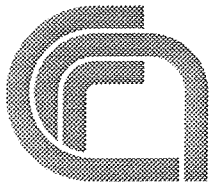
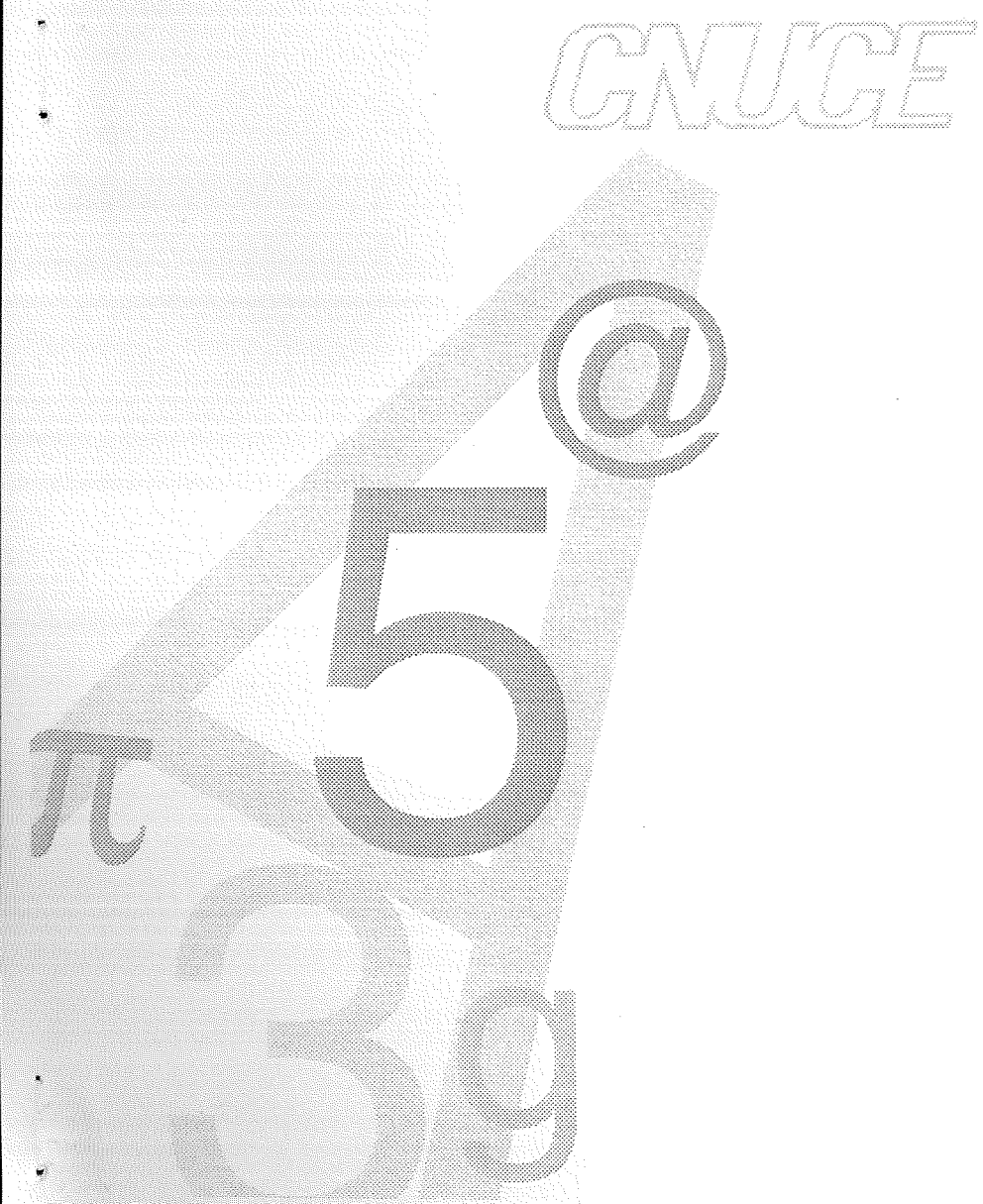


Consiglio Nazionale delle Ricerche

# Simulatore per L'Analisi di Sistemi Multiprocessore Affidabili

*Antonino Cipollone*
*Andrea Bondavalli*

CNUCE C96-07



# 1 Introduzione

La crescente complessità di sistemi informatici e le loro molteplici applicazioni (caratterizzate da un più o meno elevato livello di criticità) ha fatto della dependability un elemento essenziale nella loro progettazione. Con dependability si definisce quella proprietà di un sistema in base alla quale l'utente può giustificatamente riporre fiducia sul servizio fornito [9]. Risulta evidente che essa racchiude diversi aspetti, quali la reliability (affidabilità rispetto alla continuità del servizio), la safety (sicurezza rispetto all'occorrenza di fallimenti catastrofici), la security (sicurezza contro guasti intenzionali), che ne rappresentano gli attributi. Altro indice di dependability è la performability che, dato dalla combinazione degli attributi elencati sopra, misura la bontà prestazionale di un sistema al variare della qualità del servizio [11]. Lo sviluppo di sistemi computazionali "dependable" consiste nell'uso combinato di svariate tecniche, tra cui quelle di *fault tolerance* intese a mascherare l'effetto dei guasti a tempo di esecuzione. Pertanto, la capacità di tollerare l'occorrenza di guasti può essere garantita attraverso tecniche di *error-processing* e meccanismi di *fault treatment*. Le prime, basate normalmente sulla ridondanza (spaziale/temporale) dei servizi e/o su algoritmi di testing, provvedono a rimuovere gli errori dallo stato computazionale del sistema. I secondi, invece, consentono di identificare le unità malfunzionanti allo scopo di evitare che la loro attivazione conduca a successivi errori.

Nell'ultimo decennio sono state proposte parecchie architetture altamente affidabili basate sulla tolleranza di guasti permanenti (gli unici considerati). L'assunzione di soli guasti permanenti limita fortemente il realismo di tali architetture sebbene semplifichi notevolmente gli schemi di mascheramento e di trattamento dei guasti. Esperimenti e giustificazioni teoriche mostrano infatti che la maggior parte dei malfunzionamenti di un sistema sono causati da guasti transienti (presenti per un periodo limitato di tempo) e intermittenti (che permangono nel tempo ma che si manifestano ad intermittenza) [13]. In tale contesto l'uso esclusivo di tecniche di error-processing o di fault treatment risulta poco esaustivo in termini di dependability e performance. Un approccio ottimale è invece risultato dalla combinazione ed integrazione di tali tecniche [4]. Una strategia completa permette infatti di mascherare i malfunzionamenti e di trattare i guasti tenendo conto della loro differente natura. Ciò è indispensabile per bilanciare i due opposti requisiti, ovvero la capacità di mantenere nel sistema tutti i processori funzionanti (o affetti da guasti transienti) e l'abilità di individuare (quindi rimuovere, sostituire o riparare) velocemente quelli con guasti permanenti o intermittenti. Ovviamente la bontà di uno schema completo è strettamente legata alla bontà delle tecniche di error processing e fault treatment, nonché al setting di parametri scelti.

