati Uniti. Entrò in funzione nel dicembre 1969 con tre computer in California e uno nello Utah. Fu progettata all'apice della guerra fredda per essere una rete tollerante ai malfunzionamenti, in grado di continuare a trasmettere traffico militare anche nell'eventualità di attacchi nucleari diretti a più punti della rete, tramite l'automatico reinstradamento del traffico attorno alle macchine fuori uso.

ARPANET crebbe rapidamente negli anni '70, comprendendo alla fine centinaia di computer. In seguito furono collegate a essa una rete radio a pacchetti, una rete satellitare e infine migliaia di Ethernet, dando vita alla federazione di reti che ora conosciamo come Internet.

Internet è composta di due tipi di computer, host e router. Gli **host** sono PC, notebook, palmari, server, mainframe e altri computer di proprietà di individui o di aziende che si vogliono connettere a Internet. I **router** sono computer specializzati nella commutazione (switching) che accettano pacchetti in arrivo su una di tante linee in entrata e li inviano alle loro destinazioni lungo una delle tante vie in uscita. Un router è simile allo switch della Figura 8.28(b), ma si differenzia per caratteristiche che al momento non ci interessano. I router sono collegati fra di loro a formare reti vaste, in cui ogni router ha cavi o fibre verso molti altri router e host. Le grandi reti di router nazionali o mondiali sono gestite da compagnie telefoniche e ISP (Internet service providers) per i loro clienti.

La Figura 8.29 mostra una sezione di Internet. In alto troviamo una delle linee dorsali (backbone), gestite normalmente da un operatore di dorsali. Consiste di un certo numero di router collegati con fibre ottiche a elevata ampiezza di banda, con connessioni verso le dorsali effettuate da altre compagnie telefoniche (concorrenti). Di solito, nessun host è connesso direttamente alla dorsale, eccetto macchine test o di manutenzione della compagnia telefonica.

Collegate ai router delle dorsali ci sono le reti di zona e i router degli ISP, tramite connessioni di fibre ottiche di media velocità. A loro volta, le Ethernet aziendali hanno ognuna un router, questi router sono collegati ai router delle reti di zona. I router degli ISP sono collegati ai banchi dei modem usati dai clienti degli ISP. In questo modo, ogni host in Internet ha almeno un percorso, e spesso più percorsi, verso ogni altro host.

Tutto il traffico in Internet è inviato sotto forma di pacchetti. Ogni pacchetto porta al suo interno il proprio indirizzo di destinazione: questo indirizzo è utilizzato per l'instradamento. Quando un pacchetto arriva in un router, esso estrae l'indirizzo di destinazione e lo

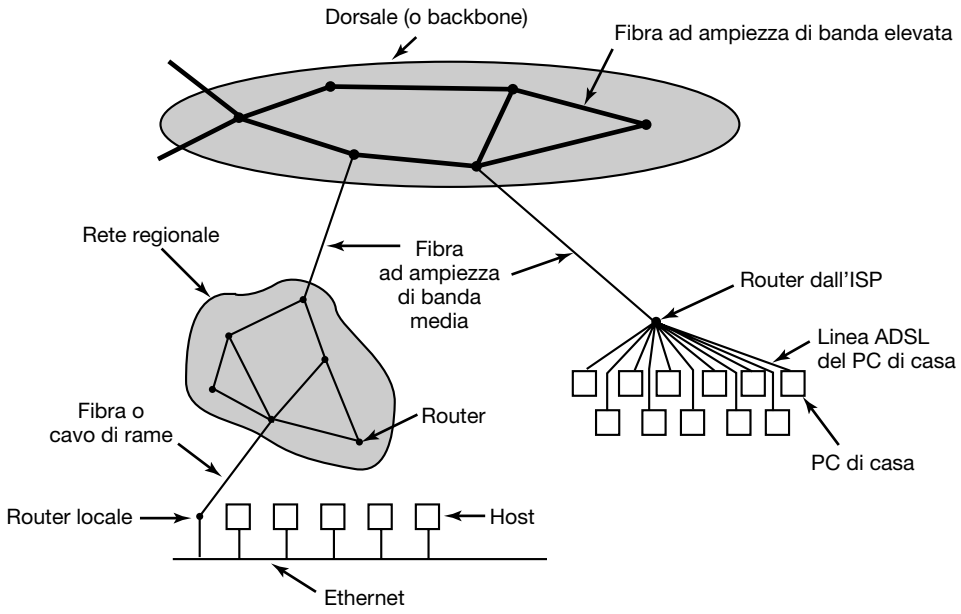

Figura 8.29 Sezione di Internet.

verifica (per una parte) in una tabella per trovare su quale linea in uscita deve inviare il pacchetto, cioè a quale router. Questa procedura è ripetuta finché il pacchetto non raggiunge l'host di destinazione. Le tabelle d'instradamento sono estremamente dinamiche e continuamente aggiornate della caduta o della riattivazione dei router, dei collegamenti e di qualsiasi cambiamento delle condizioni del traffico.

8.3.2 Protocolli e servizi di rete

Le reti di computer forniscono ai loro utenti (host e processi) determinati servizi, che implementano utilizzando regole stabilite sullo scambio di messaggi legali. Di seguito forniremo una breve introduzione a questi argomenti.

Servizi di rete

Le reti di computer forniscono servizi agli host e ai processi che le utilizzano. Il modello del **servizio orientato alla connessione** (connection-oriented) è basato sul sistema telefonico. Per comunicare con qualcuno, si prende il telefono, si digita il numero, si parla e poi si aggancia. Analogamente, per utilizzare un servizio di rete orientato alla connessione, l'utente del servizio stabilisce prima una connessione, la usa e poi la rilascia. L'aspetto essenziale della connessione è che essa agisce come un tubo: il mittente spinge dentro gli oggetti (i bit) a una estremità e il destinatario li riceve nello stesso ordine all'estremità opposta.

Diversamente, il modello del **servizio senza connessione** (*connectionless*) ha come modello il sistema postale. Ogni messaggio (lettera) porta l'indirizzo di destinazione com-

pleto e ogni messaggio è convogliato attraverso il sistema indipendentemente da tutti gli altri. Di norma, quando due messaggi sono spediti alla stessa destinazione, quello spedito per primo sarà il primo ad arrivare. Tuttavia, è possibile che quello spedito per primo possa subire un ritardo, cosicché arrivi prima il secondo. Con un servizio orientato alla connessione questo sarebbe impossibile.

Ogni servizio si può classificare in base alla **qualità del servizio**. Alcuni sono affidabili, nel senso che non perdono mai dati. Solitamente, un servizio affidabile è implementato con il destinatario che conferma la ricezione di ciascun messaggio rispedendo uno speciale **pacchetto di conferma di ricezione** (*acknowledgement packet*), in modo che il mittente abbia la certezza che il messaggio è arrivato. Il processo di conferma di ricezione provoca costi e ritardi, necessari per individuare la perdita di pacchetti, ma che rallentano il tutto.

Il trasferimento di file è una tipica situazione in cui è appropriato un servizio orientato alla connessione affidabile. Il proprietario del file vuole essere certo che tutti i bit arrivino correttamente e nello stesso ordine di spedizione. Solo pochi utenti preferirebbero un servizio di trasferimento di file più veloce che occasionalmente mescolasse o perdesse anche pochi bit.