L'analisi di strategie di sistemi multiprocessore dependable viene spesso condotta attraverso soluzioni analitiche di modelli matematici [5, 10, 15, 16]. Talvolta, a causa della complessità del sistema o degli obiettivi di studio, i modelli matematici tendono a divenire estremamente complessi e le drastiche semplificazioni necessarie per rendere il problema trattabile analiticamente limitano l'adeguatezza del modello a rappresentare il sistema stesso. Allo scopo di condurre uno studio libero da restrizioni, si rende necessaria la realizzazione di un simulatore ad hoc per l'analisi di sistemi

multiprocessore dependable [2, 7]. Esso permette l'analisi dettagliata di dependability e performance al variare della strategia di fault tolerance, delle caratteristiche fisiche del sistema, del carico di lavoro e delle caratteristiche che definiscono le classi dei guasti. In un tale contesto, la tecnica di simulazione, sebbene libera da quelle restrizioni tipiche invece dei modelli matematici, si scontra con il problema dei cosiddetti *eventi rari*. Il verificarsi di un guasto in un processore o il fallimento di una missione rappresentano degli eventi rari ma non trascurabili, dato che principalmente da essi dipende l'analisi statistica di interesse. La bassa probabilità legata al verificarsi di questi eventi si riflette sulla durata delle simulazioni che devono crescere notevolmente al fine di riuscire a catturare effetto di gusti e fallimenti e di garantire campioni di osservazioni di lunghezza significativa. Tale problema, di non facile risoluzione, può essere affrontato con l'uso di tecniche particolari basate sulla previsione dell'occorrenza dei guasti e sulla simulazione dettagliata del sistema solo ad intervalli temporali (quelli che comprendono il verificarsi dei guasti e la terminazione del loro effetto). Il problema può essere invece raggirato se si orienta l'analisi verso sistemi non ultra-dependable (dove i guasti si presentano meno raramente) e se le misure da stimare sono scelte con ragionevolezza rispetto ai parametri che caratterizzano il sistema.

In seguito, in Sezione 2, vengono richiamate le due principali tecniche che e' possibile utilizzare per valutare un sistema di computazione e viene descritta piu' in dettaglio la simulazione per eventi discreti, con una breve discussione sulle possibili tecniche di implementazione. In Sezione 3 vengono presentate le caratteristiche principali della classe di sistemi fault tolerant da analizzare, le misure di interesse e i principali parametri in termini di modello dei guasti, strategie di error processing, strategie di fault treatment e carico di input. In Sezione 4 viene descritto il modello di simulazione per la classe di sistemi della Sezione 3 e le caratteristiche principali del simulatore orientato ad eventi. Quindi seguono le Conclusioni ed una breve Appendice con il codice di alcune delle funzionalita' del simulatore.

# 2 Metodi di studio

## *2.1 Metodi analitici e simulazione*

Allo scopo di studiare un sistema, reale o ipotetico, esso viene rappresentato mediante un *modello* che, pur attraverso alcune semplificazioni, ne riproduce con precisione sufficiente gli aspetti basilari e utili per lo studio. A causa delle semplificazioni introdotte, e' possibile studiare solo le caratteristiche del sistema sulle quali ci si e' concentrati nella costruzione del modello. Un aspetto importante di un modello e' quello della sua *adeguatezza*, legato al fatto che uno stesso sistema puo' essere rappresentato da piu' modelli. Pertanto un modello risulta tanto piu' adeguato alla descrizione di un sistema quanto meglio esso rappresenta gli aspetti rilevanti per lo studio in corso. Per ottenere informazioni sul comportamento di un sistema bisogna essere in grado di studiare il modello che lo rappresenta e questo puo' essere fatto attraverso due tecniche:

- Metodi analitici
- Simulazione

I metodi analitici sono estremamente utili in quanto spesso permettono di ottenere relazioni funzionali tra le variabili di ingresso e di uscita del modello, comportando i seguenti vantaggi:

1. Relazioni di causa-effetto tra ingresso ed uscita
2. Costo limitato per la soluzione del modello
3. Facilita' di uso del modello

Purtroppo, le classi di modelli per le quali e' possibile percorrere con successo questa strada sono relativamente poche. Spesso pur riuscendo a trovare delle relazioni funzionali tra ingresso e uscita scritte in forma implicita, non si e' in grado di tradurle in forma esplicita semplice, dovendo cosi' ricorrere a metodi di valutazione numerici dall'alto costo computazionale. Una possibile via di uscita consiste nell'introdurre nel modello semplificazioni che lo rendano risolvibile, correndo pero' il rischio che semplificazioni troppo drastiche rendano il modello inadeguato. In tutti quei casi in cui i modelli realistici del sistema oggetto di studio richiedono livelli di dettaglio tale da rendere il problema non piu' trattabile analiticamente, l'unica alternativa che rimane e' quella di riprodurre la storia degli *stati* del sistema valutando le caratteristiche di comportamento del sistema per mezzo di misure condotte sul modello stesso. Questa tecnica e' quella comunemente nota come tecnica di *simulazione*, con la quale il comportamento di un sistema e' riprodotto dal comportamento *dinamico* di un modello ottenuto per mezzo di un computer [2, 7]. I modelli usati per condurre esperimenti con la tecnica di simulazione sono chiamati modelli di simulazione. Essi sono normalmente liberi da quelle ipotesi limitative che sono necessarie per rendere possibile l'analisi matematica dei modelli analitici. Gli svantaggi principali della simulazione sono:

1. Non si possono ottenere relazioni funzionali di causa-effetto tra parametri e misure di prestazione.
2. Il costo computazionale di un esperimento di simulazione e' in genere alto.
3. I risultati della simulazione devono essere considerati come stime degli indici di prestazione del modello che si intende studiare.