Il servizio affidabile orientato alla connessione ha due varianti secondarie: sequenze di messaggi (*message sequence*) e flussi di byte (*byte stream*). Nella prima, sono preservati i confini dei messaggi. Quando sono spediti due messaggi da 1 KB, essi arrivano come due distinti messaggi da 1 KB e mai come un messaggio da 2 KB. Nella seconda variante, la connessione è semplicemente un flusso (stream) di byte, senza divisione tra i messaggi. Quando arrivano al destinatario 2 KB, non c'è modo di indicare se siano stati spediti come un messaggio da 2 KB, due messaggi da 1 KB o 2048 messaggi da 1 byte. Se le pagine di un libro fossero spedite sulla rete a un impianto di stampa come messaggi separati, mantenere i confini dei messaggi potrebbe essere importante. D'altra parte, nel caso di un terminale che si collega a un sistema remoto in time sharing, basterebbe un flusso di byte dal terminale al computer.

Per alcune applicazioni il ritardo generato dalla conferma di ricezione è inaccettabile. Un'applicazione di questo genere è il traffico vocale digitalizzato. Gli utenti telefonici preferiscono sentire un po' di rumore sulla linea o una voce confusa ogni tanto piuttosto che un ritardo dovuto alla conferma di ricezione.

Non tutte le applicazioni richiedono connessioni. Per esempio, per testare la rete tutto ciò che serve è un modo per spedire un singolo pacchetto che abbia una elevata probabilità di arrivo, senza alcuna garanzia. Un servizio senza connessione non affidabile (cioè privo di conferma di ricezione) è spesso chiamato **servizio datagram**, per analogia con il servizio dei telegrammi, che allo stesso modo non fornisce una conferma di ricezione al mittente.

In altre situazioni, la convenienza di non dover stabilire una connessione per spedire un breve messaggio è auspicabile, ma l'affidabilità è essenziale. Per queste applicazioni si può fornire il **servizio datagram con conferma** (*acknowledged datagram service*). È come spedire una lettera raccomandata con ricevuta di ritorno. Quando ritorna la ricevuta, il mittente è assolutamente certo che la lettera sia stata consegnata al destinatario e che non sia stata smarrita lungo il percorso.

Un altro servizio è quello del **servizio richiesta-risposta** (**request-reply service**). In questo servizio il mittente trasmette un singolo datagram contenente una richiesta; la replica contiene la risposta. Il servizio richiesta-risposta è comunemente utilizzato per implementare le comunicazioni nel modello client-server: il client invia una richiesta e il server gli risponde. La Figura 8.30 riassume i tipi di servizio di cui si è appena discusso.

	Servizio	Esempio
Orientato alla connessione	Flusso di messaggi affidabile	Sequenza di pagine di un libro
	Flusso di byte affidabile	Login remote
	Connessione inaffidabile	Voce digitalizzata
Senza connessione	Datagram inaffidabile	Pacchetti di test della rete
	Datagram con conferma	Posta raccomandata
	Richiesta-risposta	Interrogazione di una base di dati

Figura 8.30 Sei differenti tipologie di servizio rete.

Protocolli di rete

Tutte le reti hanno regole altamente specialistiche relative a quali messaggi possono essere spediti e a quali repliche possono essere restituite in risposta a questi messaggi. Per esempio, in certe circostanze (per esempio, il trasferimento di file), quando un messaggio è spedito da una sorgente a una destinazione, si richiede alla destinazione di spedire una conferma di ricevuta indicante la corretta ricezione del messaggio. In altre circostanze (per esempio, nella telefonia digitale), non ci si aspetta questa conferma di ricezione. L'insieme di regole tramite le quali comunicano particolari computer è detto **protocollo**. Esistono molti protocolli, inclusi protocolli router-router, protocolli host-host e altri. Per un'analisi approfondita delle reti di computer e dei loro protocolli, si faccia riferimento a *Reti di computer* (traduzione italiana, Tanenbaum, 2003).

Per stratificare protocolli diversi uno sopra all'altro tutte le reti moderne utilizzano quella che si definisce **pila di protocolli** (*protocol stack*). A ogni strato vengono trattate questioni diverse. Per esempio, al livello più basso i protocolli definiscono come indicare l'inizio e la fine di un pacchetto nel flusso dei bit. A un livello superiore i protocolli trattano di come instradare i pacchetti dalla sorgente alla destinazione attraverso reti complesse. A un livello ancora più alto, essi si assicurano che tutti i pacchetti, in un messaggio multipacchetto, siano arrivati correttamente e nell'ordine corretto.

Dal momento che molti sistemi distribuiti utilizzano Internet come base, i protocolli chiave sfruttati da questi sistemi sono i due principali protocolli di Internet: **IP** e **TCP**. **IP** (**Internet protocol**) è un protocollo datagram nel quale un mittente inserisce un datagram di circa 64 KB sulla rete, auspicando che esso arrivi. Non gli è data alcuna garanzia. Mentre passa attraverso Internet il datagram può essere frammentato in pacchetti più piccoli, che viaggiano in maniera indipendente, forse lungo percorsi diversi. Raggiunta la loro destinazione, tutti i pezzi sono assemblati nell'ordine corretto e consegnati.

Attualmente sono in uso due versioni di IP, la v4 e la v6. Al momento la v4 ha ancora il sopravvento, per cui qui descriveremo quella, ma la v6 sta avanzando. Ogni pacchetto v4 comincia con un'intestazione di 40 byte e contiene fra gli altri campi un indirizzo sorgente di 32 bit e un indirizzo destinazione di 32 bit. Questi sono denominati indirizzi IP e formano le basi dell'instradamento in Internet. Sono convenzionalmente scritti come quattro numeri decimali che vanno da 0 a 255, separati dai punti, come 192.31.231.65. Quando giunge

un pacchetto al router, questo estrae l'indirizzo IP di destinazione e lo usa per l'instradamento del pacchetto.

Dato che i datagram IP non comportano conferma, l'IP da solo non è sufficiente per ottenere comunicazioni affidabili su Internet. Per fornire comunicazioni affidabili solitamente si stratifica un altro protocollo al di sopra dell'IP, il **TCP** (**transmission control protocol**). Il TCP utilizza l'IP per fornire flussi orientati alla connessione. Per usare il TCP, un processo prima stabilisce una connessione a un processo remoto. Il processo richiesto è specificato dall'indirizzo IP di una macchina e da un numero di porta di quella macchina, su cui stanno in ascolto i processi interessati alla ricezione di connessioni in ingresso. A connessione avvenuta, semplicemente pompa byte lungo di essa con la garanzia che essi arrivino dall'altro capo integri e nell'ordine corretto. L'implementazione del TCP ottiene questa garanzia usando numeri di sequenza, somme di controllo (checksums) e la ritrasmissione dei pacchetti ricevuti in modo scorretto. Tutto ciò è trasparente ai processi di trasmissione e di ricezione. Tutto quello che vedono è una comunicazione fra processi affidabile, proprio come una pipe UNIX.

Per vedere come interagiscono tutti questi protocolli considerate il caso di un piccolissimo messaggio che non debba essere frammentato, ad alcun livello. L'host è su una Ethernet connessa a Internet. Che cosa accade esattamente? Il processo utente genera il messaggio e fa una chiamata di sistema per inviarlo su una connessione TCP precedentemente stabilita.

La pila dei protocolli del kernel aggiunge un'intestazione TCP e poi un'intestazione IP davanti al messaggio. Poi va al driver Ethernet, che aggiunge un'intestazione Ethernet che direziona il pacchetto al router sulla Ethernet. Questo router a sua volta inserisce il pacchetto su Internet, come descritto nella Figura 8.31.

Per stabilire una connessione con un host remoto (o anche per spedirgli un datagram), serve conoscere il suo indirizzo IP. Poiché gestire elenchi di indirizzi IP a 32 bit è scomodo per gli utenti, fu inventato uno schema chiamato **DNS** (**domain name system** – sistema di denominazione dei domìni), una base di dati che mappi i nomi ASCII degli host ai loro indirizzi IP. In questo modo è possibile usare il nome DNS *star.cs.vu.nl* invece del suo corrispondente indirizzo IP 130.37.24.6. I nomi DNS sono largamente conosciuti, dato che gli

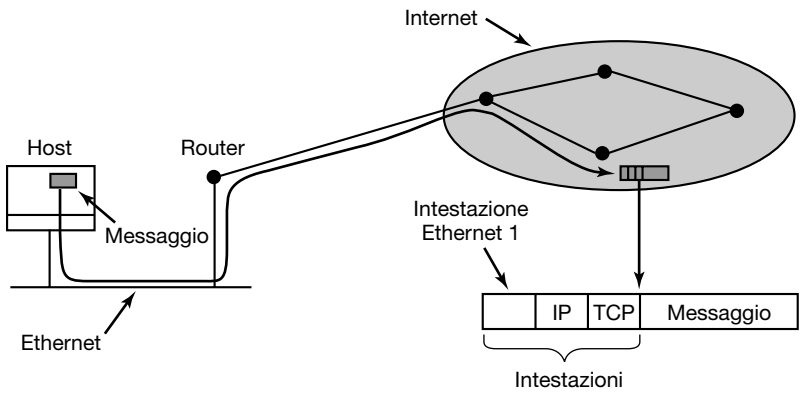

Figura 8.31 Accumulazione di intestazioni dei pacchetti.

indirizzi Internet di posta elettronica sono nel formato *nome-utente@nome-host-DNS*. Questo sistema di nomenclatura consente al programma di posta elettronica sull'host mittente di cercare l'indirizzo IP dell'host di destinazione, stabilire una connessione TCP con il processo demone della posta elettronica e spedire il messaggio come un file. Il *nome-utente* è spedito per identificare in quale casella postale posizionare il messaggio.

8.3.3 Middleware basato sui documenti

Adesso che abbiamo acquisito alcune cognizioni sulle reti e sui protocolli possiamo cominciare ad analizzare i diversi livelli di middleware che possono posizionarsi sulla rete base per produrre un paradigma coerente per le applicazioni e gli utenti. Partiremo da un esempio semplice ma ben conosciuto: il World Wide Web. Il Web fu inventato nel 1989 da Tim BernersLee al CERN, il centro europeo per la ricerca fisico-nucleare e si è allargato nel mondo a macchia d'olio.

Il paradigma originale che sta alla base del Web è abbastanza semplice: ogni computer può contenere uno o più documenti, chiamati **pagine web**. Ciascuna pagina web contiene testo, immagini, icone, suoni, filmati e cose simili, così come **link ipertestuali** o **hyperlink** (puntatori) ad altre pagine web. Quando, tramite un programma chiamato **browser web**, un utente richiede una pagina web, questa è visualizzata sullo schermo. Cliccando su un link la pagina attuale viene sostituita sullo schermo dalla pagina cui il link punta. Sebbene recentemente siano stati aggiunti al Web molti fronzoli, il paradigma sottostante rimane chiaramente valido: il Web è un immenso grande grafo orientato di documenti che puntano ad altri documenti, come mostrato nella Figura 8.32.

Ogni pagina web ha un indirizzo univoco, chiamato **URL** (**uniform resource locator**), nella forma *protocollo://nome-DNS/nome-file*. Nella maggior parte dei casi il protocollo è *http* (*hypertext transfer protocol*), ma esistono anche *ftp* e altri. Quindi segue il nome DNS dell'host contenente il file. Alla fine vi è il nome locale del file che indica quale file si desidera.

Il modo in cui l'intero sistema sta insieme è il seguente. Il Web fondamentalmente è un sistema client-server, con l'utente che è il client e il sito web che è il server. Quando l'utente fornisce un URL al browser, sia digitando un indirizzo o cliccando su un link iperte-

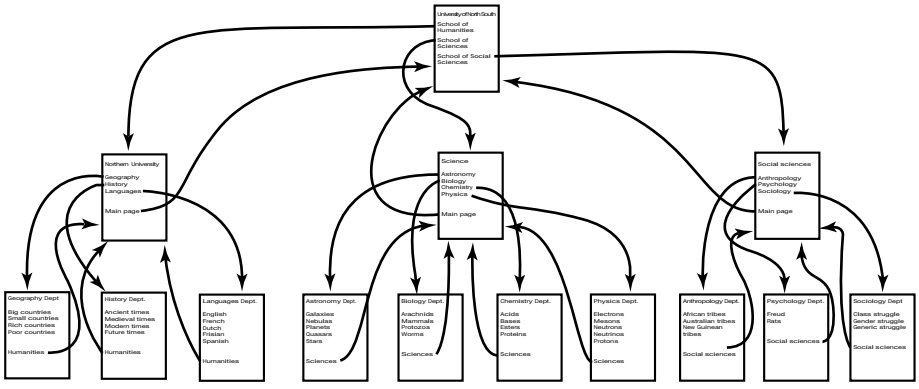

Figura 8.32 Il Web è un grande grafo orientato di documenti.

stuale della pagina attuale, il browser mette in atto alcuni passaggi per prelevare la pagina web richiesta.

Come semplice esempio supponete che l'URL fornito sia *http://minix3.org/doc/faq.html*. Per ottenere la pagina il browser fa i seguenti passaggi.

1. Il browser chiede al DNS l'indirizzo IP di *www.minix3.org*.
2. Il DNS risponde con 130.37.20.20.
3. Il browser esegue una connessione TCP alla porta 80 su 130.37.20.20.
4. A seguire invia una richiesta per il file *doc/faq.html*.
5. Il server *www.acm.org* invia il file *doc/faq.html*.
6. La connessione TCP è rilasciata.
7. Il browser visualizza tutto il testo contenuto in *doc/faq.html*.
8. Il browser prende e visualizza tutte le immagini contenute in *doc/faq.html*.

Approssimativamente questa è la base del Web e del suo funzionamento. A oggi sono state aggiunte molte altre caratteristiche, come gli stili, le pagine web dinamiche generate al momento, le pagine web contenenti piccoli programmi o script che si eseguono sulla macchina del client e altro, argomenti però al di là dello scopo di questo libro.

8.3.4 Middleware basato sul file system

L'idea alla base del Web è rendere un sistema distribuito simile a una gigantesca raccolta di documenti collegati tramite link ipertestuali. Un secondo approccio è considerarlo come un grandissimo file system. In questo paragrafo analizzeremo alcune delle questioni riguardanti la progettazione di un file system su scala mondiale.

Usare un modello a file system per un sistema distribuito significa che vi è un solo file system globale, con utenti sparsi nel mondo in grado di leggere e scrivere i file cui sono autorizzati. La comunicazione è ottenuta tramite un processo che scrive dati in un file e gli altri che li leggono. Anche in questo caso sono presenti molti dei problemi relativi ai file system standard, ma ne sorgono anche altri relativi alla distribuzione.