Questi svantaggi devono spingere a dare precedenza all'uso dei modelli analitici. La tendenza corretta dovrebbe essere quella di analizzare il problema reale, in modo da scoprire e comprendere quali aspetti della realta' sono essenziali e quali marginali rispetto a quello che e' l'obiettivo dell'analisi. Evidentemente esistono numerosi casi in cui la simulazione rimane l'unica metodologia di analisi disponibile; in questi casi i "costi" della simulazione rappresentano il prezzo necessario per lo studio del comportamento del sistema.

## *2.2 Simulazione per eventi discreti*

La simulazione per eventi discreti indica la simulazione di modelli nei quali i cambiamenti di stato sono prevalentemente discontinui ed avvengono ad intervalli di tempo di durata arbitraria. Con questa tecnica si riproduce una possibile storia degli stati del modello. I concetti fondamentali che riguardano questo tipo di simulazione sono l'*evento* e l'attivita'. Un evento e' responsabile di un cambiamento istantaneo di stato

del sistema simulato, mentre un'attivita' rappresenta una operazione del sistema (o di un suo componente) la cui esecuzione richiede del tempo. Nel modello, il senso con cui viene usato il termine attivita' cambia. Il modello di simulazione, infatti, non e' tanto interessato al contenuto delle attivita' reali, quanto ai loro effetti sul cambiamento di stato del sistema simulato e agli istanti di tempo in cui si verificano gli eventi che ne segnano l'inizio e la fine.

Una struttura dati essenziale nell'implementazione di un simulatore e' il "calendario degli eventi" (*event list*). L'event list e' una lista in cui vengono inseriti, secondo un ordinamento cronologico, tutti gli eventi schedulati e non ancora presi in considerazione. Ogni nodo di questa lista e' costituito da un descrittore di evento (*event notice*) il quale contiene tutti gli attributi necessari per specificare le azioni che il simulatore deve intraprendere come conseguenza dell'essersi verificato quel determinato evento. I due attributi fondamentali sono evidentemente l'istante di tempo in cui tale evento si verifichera' e il tipo dell'evento stesso.

Il compito principale di un simulatore e' quello di scandire questo calendario e provvedere al richiamo delle opportune procedure dedicate alla gestione del cambiamento di stato determinato dal verificarsi di un certo tipo di evento.

Un altro concetto fondamentale nel campo della simulazione e' quello di *tempo*. In simulazione si possono individuare tre concetti differenti di tempo:

- *Tempo reale*: variabile continua che si riferisce al tempo del sistema che si sta simulando.
- *Tempo simulato*: variabile che assume valori discreti ad ogni cambiamento di stato e che riproduce il tempo reale nel modello costruito.
- *Tempo di esecuzione*: rappresenta il tempo impiegato dal calcolatore per terminare l'esperimento di simulazione ed e' dipendente dalla velocita' di elaborazione, dalla efficienza del programma simulatore e dalla complessita' del modello da studiare.

Il trascorrere del tempo all'interno del simulatore e' rappresentato da una variabile detta *clock* del simulatore. Il tempo di esecuzione e' dipendente dalla efficienza delle procedure che gestiscono il cambiamento di stato e non dall'avanzamento del tempo simulato, mentre le attivita' dei componenti del sistema simulato consumano tempo di simulazione ma non tempo di esecuzione. Esistono due distinte tecniche di avanzamento per il clock del simulatore:

- *Sincrono*: il clock viene avanzato ad istanti di tempo determinati, per esempio ogni N unita' di tempo, e il sistema e i cambiamenti di stato vengono esaminati ad ogni avanzamento del clock.
- *Asincrono*: il clock e' aggiornato ad ogni cambiamento di stato e quindi il sistema viene continuamente osservato.

Questi due approcci hanno diverse implicazioni. Con il primo vi e' il vantaggio di gestire un solo cambiamento di stato anche al verificarsi di piu' eventi tra due aggiornamenti successivi, ma e' necessario aggiornare il clock anche se non si e'

verificato nessun evento. Nell'approccio asincrono, invece, il numero di aggiornamenti e' dipendente dal numero di eventi e non c'e' possibilita' di lavoro a vuoto. Il bilancio tra vantaggi e svantaggi dei due metodi varia da esperimento a esperimento in base al numero di eventi registrati. In generale si adotta il metodo asincrono in quanto evita la gestione contemporanea di eventi distinti e genera un tempo di esecuzione minore, eccetto che nel caso di molti eventi simultanei.

Spesso i modelli usati in simulazione sono *stocastici*. Infatti, i parametri di ingresso usati sia nel modello che nel simulatore derivano da misurazioni effettuate in precedenza sul sistema reale. La natura stocastica di questi parametri dipende o dalla intrinseca aleatorieta' del sistema reale o dal fatto che ogni misura e' corredata da un errore di natura stocastica. Questo implica che durante la simulazione saremo costretti a generare istanze di variabili casuali. In realta', per completezza e' necessario distinguere due possibili modi per generare queste quantita'. In certi studi puo' essere interessante riprodurre la dinamica di un sistema reale facendo uso di tempi e di eventi misurati sul sistema reale e memorizzati in una cosiddetta traccia. In questi casi, i valori successivamente assunti dalle variabili, invece di essere generati, sono prelevati dalla traccia misurata. Un simulatore di questo tipo, detto *Trace Driven*, non ha natura stocastica, ma e' un programma deterministico il cui scopo e' spesso quello di permettere lo studio di un sistema reale analizzando un modello semplificato dello stesso. In altri casi, quando certi parametri di ingresso sono definiti per mezzo di distribuzioni di probabilita', la generazione di quantita' caratterizzate da tali distribuzione corrisponde alla generazione di istanze di variabili casuali cosi' distribuite. Simulatori che adottano questo metodo sono detti di tipo *Monte Carlo*.

Sono stati ideati ed impiegati diversi metodi per la generazione di sequenze di numeri casuali, ma noi siamo interessati solo a quelli ottenibili grazie ad un elaboratore elettronico. Bisogna tenere presente che l'uso di metodi matematici puo' solo produrre sequenze "pseudo casuali" che possono essere considerate o meno accettabili da un punto di vista statistico. Alla base di ogni studio di simulazione, vi e' il problema della stima dei parametri delle distribuzioni di probabilita' a cui si e' interessati sia nella fase di preparazione del modello che nella fase di valutazione dei risultati dell'esperimento di simulazione. Nella preparazione, partendo da osservazioni sul sistema reale, occorre inferire le caratteristiche delle distribuzioni usate nel modello come sorgenti di arrivi o dispositivi di servizio. Nella valutazione dei risultati, partendo invece da osservazioni sul sistema simulato, occorre inferire le caratteristiche stocastiche delle variabili per le quali si effettua lo studio. I processi di stima sono normalmente semplici nella fase preparatoria, in quanto si puo' facilmente assumere la stazionarieta' della popolazione osservata e l'indipendenza delle osservazioni, che sono le due condizioni su cui si basano i metodi della statistica classica. Purtroppo le stesse assunzioni non possono essere fatte nella fase di valutazione dei risultati, poiche' la sequenza di osservazioni accumulate durante il corso della simulazione per una data variabile risulta autocorrelata.