Modello di trasferimento

Il primo problema è la scelta fra il **modello upload/download** e il **modello ad accesso remoto**. Nel primo, mostrato nella Figura 8.33(a), un processo accede a un file per prima cosa copiandolo dal file server dove risiede. Se il file deve solo essere letto, la lettura è fatta localmente, per avere alte prestazioni. Se dev'essere scritto, è scritto localmente. Quando il processo ha terminato la scrittura, il file aggiornato è inviato al server. Con il modello ad accesso remoto, il file rimane sul server e i client inviano i comandi per fare il lavoro sul server, come illustrato nella Figura 8.33(b).

I vantaggi del modello upload/download stanno nella sua semplicità e nel fatto che trasferire interi file in un sol colpo è più comodo che trasferirli a piccoli pezzi. Gli svantaggi consistono nella necessità di spazio sufficiente in locale per l'intero file, nel fatto che lo spostamento di un file intero è uno spreco qualora ne servano solo delle piccole parti e nella possibilità che sorgano dei problemi di consistenza nel caso di molteplici utenti concorrenti.

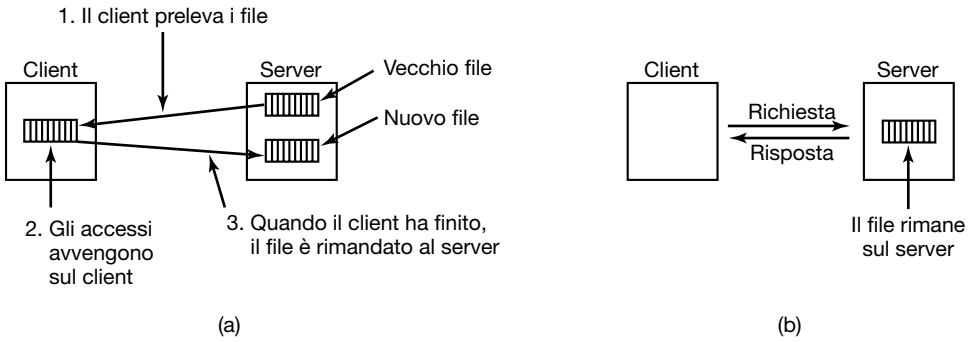

Figura 8.33 (a) Il modello upload/download. (b) Il modello ad accesso remoto.

Gerarchia della directory

I file sono solo una parte della storia. L'altra parte è il sistema delle directory. Tutti i file system distribuiti supportano directory che contengono molteplici file. Il successivo problema progettuale è se tutti i client abbiano la medesima vista della gerarchia della directory. Come esempio da prendere in esame considerate la Figura 8.34. Nella Figura 8.34(a) abbiamo due file server, ognuno contenente tre directory e alcuni file. Nella Figura 8.34(b) abbiamo un sistema in cui tutti i client (e altre macchine) hanno la stessa vista del file system distribuito. Se il percorso */D/E/x* è valido su di una macchina, lo è anche su tutte le altre.

Diversamente nella Figura 8.34(c), macchine diverse hanno viste diverse del file system. Per ripetere l'esempio precedente, il percorso */D/E/x* potrebbe essere valido sul client 1 ma non sul client 2. In sistemi che gestiscono molteplici file server tramite il mounting (montaggio) remoto, la Figura 8.34(c) è la norma. È flessibile e semplice da implementare, ma ha lo svantaggio di non far sì che l'intero sistema si comporti come un solo sistema time sharing dei vecchi tempi. In un sistema time sharing il file system appare lo stesso a qualsiasi processo, come nel modello della Figura 8.34(b). Questa proprietà rende un sistema più semplice da programmare e da capire.

Una questione strettamente correlata è se vi sia o meno una *root* directory globale, che tutte le macchine riconoscano come *root*.

Un sistema per avere una root directory globale è che vi sia una root contenente una voce per ciascun server e nient'altro. A queste condizioni, i percorsi hanno una forma del tipo */server/path*, con i suoi svantaggi, ma quanto meno con il vantaggio di essere lo stesso in tutto il sistema.

Trasparenza del naming

Il problema principale di questo formato di nomenclatura è che non è completamente trasparente. In questo contesto sono importanti due forme di trasparenza che vale la pena distinguere. La prima, la **trasparenza dalla posizione**, significa che il percorso non dà adito a dubbi su dove si trovi il file. Un percorso del tipo */server1/dir1/dir2/x* indica a tutti che *x*

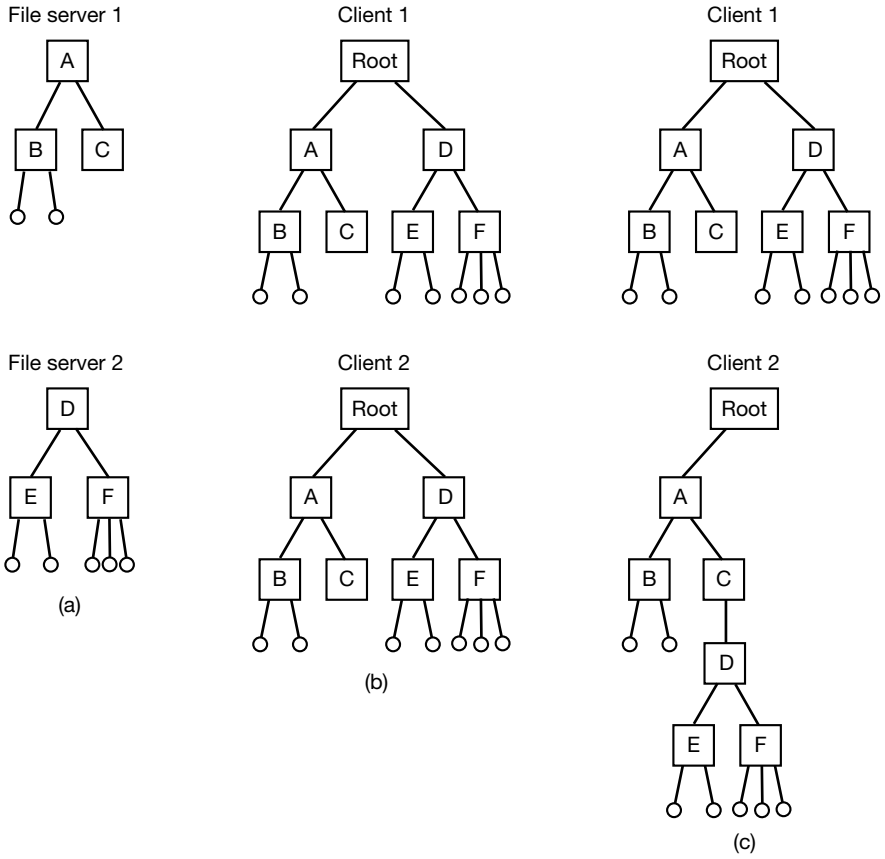

Figura 8.34 (a) Due file server. I quadrati sono le directory e i cerchiolini sono i file. (b) Un sistema nel quale i client hanno la stessa vista del file system. (c) Un sistema nel quale i client possono avere viste differenti del file system.

è posizionato sul server 1, ma non dice dove il server si trovi. Il server è libero di muoversi ovunque voglia nella rete senza che il nome del percorso debba essere modificato. Ciò significa che questo sistema ha la *location transparency*.