Un metodo per ottenere osservazioni *indipendenti* e *identicamente distribuite* e' quello di ripetere l'esperimento piu' volte ottenendo ogni volta un esperimento indipendente dai precedenti. Condizione necessaria e' che il processo stocastico degli stati del sistema simulato $\{S(t) \mid t \geq 0\}$ sia *rigenerativo*, cioe' tale che esista una sequenza di istanti

random in cui il processo ricomincia da uno stesso stato ed evolve con la stessa legge di probabilita'. Nella simulazione discreta esistono tre metodi per la generazione di esperimenti indipendenti:

1. *Metodo delle prove ripetute.*

   Questo metodo, chiamato anche *rigenerativo artificiale*, e' un modo per far rispettare, forzatamente, al processo di simulazione la proprieta' di ciclicita' su cui si basano i processi di valutazione statistica. Con questo metodo si eseguono p esperimenti usando ogni volta sequenze di numeri casuali diverse e facendo partire tutti gli esperimenti di simulazione da un medesimo stato scelto come stato caratteristico del sistema in equilibrio.

2. *Metodo del punto ciclico.*

   Conosciuto anche come metodo *rigenerativo naturale*, si basa su processi rigenerativi permettendo un passaggio piu' diretto all'analisi dei dati. In molti sistemi stocastici da simulare, e' possibile trovare raggruppamenti casuali di osservazioni che producono blocchi globalmente indipendenti e identicamente distribuiti. Questi blocchi si presentano in modo naturale sin dall'inizio, purche' esista uno stato nel quale il sistema ritorna ciclicamente e con certezza fino alla fine dei tempi, con un tempo medio tra due ritorni finito. In pratica un unico esperimento di simulazione viene naturalmente suddiviso in esperimenti indipendenti che iniziano e terminano con stesso stato del sistema. Purtroppo questo metodo e' utilizzabile solo nei casi in cui e' semplice individuare i punti di rigenerazioni e in cui questi si ripetono con una certa frequenza, per non compromettere il costo dell'esperimento stesso.

3. *Metodo batch.*

   L'idea e' quella di spezzare un unico giro di simulazione molto lungo in maniera arbitraria, ottenendo degli spezzoni (*batch*) che vengono assunti mutuamente indipendenti. Eseguendo questi troncamenti in maniera random non si e' affatto garantiti di ottenere esperimenti singoli con stato iniziale comune, ne' che lo stato iniziale di ogni esperimento non sia anomalo (cioe' richiedente un periodo di stabilizzazione che non viene invece considerato). La semplicita' di questo metodo deve essere pertanto pagata dal fatto di avere esperimenti correlati tra loro. Per minimizzare il grado di correlazione tra i diversi spezzoni si potrebbero considerare esperimenti molto lunghi, capaci cioe' di attenuare, grazie al grande numero di osservazioni fatte per ognuno di essi, la loro dipendenza statistica. Un criterio piu' elegante e sicuramente piu' efficiente e' quello di far precedere l'esperimento da giri di simulazione pilota in cui viene stabilita la lunghezza che rende minima la correlazione di batch distinti.

# 3 Sistemi distribuiti affidabili e misure di interesse

In questa sezione vengono presentate le caratteristiche principali della classe di sistemi fault tolerant da analizzare, le misure di interesse e i principali parametri in termini di modello dei guasti e dei failure, strategie di error processing, strategie di fault treatment e carico di input.

## *3.1 Caratteristiche generali*

L'architettura che consideriamo e' un sistema multiprocessore composto da $n$ unita' di elaborazione ($u_0$, ..., $u_{n-1}$) assunte atomiche ed identiche. L'architettura multiprocesore si presenta abbastanza generale da essere normalmente utilizzata in molti contesti, sia per semplice software applicativo sia per applicazioni altamente critiche. Il nostro principale interesse e' quello di organizzare un'architettura in grado di supportare applicazioni con requisiti di dependability. A tale scopo e' necessario includere nel sistema componenti responsabili: i) dell'allocazione delle richieste ai processori serventi, ii) dell'esecuzione dell'algoritmo di error processing e iii) dell'identificazione dei processori danneggiati da rimuovere. L'architettura logica del sistema e' mostrata in Figura 1.

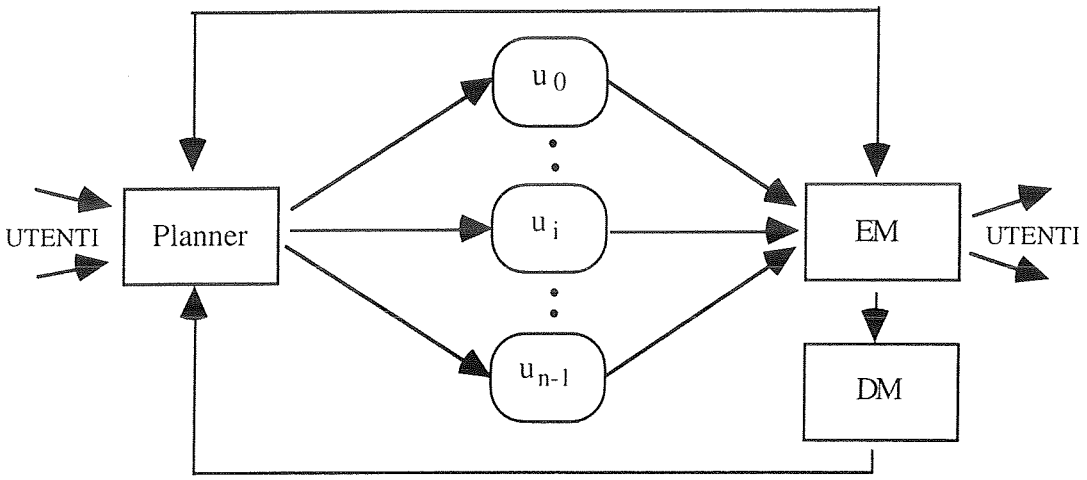



**Figura 1 - Architettura logica del sistema**

Su richiesta di servizio, il Planner seleziona un certo numero di processori (in ottemperanza alla strategia di error processing) e fornisce loro sia i dati di input sia l'applicazione software da eseguire. L'esecuzione corretta di uno stesso task con lo stesso input, su processori differenti, produce sempre lo stesso risultato. I risultati restituiti dai processori nell'esecuzione dello stesso task vengono collezionati dall'Error Management (EM) il quale si incarica, attraverso un apposito algoritmo di aggiudicazione, di selezionare e rilasciare il risultato che ritiene corretto. Qualora lo schema di error processing faccia uso di ridondanza sia spaziale che temporale [5, 12], l'operazione di aggiudicazione potrebbe richiedere l'esecuzione di piu' fasi, il che implica scambio di informazioni tra EM e Planner. Cio' e' dovuto al fatto che le esecuzioni delle copie aggiuntive dell'applicazione e' condizionale all'impossibilita' di aggiudicare un risultato da parte dell'EM. Consistentemente con la selezione del risultato corretto per ciascuna applicazione, EM rilascia a DM il suo giudizio sul comportamento dei processori coinvolti nell'espletazione di un servizio, informandolo cosi' dei processori sospetti. Sulla base di tali informazioni, DM identifica i processori danneggiati e comunica il suo responso al Planner il quale si incarica di bloccare l'invio di richieste di servizio alle suddette unita'. Cosi' facendo il numero di processori attivi

tende progressivamente a diminuire, finche' non raggiunge una soglia minima (predefinita) al di sotto della quale il sistema cessa di funzionare. Le strategie di hardware fault tolerance considerate sono basate su un insieme integrato di tecniche per l'error processing, per il fault treatment e per la gestione delle risorse.

## *3.2 Il modello dei guasti e dei failure*

Data l'assunzione che tutti i link dell'architettura e i componenti Planner, EM e DM siano affidabili, gli unici guasti contemplati sono quelli di tipo fisico (operazionali) che coinvolgono le unita` serventi. Essi vengono classificati in:

- *permanenti,*
- *intermittenti,*
- *transitori.*

Un guasto in una unita` puo` provocare il failure dei servizi da questa effettuati. In particolare, se il guasto e` permanente (hard) ogni servizio futuro su quell'unita` sara` failure; se intermittente, da quel momento in poi solo una percentuale predefinita dei servizi eseguiti su quel processore sara` un failure; se transitorio solo il servizio durante il quale si e' verificato il guasto sara' un failure.

Ogni unita` si guasta indipendentemente (in senso probabilistico) dalle altre e la con stessa probabilita'. Si assume inoltre che vi siano solo failure di valore; i failure di timing (o di omissione) vengono individuati da un timer affidabile al momento in cui la "deadline" (cioe` il tempo massimo entro cui un risultato corretto deve essere fornito) viene raggiunta e vengono ricondotti a fallimenti di valore.

## *3.3 Schemi di error processing*

Gli schemi di error processing che consideriamo nel sistema sono quelli basati su replicazione (senza diversita`, in cui ogni replica del task e` eseguita su di un processore differente) e votazione e comunque riconducibili allo schema SCOP [5]. Casi particolari sono gli schemi di error processing tradizionali TMR, NMR [1, 10], RB [12] e gli schemi basati su uso dinamico di ridondanza spaziale e temporale come SCOP2+1 e SCOP2+2 [3, 4, 8]. Per questi schemi i risultati dell'esecuzione di un task possono essere:

i) successo, cioe` la fornitura di un risultato corretto,

ii) errore "detected" (scoperto o dal confronto di risultati ridondanti o da un timer watchdog),

iii) errore "undetected" (fornitura di un risultato erroneo).

## *3.4 Tecniche di fault treatment*

La tecnica di fault treatment adottata consente una gestione flessibile ed efficiente dei processori senza eccessiva degradazione di affidabilita`, in presenza di guasti permanenti e/o transienti (e/o eventualmente intermittenti) [4]. Essa cerca un

compromesso fra il costo in termini di prestazioni complessive del sistema nel caso di eliminazione immediata (o sospensione per un periodo di riparazione) delle unità in cui si verifica un failure (magari a causa di un guasto temporaneo) ed il costo in termini di affidabilità associato a mantenere nel sistema unità non piu` funzionanti. E` basata sulla raccolta di informazioni sul corretto funzionamento delle unità, informazioni che, se in linea di principio possono essere ricavate in vari modi, in pratica per la classe di sistemi considerata sono un risultato immediato della aggiudicazione. Ogni unità viene riutilizzata e i suoi eventuali failure mascherati, finche` non viene ritenuta guasta permanentemente. Le due varianti della strategia di diagnosi che noi consideriamo sono:

1) alfa strategia semplice: in cui per ogni processore vi e` una funzione ALFA che viene incrementata ogni volta che viene rilevato il (presunto) failure del processore, e decrementata opportunamente ad ogni rilevazione di successo, non appena ALFA supera un valore di soglia il processore viene considerato guasto permanentemente;

2) alfa strategia con sospensione dei processori sospetti: a differenza della precedente esistono due valori di soglia, uno inferiore per sospendere un processore che quindi viene considerato sospetto ed uno superiore per considerare il processore guasto permanentemente; il processore sospetto viene utilizzato per l'esecuzione di repliche aggiuntive rispetto a quelle eseguite per ogni task; i risultati aggiuntivi cosi` ottenuti vengono utilizzati non ai fini dell'aggiudicazione del risultato da inviare all'utente per ciascun task, ma solo per raccogliere informazioni diagnostiche sulla funzione ALFA. Se il valore di ALFA scende al di sotto della soglia di sospensione, prima di superare la soglia di rimozione, esso viene di nuovo riutilizzato per la normale computazione, perche` non piu` sospetto. Questa strategia puo` essere considerata come un caso piu` generale della precedente.

I processori ritenuti guasti vengono *riparati*, *sostituiti*, oppure *rimossi* a seconda della classe di applicazioni che si intende analizzare. In caso di riparazione il processore e' inutilizzato per tutta la durata della riparazione; questo periodo di inutilizzazione risulta inferiore nel caso di sostituzione del processore con uno nuovo. Al momento in cui un servente viene considerato guasto (o sospeso) i risultati delle singole repliche da esso prodotti e pendenti (cioe` quelli non ancora confrontati con gli altri per l'aggiudicazione del risultato del task) possono essere trattati in due modi: 1) sono considerati non affidabili e di conseguenza vengono riottenuti mediante riesecuzione della replica dello stesso task su un altro servente diverso da quelli gia` usati per eseguire lo stesso task, o 2) vengono utilizzati come se non fossero per niente sospetti, confidando nella efficacia della strategia di error processing. In questo secondo caso, si riduce l'affidabilta` del sistema rispetto al caso di riesecuzione.