Supponete tuttavia che il file *x* sia estremamente grande e stia alla stretta sul server 1. Inoltre supponete che sul server 2 vi sia una gran quantità di spazio. Il sistema dovrebbe a ragion veduta muovere *x* sul server 2 in automatico. Sfortunatamente, quando il primo componente dei nomi del percorso è il server, il sistema non può spostare il file a un altro server automaticamente, anche se *dir1* e *dir2* esistono su entrambi i server. Il problema è che lo spostamento del file in automatico cambia il suo path name da */server1/dir1/dir2/x* a */server2/dir1/dir2/x*. I programmi che hanno la prima stringa inserita al loro interno smetteranno di funzionare se il percorso cambia. Un sistema in cui i file possono essere spostati senza che i loro nomi cambino è detto avere **indipendenza alla posizione**. Un sistema di-

stribuito che incorpori nomi di macchine o di server nei path name è chiaramente non indipendente dal posizionamento. Nemmeno uno basato sul *remote mounting* lo è, dato che non è possibile spostare un file da un gruppo di file (l'unità del *mounting*) a un altro ed essere ancora in grado di usare il vecchio path name. L'indipendenza dalla posizione non è semplice da ottenere, ma è una proprietà auspicabile in un sistema distribuito.

Per riassumere quanto detto finora, gli approcci comuni alla nomenclatura dei file e delle directory in un sistema distribuito sono tre.

1. Macchina + nome del percorso, come in */macchina/percorso* o *macchina:percorso*.
2. Mounting di file system remoti sulla gerarchia dei file locali.
3. Uno spazio dei nomi singolo che appaia uguale su tutte le macchine.

I primi due approcci sono semplici da implementare, specialmente come modo di connettere dei file system esistenti non progettati per un uso distribuito. L'ultimo è difficile e richiede un'attenta progettazione, ma semplifica la vita a utenti e programmatori.

Semantica della condivisione dei file

Quando due o più utenti condividono lo stesso file per evitare problemi è necessario definire con precisione le semantiche di lettura e di scrittura. Nei sistemi a processore singolo le semantiche dichiarano che quando una chiamata di sistema `read` segue una chiamata di sistema `write`, la `read` restituisce il valore appena scritto, come mostrato nella Figura 8.35(a). In modo analogo, quando avvengono due `write` consecutive e poi una `read`, il valore letto è quello memorizzato dall'ultima scrittura. In effetti il sistema fa rispettare un ordine su tutte le chiamate di sistema e tutti i processori vedono lo stesso ordinamento. Faremo riferimento a questo modello con il nome di **consistenza sequenziale**.

In un sistema distribuito, la consistenza sequenziale può ottenersi facilmente purché vi sia un solo file server e i client non mettano i file nella cache. Tutte le `read` e le `write` vanno direttamente al file server, che le processa in modo strettamente sequenziale. In pratica, tuttavia, le prestazioni di un sistema distribuito in cui tutte le richieste dei file devono andare a un singolo server sono spesso scadenti. Questo problema è frequentemente risolto permettendo ai client di avere delle copie locali dei file usati più frequentemente nelle loro cache private. Tuttavia, se il client 1 modifica un file che ha nella cache locale e subito dopo il client 2 legge il file dal server, il secondo client prenderà un file obsoleto, come illustrato nella Figura 8.35(b).

Una soluzione per superare questo problema è la propagazione immediata al server di tutte le modifiche effettuate sui file nelle cache. Sebbene concettualmente semplice, questo metodo non è efficace. Un'alternativa è quella di rilassare la semantica della condivisione dei file. Invece di richiedere una `read` per vedere gli effetti di tutte le `write` precedenti, si potrebbe avere una nuova regola che afferma: "I cambiamenti a un file aperto sono inizialmente visibili solo al processo che li ha effettuati. Solo una volta che il file è stato chiuso i cambiamenti sono visibili agli altri processi". Adottare una regola di questo genere non modifica quanto accade nella Figura 8.35(b), ma ridefinisce il comportamento reale (*B* che prende il valore originale del file) come quello corretto. Quando il client 1 chiude il file, restituisce una copia al server, in modo che le successive `read` prendano il nuovo valore, come richiesto. Questa regola semantica è diffusamente implementata ed è conosciuta come **semantica di sessione**.

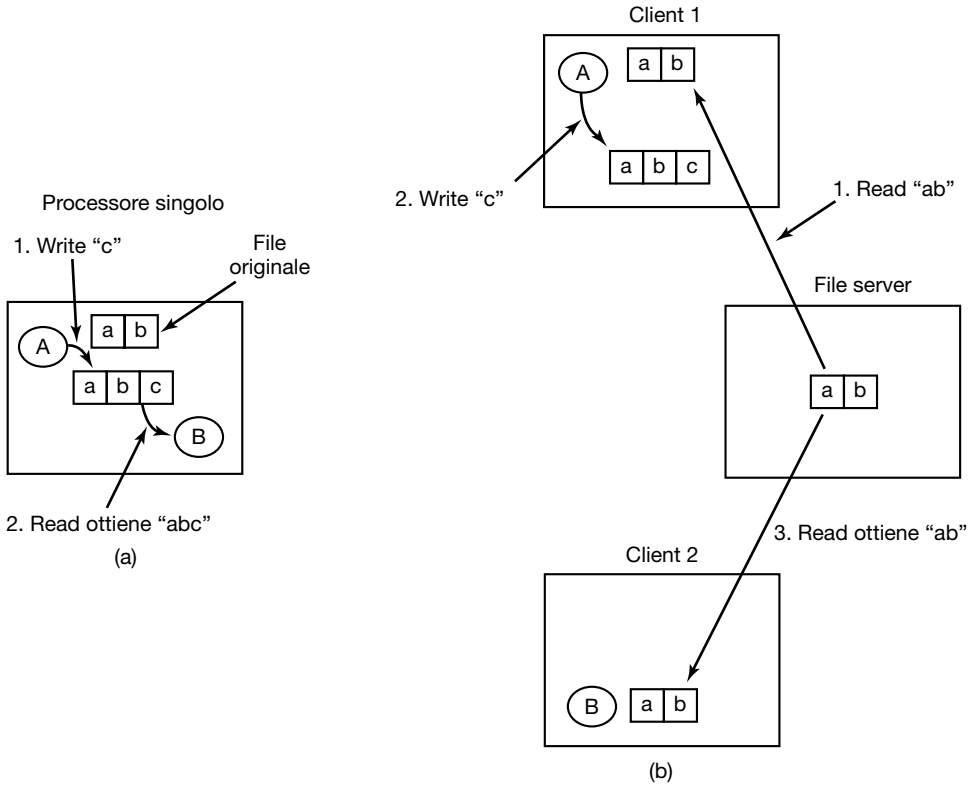

Figura 8.35 (a) Consistenza sequenziale. (b) In un sistema distribuito che fa uso della cache, la lettura di un file può restituire un valore obsoleto.

L'uso della semantica di sessione porta al problema di che cosa accade quando due o più client mettono nella cache e modificano il medesimo file contemporaneamente. Una soluzione indica che ciascun file è chiuso a turno, il suo valore rispedito al server e che il risultato finale dipende da chi lo chiude per ultimo. Un'alternativa meno piacevole, ma più semplice da implementare, è che il risultato finale è uno dei due candidati, ma non specifica su quale dei due cada la scelta.

Un approccio alternativo alla semantica di sessione è l'uso del modello upload/download, ma ciò causa il *lock* immediato del file appena scaricato. I tentativi degli altri client di scaricare il file sarebbero posticipati a quando il primo client lo abbia restituito. Nel caso della massiccia richiesta di un file, il server potrebbe inviare dei messaggi al client che lo sta utilizzando, sollecitandogli di affrettarsi, ma non è detto che questo possa essere d'aiuto. In conclusione, far sì che la semantica della condivisione dei file funzioni bene è un problema non è semplice e non ci sono soluzioni eleganti ed efficienti.

8.3.5 Middleware basato sugli oggetti

Prestiamo adesso attenzione a un terzo paradigma. Invece di dire che qualunque cosa è un documento o un file, diciamo che tutto è un oggetto. Un **oggetto** è una raccolta di variabili impacchettate con un insieme di procedure d'accesso, chiamate **metodi**. Ai processi non è consentito accedere direttamente alle variabili. Devono invece richiamare i metodi.

Alcuni linguaggi di programmazione, come C++ e Java, sono orientati agli oggetti (object-oriented), ma questi sono oggetti a livello di linguaggio, piuttosto che oggetti runtime. Un sistema ben conosciuto che si basa sugli oggetti run-time è **CORBA** (**common object request broker architecture**) (Vinoski, 1997). CORBA è un sistema client-server, in cui processi client su macchine client possono richiamare operazioni su oggetti posizionati su macchine server (eventualmente remote). CORBA fu progettato per un sistema eterogeneo eseguito su una varietà di piattaforme hardware e sistemi operativi e programmato in una varietà di linguaggi. Per rendere possibile che un client su una piattaforma richiami un server su una piattaforma differente sono interposti fra i client e i server degli **ORB** (**object request broker**), che ne permettono l'unione. Gli ORB hanno un ruolo importante in CORBA, come si capisce anche dal nome stesso dato al sistema.

Ogni oggetto CORBA è definito tramite una definizione d'interfaccia in un linguaggio chiamato **IDL** (**interface definition language**) che descrive quali metodi sono esportati dall'oggetto e che tipi di parametri ognuno si aspetti. La specifica IDL può essere compilata in una procedura client stub e memorizzata in una libreria.

Se il processo del client sa in anticipo che avrà bisogno di accedere a un determinato oggetto, verrà fatto il link del processo al codice del client stub dell'oggetto. La specifica IDL può anche essere compilata all'interno di una procedura **scheletro** usata sul lato server. Nel caso non si conoscano in anticipo gli oggetti che un processo ha bisogno di utilizzare, è possibile richiamarli dinamicamente, ma come ciò funzioni è al di là degli scopi della nostra analisi.

Quando si crea un oggetto CORBA, si crea anche un riferimento a esso e lo si restituisce al processo che lo crea. Questo riferimento è il modo in cui il processo identifica l'oggetto per le successive chiamate ai suoi metodi. Il riferimento può essere passato ad altri processi o memorizzato in una directory di oggetti.

Per richiamare un metodo su un oggetto, un processo client deve prima acquisire un riferimento a quell'oggetto. Il riferimento può arrivare o direttamente dal processo creatore o, più probabilmente, cercandolo per nome o per funzione in alcuni tipi di directory. Una volta che il riferimento dell'oggetto è disponibile, il processo client organizza i parametri per le chiamate dei metodi in una struttura opportuna e poi contatta l'ORB del client. A sua volta, l'ORB del client manda un messaggio all'ORB del server, che richiama il metodo sull'oggetto. L'intero meccanismo è simile alle RPC.

La funzione degli ORB è quella di nascondere tutta la distribuzione a basso livello e i dettagli della comunicazione dal codice del client e del server. In particolare gli ORB nascondono al client la posizione del server, se il server sia un programma binario o uno script, su che hardware e su che sistema operativo è eseguito il server, se l'oggetto sia attualmente attivo e come i due ORB comunicano (per esempio, TCP/IP, RPC, memoria condivisa, e così via).

Nella prima versione di CORBA, il protocollo fra l'ORB del client e l'ORB del server non era specificato, per cui ogni fornitore di ORB usava un diverso protocollo e non ce

n'erano due che potessero comunicare fra loro. Nella versione 2.0 fu specificato il protocollo. Per le comunicazioni via Internet il protocollo si chiama **IIOP** (**Internet interorb protocol**).

Per permettere l'utilizzo di oggetti non scritti per CORBA all'interno di sistemi CORBA, ogni oggetto può essere dotato di un **adattatore dell'oggetto** (**object adapter**). Si tratta di un "involucro" (*wrapper*) che gestisce faccende come la registrazione dell'oggetto, la generazione dei riferimenti all'oggetto e l'attivazione dell'oggetto se è richiamato quando non è attivo. La sistemazione di tutte queste parti di CORBA è illustrata nella Figura 8.36.

Un problema serio di CORBA è che ogni oggetto è posizionato su un solo server, il che significa che le prestazioni saranno disastrose per quegli oggetti molto usati sulle macchine client in giro per il mondo. In pratica CORBA funziona a un livello accettabile solo in sistemi a scala ridotta, come nel connettere processi su un singolo computer, una LAN o all'interno di una singola azienda.

8.3.6 Middleware basato sulla coordinazione

Il nostro ultimo paradigma per un sistema distribuito è chiamato **middleware basato sulla coordinazione**. Cominceremo dal sistema Linda, un progetto di ricerca accademico che diede avvio all'intero movimento.

Linda è un originale sistema di comunicazione e sincronizzazione sviluppato all'Università di Yale da David Gelernter e il suo studente Nick Carriero (Carriero e Gelernter, 1968; Carriero e Gelernter, 1989 e Gelernter, 1985). In Linda, i processi indipendenti comunicano attraverso uno **spazio delle tuple** astratto. Lo spazio delle tuple è globale per l'intero sistema e i processi su qualsiasi macchina possono inserire le tuple nello spazio delle tuple o anche rimuoverle senza interessarsi di come o dove sono memorizzate. Per l'utente, lo spazio delle tuple appare come una grande memoria condivisa globale, come già abbiamo visto in diverse forme, come nella Figura 8.21(c).

Una **tupla** è come una struttura in C o in Java. È composta da uno o più campi, ciascuno dei quali è un valore di una certa tipologia supportato dal linguaggio base (Linda è implementato aggiungendo una libreria a un linguaggio esistente, come il C). Per il C-Linda,

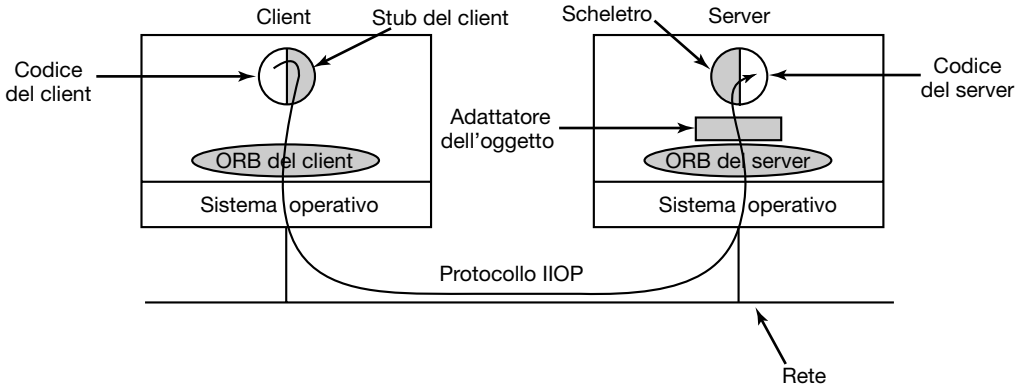

Figura 8.36 Elementi principali di un sistema distribuito basato su CORBA. Le parti di CORBA sono raffigurate in grigio.

le tipologie dei campi includono interi, interi lunghi, numeri in virgola mobile, così come tipologie composte come array (incluse le stringhe) e strutture (ma non altre tuple). Diversamente dagli oggetti, le tuple sono dati puri; non hanno associato alcun metodo. La Figura 8.37 mostra tre esempi di tuple.