## *3.5 Gestione dinamica dei processori*

La gestione dei processori e` dinamica e flessibile. La scelta delle unita` da utilizzare per effettuare un servizio viene fatta dinamicamente a tempo di esecuzione in modo da evitare ritardi dovuti alla sincronizzazione. Per ogni task da eseguire non si aspetta che tutte le unita` necessarie per l'esecuzione delle repliche siano libere, eliminando cosi`

l'intervallo di tempo, tipico del caso sincrono, in cui unita` libere sono mantenute inattive in attesa che si liberino tutte quelle necessarie.

## *3.6 Carico di input del sistema e tipi di task*

Per quanto riguarda il carico di input a cui il sistema e` sottoposto si considerano i 2 seguenti casi:

1) il sistema lavora con coda di input dall'utente sempre piena (carico infinito): non appena si libera un processore esiste sempre una richiesta di servizio per quel processore;
2) il sistema lavora con un carico di input limitato.

I task eseguiti dal sistema possono essere di tipi differenti. Ad ogni tipo e` associato:

1) il tempo di servizio di ciascun task: sono considerate distribuzioni dei tempi uniformi ed esponenziali (l'implementazione di ulteriori distribuzioni deve essere fattibile con il minimo sforzo di programmazione);
2) il tipo del risultato (cioe` della cardinalita` dell'insieme dei possibili risultati): sono stati considerati Booleani, Caratteri, Interi, e di tipo utente;
3) il tempo in cui le richieste di esecuzione dei task arrivano al sistema (stesse possibili distribuzioni dei tempi di servizio).

Le repliche di ciascun task possono avere lo stesso tempo di esecuzione o tempi differenti.

## *3.7 Caratteristiche di real-time*

Per quanto riguarda le caratteristiche di real-time del sistema esse non sono state considerate in maniera esplicita. La flessibilita` e modularita` del simulatore devono poter consentire l'introduzione di tali caratteristiche con un minimo sforzo di programmazione ulteriore.

## *3.8 Misure di interesse*

Le varie misure di dependability, performance e performability [6, 9, 11, 14] che interessa ricavare sono:

- il tempo medio in cui si verifica una combinazione o il primo dei seguenti eventi: esaurimento dei processori, detected failure, undetected failure;
- la probabilita` che in una missione di durata limitata si verifichi una combinazione o il primo dei seguenti eventi: esaurimento dei processori, detected failure, undetected failure;
- per missioni che terminano all'esaurimento del numero di processori, il tempo medio, e/o il numero medio di task corretti all'istante, in cui si verifica il primo

detected o undetected failure, condizionati dal fatto che tale evento si e` verificato;

- il throughput medio (numero medio di servizi per unita` di tempo) per missione;
- numero medio di task corretti accumulati nella missione (la missione termina al verificarsi di una combinazione o del primo dei seguenti eventi: esaurimento dei processori, detected failure, undetected failure, esaurimento del tempo di missione se limitato);
- numero medio di detected o undetected failure accumulati nella missione;
- "reward" medio accumulato nella missione secondo differenti possibili strutture di "reward" [11, 14];
- la discrepanza tra quanti processori la strategia di fault diagnosis ha individuato come guasti e quanti sono realmente guasti, cioe`: il numero medio di processori ritenuti guasti permanentemente e il numero medio di processori ritenuti guasti permanentemente che lo sono effettivamente.

Altre misure di interesse (ad es. per analisi steady-state) devono poter essere facilmente calcolate con piccole integrazioni del software del simulatore.

# 4 Modello e software del simulatore

In questa sezione descriviamo il modello di simulazione per la classe di sistemi della Sezione 3 e il simulatore orientato ad eventi, implementato in linguaggio di programmazione C. Il software che implementa il simulatore e` strutturato in moduli. Descriviamo in dettaglio la struttura del simulatore e le principali funzionalita'.

## *4.1 Modello del sistema e struttura principale del software*

Il modello a rete di code che il simulatore ad eventi risolve e` mostrato in Figura 2. A differenza dell'architettura logica del sistema mostrata in Figura 1, qui vengono riportati le code Q_IN e Q_RIESEC, i componenti che richiedono tempo di esecuzione (Planner, EM e DM) e quelli che consumano solo tempo simulato (processori).

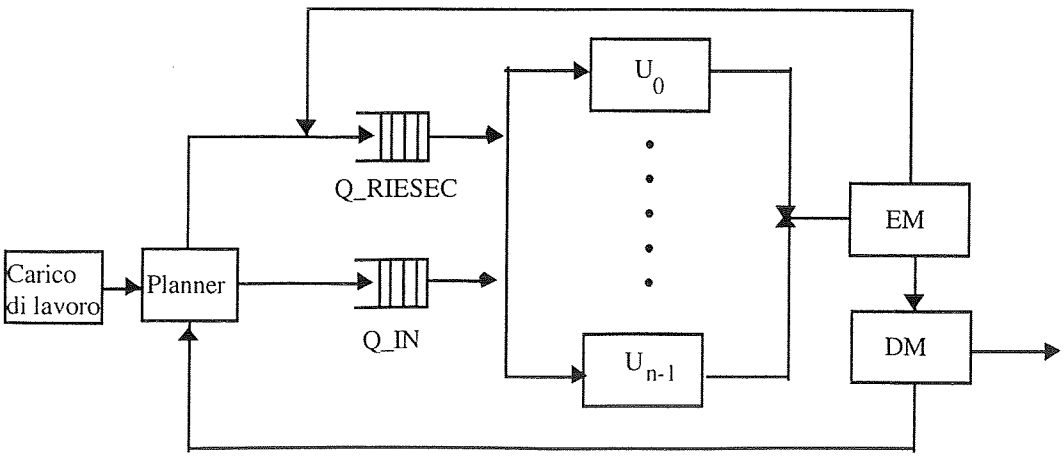



**Figura 2 - Modello a reti di code del sistema**

Il software del simulatore e` strutturato in moduli, come mostrato in Figura 3. Alcuni costituiscono il nucleo del simulatore, altri caratterizzano il sistema da simulare.