Sono fornite quattro operazioni sulle tuple. La prima, *out*, pone una tupla nello spazio delle tuple. Per esempio,

```
out("abc", 2, 5);
```

mette la tupla ("abc", 2, 5) nello spazio delle tuple. I campi di *out* normalmente sono costanti, variabili o espressioni, come in

```
out("matrix-1", i, j, 3.14);
```

che esegue l'output di una tupla con quattro campi, di cui il secondo e il terzo sono determinati dagli attuali valori delle variabili *i* e *j*.

Le tuple sono caricate dallo spazio delle tuple tramite la primitiva *in*. Sono indirizzate tramite il contenuto piuttosto che per nome o per indirizzo. I campi di *in* possono essere espressioni o parametri formali.

Considerate per esempio:

```
in("abc", 2, ?i);
```

Questa operazione cerca nello spazio delle tuple una tupla composta dalla stringa "abc", dall'intero 2 e da un terzo campo contenente qualunque intero (assumendo che *i* sia un intero). Se lo trova, la tupla è rimossa dallo spazio delle tuple e alla variabile *i* è assegnato il valore del terzo campo.

Il *matching* e la rimozione sono atomici, per cui se due processi eseguono la stessa operazione simultaneamente, solo uno dei due ci riuscirà, a meno che non siano presenti due o più tuple corrispondenti. Lo spazio delle tuple può anche contenere più copie della stessa tupla.

L'algoritmo di confronto usato da *in* è semplice. I campi della primitiva *in*, chiamati **template**, sono (concettualmente) confrontati con i campi corrispondenti di ogni tupla nello spazio delle tuple. Si verifica il matching quando si realizzano le seguenti tre condizioni:

1. il template e la tupla devono avere lo stesso numero di campi;
2. le tipologie dei campi corrispondenti devono essere le stesse;
3. ogni costante o variabile nel template deve trovare riscontro nel campo della sua tupla.

I parametri formali, indicati da un punto di domanda seguito da un nome di variabile o di tipo, non prendono parte a questo confronto (a eccezione del controllo del tipo), sebbene quelli che contengono un nome di variabile siano assegnati dopo un confronto con esito positivo.

("abc", 2, 5)
("matrice-1", 1, 6, 3.14)
("famiglia", è_sorella", "Stefania", "Roberta")

Figura 8.37 Tre tuple di Linda.

Se non si trova alcuna tupla, il processo chiamante è sospeso finché un altro processo inserisce la tupla necessaria, al che il chiamante è risvegliato immediatamente e gli è data la nuova tupla. Il fatto che i processi si blocchino e si sblocchino automaticamente significa che se un processo sta per far l'output di una tupla e un altro ne sta per fare l'input, non importa chi va per primo. La sola differenza è che se *in* è eseguito prima di *out* ci sarà un piccolo ritardo prima che la tupla sia disponibile per la rimozione.

Il fatto che i processi si blocchino quando non c'è la tupla necessaria può essere sfruttato per più utilizzi. Può essere usato per esempio per implementare i semafori. Per creare o fare un `up` sul semaforo *S*, un processo può eseguire

```
out("semaphore S");
```

Per fare un down

```
in("semaphore S");
```

Lo stato del semaforo *S* è determinato dalla quantità di tuple ("semaphore S") nello spazio delle tuple. Se non ce n'è alcuna, ogni tentativo di prelevarne una causerà un blocco, finché un altro processo non ne fornisca una.

Oltre a *out* e *in*, Linda ha anche un'altra primitiva, *read*, uguale a *in* eccetto che non rimuove la tupla dal suo spazio. C'è anche la primitiva *eval*, che fa sì che i suoi parametri siano valutati in parallelo e la tupla risultante sia messa nello spazio delle tuple. Questo meccanismo può essere usato per eseguire un calcolo arbitrario. È così che si creano i processi paralleli in Linda.

Publish/subscribe

Il nostro prossimo esempio di modello basato sulla coordinazione fu ispirato da Linda ed è denominato **publish/subscribe** (Oki et al., 1993). È composto da un numero di processi connessi da una rete di trasmissione (*broadcast*). Ogni processo può essere un produttore di informazioni, un consumatore di informazioni o entrambe le cose.

Quando un produttore di informazioni ha un nuovo pezzo di informazione (per esempio il valore di un'azione), esegue il broadcast dell'informazione sulla rete come una tupla. Questa azione è detta **pubblicazione** (**publishing**). Ogni tupla contiene una linea gerarchica di argomenti contenente molteplici campi separati da dei punti. I processi interessati a determinate informazioni possono **iscriversi** (**subscribe**) a determinati argomenti, incluso l'utilizzo di *wildcard* nella linea degli argomenti. L'adesione avviene indicando al processo demone delle tuple sulla stessa macchina di monitorare le tuple pubblicate in base agli argomenti ricercati. Il publish/subscribe si implementa come nella Figura 8.38. Quando un processo ha una tupla da pubblicare, la emette sulla LAN locale. Il demone delle tuple copia nella RAM di ciascuna macchina tutte le tuple trasmesse. Verifica poi la linea degli argomenti per vedere quali processi vi siano interessati, inoltrandone una copia a chiunque lo sia. Le tuple possono anche essere trasmesse su una rete geografica o su Internet, tramite una macchina su ogni LAN che si comporta da router delle informazioni che raccoglie tutte le tuple pubblicate e le invia alle altre LAN per la ritrasmissione. Questo inoltro può essere fatto anche in modo intelligente, inviando una tupla a una LAN remota solo se in questa vi sia almeno un processo *subscriber* che voglia la tupla. Tutto questo necessita che i router (broker) si scambino le informazioni sui rispettivi *subscriber*.

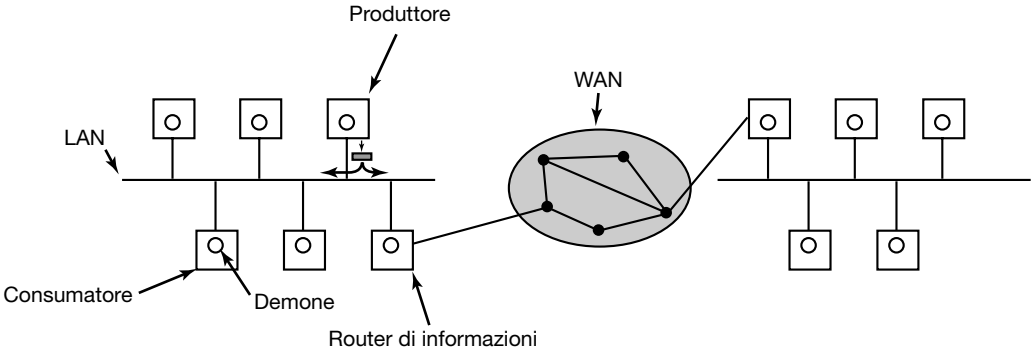

Figura 8.38 Architettura publish/subscribe.