Nel nucleo del simulatore sono implementate le funzioni di:

1) inizializzazione del simulatore, in INIT_EVFUNZG.C,
2) gestione della EVENT_LIST e l'avanzamento del tempo simulato, in MAIN.C,
3) gestione degli eventi considerati, in EVENT_FUNZ_GEST.C,
4) accumulo delle statistiche e valutazione delle misure, in EVAL.C,
5) gestione di liste e code, in LIST_FUNZ_GEST.C,
6) generazione dei numeri pseudo-casuali, in GEN.C.

Nel modulo ADJUDICATION.C sono implementate differenti strategie di aggiudicazione del risultato. In ALFA_DIAGNOSE.C la strategia di diagnosi. In FAULT_MODEL.C le funzioni per implementare il modello dei guasti adottato. In REM_TREATM.C, incluso in ALFA_DIAGNOSE.C (e SUSPALFA_DIAGN.C), le strategie di trattamento dei risultati pendenti prodotti dal processore ritenuto guasto e le eventuali strategie di riparazione o sostituzione del processore rimosso. In REPL_TIME.C vengono gestiti i tempi di esecuzione delle repliche.

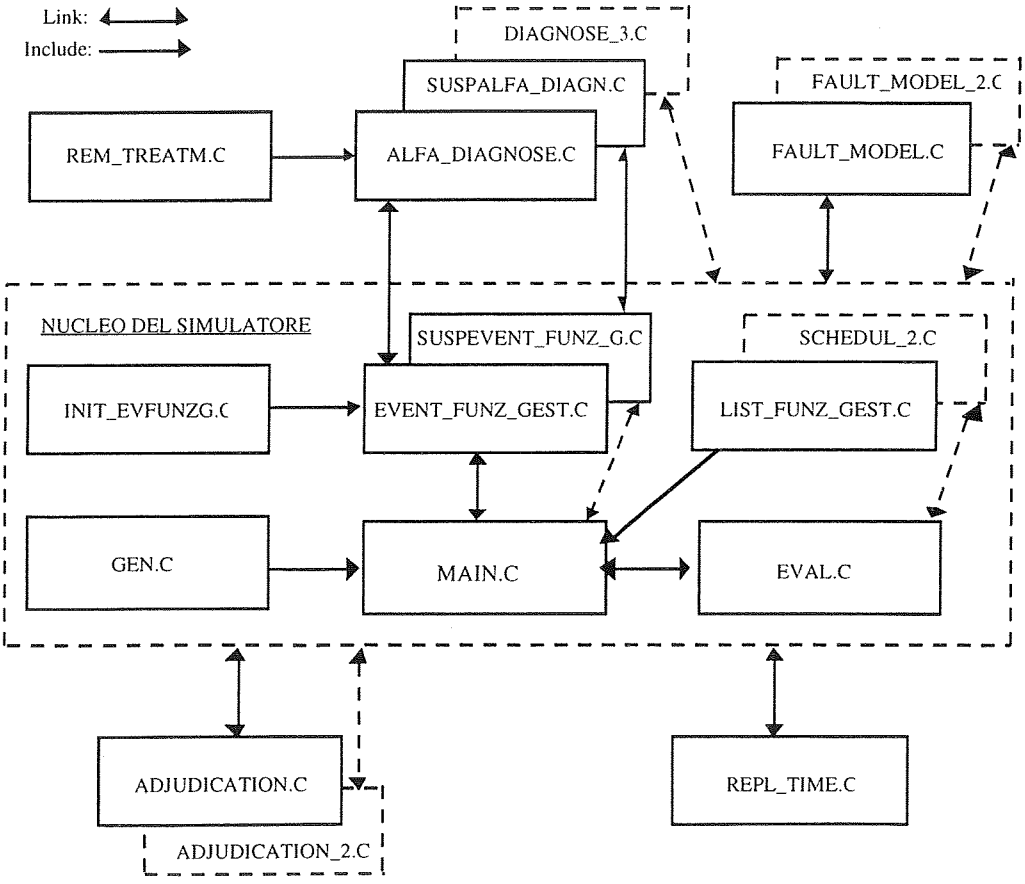



**Figura 3 - Struttura del software del simulatore**

Ogni nodo del grafo di Figura 3 puo` avere piu` moduli (scatole sovrapposte), ciascuno dei quali rappresenta differenti soluzioni adottate per la funzionalita` del sistema rappresentata da quel nodo. Per molte delle funzionalita` del sistema infatti possono essere adottate differenti soluzioni e strategie a seconda della classe di applicazione di interesse. Alcune di queste differenti soluzioni, sono state esplicitamente implementate nel software del simulatore, consentendo all'utente a tempo di esecuzione di scegliere tra piu` opzioni differenti.

Al contrario le due possibili soluzioni per la ALFA-strategia di diagnosi proposte nel Paragrafo 3.4 sono state implementate con moduli differenti che vanno linkati separatamente. Se si vuole simulare la ALFA-strategia di diagnosi con una soglia semplice, i moduli ".C" che in Figura 3 sono collegati con archi continui vanno compilati e poi linkati tra di loro. Se invece si vuole simulare la ALFA-strategia di diagnosi con doppia soglia e processori sospesi, occorre sostituire EVENT_FUNZ_GEST.C e ALFA_DIAGNOSE.C con rispettivamente SUSPEVENT_FUNZ_G.C e SUSPALFA_DIAGN.C (Figura 3).

I moduli rappresentati da scatole tratteggiate in Figura 3 sono un esempio di moduli che possono essere sviluppati per implementare soluzioni differenti da quelle considerate e che vanno quindi linkati (o inclusi) con tutti gli altri al posto dei corrispondenti rappresentati con scatole continue. L'implementazione di questi moduli ovviamente va fatta in maniera consistente con il resto del software. Il simulatore per come e` strutturato è quindi sufficientemente modulare e flessibile.