Diversi tipi di semantica possono essere implementati, inclusa la consegna affidabile e la consegna garantita, anche a fronte di un crash. Nell'ultimo caso è necessario memorizzare le vecchie tuple nell'eventualità siano necessarie in seguito. Un modo per memorizzarle è quello di collegare al sistema un sistema di basi di dati e far sì che esegua il *subscribe* a tutte le tuple. Ciò può avvenire racchiudendo il sistema di basi di dati in un adattatore, per consentire a una base di dati esistente di funzionare con il modello publish/subscribe. Quando le tuple arrivano, l'adattatore le cattura e le mette nella base di dati.

Il modello publish/subscribe scollega completamente i produttori dai consumatori, come fa Linda. Tuttavia talvolta è utile sapere che altro c'è al di fuori. Questa informazione può essere acquisita pubblicando una tupla che fondamentalmente chiede: "Chi è interessato a *x* là fuori?" Le risposte tornano indietro sotto forma di tuple che dicono: "*x* interessa a me".

8.4 Stato della ricerca sui sistemi a più processori

Pochi sono gli argomenti, nell'ambito della ricerca sui sistemi operativi, più trattati di multicore, multiprocessori e sistemi distribuiti. Al di là dei problemi diretti rappresentati dalla mappatura delle funzionalità di un sistema su un sistema consistente di più core di elaborazione, vi sono molti problemi ancora aperti relativi a sincronizzazione e coerenza e a come rendere più veloci e affidabili i sistemi di questo genere.

Vi sono diverse ricerche indirizzate alla realizzazione da zero di nuovi sistemi operativi pensati specificamente per l'hardware multicore. Per esempio, il sistema operativo Corey risolve i problemi di prestazioni causati dalla condivisione delle strutture dati su più core (Boyd-Wickizer et al., 2008). Ordinando con attenzione le strutture dati del kernel in modo che non sia necessario condividerle, molti dei colli di bottiglia non si presentano più. Anche Barrelfish (Baumann et al., 2009) è un nuovo sistema operativo, nato da un lato a causa della rapida crescita del numero di core, e dall'altro a causa dell'aumento nella diversità dell'hardware. Si tratta di un sistema operativo modellato sui sistemi distribuiti che hanno come modello di comunicazione lo scambio di messaggi, e non la condivisione della memo-

ria. Vi sono poi altri sistemi operativi pensati per la scalabilità e le prestazioni. Fos (Wentzlaff et al., 2010) è un sistema operativo progettato per adattarsi ad ambienti che vanno dai più piccoli (CPU multicore) ai più grandi (cloud). NewtOS (Hruby et al., 2012; e Hruby et al., 2013) è un nuovo sistema operativo multiserver realizzato tenendo a mente affidabilità (con un design modulare e molti componenti isolati originariamente basati su Minix 3) e prestazioni (tradizionalmente il punto debole di questi sistemi modulari multiserver).

Le architetture multicore non sono adatte solamente ai sistemi operativi di nuova progettazione. In Boyd-Wickizer et al. (2010), i ricercatori studiano i colli di bottiglia ed eliminano quelli che incontrano quando si passa a un sistema Linux con 48 core, dimostrando che, se progettati con attenzione, questi sistemi riescono a essere discretamente scalabili. Clements et al. (2013) svolgono una ricerca sul principio fondamentale che governa se un'API possa essere implementata in maniera scalabile o no, dimostrando che, ogniqualvolta le operazioni di interfaccia commutano, esiste un'implementazione scalabile di quella interfaccia. Forti di questa informazione, i progettisti possono realizzare sistemi operativi più scalabili.

In molte recenti ricerche sui sistemi si è lavorato per fare in modo che applicazioni di grandi dimensioni siano in grado di adattarsi ad ambienti multicore e multiprocessore. Un esempio è il motore di database scalabile descritto da Salomie et al. (2011). Anche in questo caso, la soluzione consiste nell'ottenere la scalabilità replicando il database, invece che cercando di nascondere la natura parallela dell'hardware.

È molto difficile svolgere il debug delle applicazioni parallele ed è complicato riuscire a riprodurre le race condition. Viennot et al. (2013) dimostrano in che modo il replay può aiutare a svolgere il debug del software sui sistemi multicore. Lachaize et al. forniscono un profiler di memoria per i sistemi multicore, mentre Kasikci et al. (2012) presentano un lavoro che non parla solo di come si rilevano le race condition nel software, ma anche come distinguere quelle benigne da quelle maligne.

Molto lavoro, infine, è stato svolto anche per la riduzione del consumo energetico nei sistemi multiprocessore. Chen et al. (2013) propongono l'uso di power container per garantire una migliore granularità nella gestione energetica.

8.5 Riepilogo

I sistemi informatici possono essere resi più veloci e affidabili utilizzando più CPU. Esistono quattro modi di organizzare i sistemi con più CPU: multiprocessori, multicomputer, macchine virtuali e sistemi distribuiti, ciascuno dei quali presenta vantaggi e problemi.

Un multiprocessore è costituito da due o più CPU che condividono una RAM comune; spesso le CPU sono a loro volta costituite da più core. I core e le CPU possono essere connessi fra loro mediante un bus, un crossbar switch o una multistage switching network. Sono possibili diverse configurazioni dei sistemi operativi: può esserci un solo sistema operativo master, con gli altri sistemi slave, oppure un multiprocessore simmetrico, in cui ciascuna CPU può eseguire la propria copia del sistema operativo. In quest'ultimo caso, è necessario prevedere dei lock che garantiscano la sincronizzazione. Quando non è disponibile un lock, la CPU può eseguire uno spinning o uno scambio di contesto. Esistono diversi possibili algoritmi di scheduling: condivisione del tempo, condivisione dello spazio e gang scheduling.

Anche i multicomputer hanno due o più CPU, le quali, però, hanno la propria memoria privata. Non condividono RAM, pertanto tutte le comunicazioni utilizzano il passaggio di messaggi. In alcuni casi, la scheda di rete è provvista della propria CPU, nel qual caso le comunicazioni fra la CPU principale e quella della scheda devono essere organizzate con attenzione, per evitare race condition. La comunicazione a livello utente sui multicomputer utilizza spesso le chiamate a procedure remote (RPC), ma si usa anche la memoria condivisa distribuita. In questo caso, un problema può essere rappresentato dal bilanciamento del carico e per risolverlo si utilizzano diversi algoritmi: avviati dal mittente, avviati dal destinatario e su richiesta.

I sistemi distribuiti sono sistemi non fortemente accoppiati i cui nodi sono computer completi con un completo insieme di periferiche e dotati del proprio sistema operativo. Spesso si tratta di sistemi dispersi in ampie aree geografiche. Sovente, al di sopra del sistema operativo si inserisce un middleware che garantisce un livello uniforme con cui le applicazioni possono interagire. I vari tipi di middleware sono quelli basati sui documenti, basati sul file system, basati sugli oggetti e basati sulla coordinazione. Alcuni esempi sono il World Wide Web, CORBA e Linda.