## *4.2 Eventi rari: i guasti e i failure*

Allo scopo di cercare di ridurre i tempi di esecuzione dovuti al problema degli eventi rari, il simulatore e' stato corredato di una tecnica particolare basata sulla previsione dell'occorrenza dei guasti e sulla simulazione dettagliata del sistema solo ad intervalli temporali (quelli che comprendono il verificarsi dei guasti e la terminazione del loro effetto). Finche' dura l'effetto di uno o piu' guasti, l'esecuzione dei vari task replicati sui vari processori viene simulata dettagliatamente in modo da riuscire a rappresentare tutti i fenomeni connessi alla presenza di guasti permanenti (latenti) nel sistema. Cioe', ad ogni esecuzione di un task su un processore viene generato esplicitamente un evento che rappresenta il possibile verificarsi di un guasto, di un failure o di un successo. Dal momento in cui si esaurisce l'effetto dei guasti (quando ad esempio il processore coinvolto viene rimosso) fino al verificarsi del successivo guasto il sistema viene simulato con un minor livello di dettaglio. Infatti, sotto le ipotesi di guasto e failure viste nel Paragrafo 3.2, la sequenza di guasti cui il sistema e' affetto puo' essere adeguatamente rappresentata attraverso l'uso di *generatori poissoniani di guasto* associati alle singole unita' serventi. Al verificarsi di un guasto al tempo $t$ dell'unita' i, esso verra' classificato in transiente, intermittente o permanente in base a percentuali predefinite e lo stato del processore modificato opportunamente. Data l'indipendenza probabilistica del funzionamento dei processori, e' possibile combinare i generatori di guasto in un unico processo globale mantenente le stesse caratteristiche dei processi componenti. Cosi' se N e' il numero di unita' serventi di cui e' composto il sistema e $\lambda_f$ e' il *failure rate* associato ad ognuno di esse, allora il generatore globale produrra'

guasti ad intervalli distribuiti esponenzialmente con tasso N * $\lambda_f$. In questo modo, l'associazione <guasto, processore> verra' fatta al verificarsi del guasto estraendo un'unita' dalle N complessive. La correttezza del meccanismo e' garantita dalle caratteristiche e dalle tecniche di composizione e scomposizione tipiche dei *Processi Poissoniani*.In questo modo si riduce il numero di eventi che e' necessario generare tra due successivi guasti.

Tuttavia, questa tecnica risolve solo parzialmente il problema degli eventi rari. Tale problema, di non facile risoluzione, può essere però raggirato se si orienta l'analisi verso sistemi non ultra-dependable (dove i guasti si presentano meno raramente) e se le misure da stimare sono scelte con ragionevolezza rispetto ai parametri che caratterizzano il sistema.

## *4.3 Modello degli schemi di error processing*

Gli schemi di error processing sono classificati assegnando alla funzionalita' che li rappresenta una condizione di fornitura del risultato, il massimo numero di repliche eseguibili per ciascun task e il massimo numero di fasi possibili (per rappresentare eventuali vincoli di real-time).

Nel modello sono state realizzate tre condizioni di fornitura del risultato basate sul confronto degli output dei serventi:

1) a maggioranza assoluta, in cui per ogni task l'aggiudicatore fornisce in output, se possibile, il risultato prodotto da un numero di processori che e` la maggioranza assoluta di quelli usati per eseguire tutte le repliche di quel task, (si ricorda che due processori hanno prodotto lo stesso risultato, se i due risultati sono uguali, oppure sono tra loro consistenti),
2) a maggioranza relativa, in cui per ogni task l'aggiudicatore fornisce in output, se possibile, il risultato prodotto da un numero di processori che e` la maggioranza relativa di quelli usati per eseguire tutte le repliche di quel task,
3) per massimo numero di failure possibili [5], in cui l'aggiudicatore sceglie il risultato corretto basandosi sull'ipotesi che per ogni task al piu` si puo` verificare un certo numero di risultati errati.

Esse consentono implicitamente di rappresentare un vasto range di schemi di error processing come previsto nel sistema.

## *4.4 Politiche di schedulazione dei task e di allocazione dei processori*

L'allocazione dei processori e' dinamica e libera da sincronismi. I task in attesa di esecuzione (nessuna replica e` stata eseguita) o parzialmente eseguiti (qualche replica del task e` gia` in esecuzione, ma non tutte), sono accodati nella coda dei task in arrivo dall'utente Q_IN, gestita mediante una politica FIFO. Le richieste di esecuzione di repliche aggiuntive di un task (nel caso lo schema di error processing comporti piu` fasi di esecuzione) sono inserite nella Q_RIESEC, anch'essa gestita mediante una politica

FIFO. A quest'ultima viene data priorita` maggiore rispetto alla prima, privilegiando in tal modo i task che per primi hanno cominciato la computazione Nel caso di riesecuzione, la replica di un task che ha prodotto un risultato pendente (probabilmente errato) viene immediatamente rieseguita su un processore differente da quelli gia` usati per lo stesso task, se ve ne sono liberi. Viceversa, la richiesta viene messa in *testa* nella coda di riesecuzione, se la fase a cui e` giunto il task e` superiore alla prima, altrimenti in *testa* alla coda dei task in arrivo dall'utente.

## *4.5 Funzionalita' principali*

In questo paragrafo vengono richiamate brevemente solo alcune delle principali funzionalita' del simulatore. Le funzioni principali del modulo main.c sono:

1) main():

   che si occupa dei richiami delle funzioni di acquisizione degli input e dell'inizializzazione dello stato del simulatore, e dell'avanzamento controllato della simulazione in batch;

2) cloc():

   per l'avanzamento del tempo simulato, l'estrazione dell'evento corrente dalla lista degli eventi schedulati e l'attivazione delle funzioni di gestione degli eventi.

Le funzioni principali di EVENT_FUNZ_GEST.C, che gestiscono i cambiamenti di stati del simulatore al verificarsi degli eventi considerati, sono:

1) end_phase(ind_proc_curr,j_rif):

   riceve in input l'indice del processore corrente che si e` appena liberato e il riferimento alle informazioni associate al task; modifica lo stato del sistema perche` si e` liberato un processore e la fase corrente e` terminata; si puo` eseguire una nuova fase oppure la fase appena eseguita e` l'ultima, nel qual caso viene effettuata l'aggiudicazione del risultato e la diagnosi dei processori guasti; alla diagnosi possono seguire operazioni di rimozione di processori considerati guasti e di trattamento dei risultati pendenti; il processore liberato puo` essere riutilizzato, se non e` stato nel frattempo considerato da rimuovere;

2) next_version_satur(ind_proc_curr), nel caso di input a saturazione:

   non appena si libera un processore vi e` un task pronto per l'esecuzione; modifica lo stato del sistema perche` si e` liberato un processore e la replica del task appena eseguito sul processore liberato non e` l'ultima; il processore liberato viene immediatamente riutilizzato, se nel frattempo non viene ritenuto guasto permanente, o da task in attesa di poter eseguire repliche su quel processore, oppure da un nuovo task di input, scelto tra i possibili tipi di task con la probabilita` indicata dall'utente;

4) job_arrive_satur(), nel caso di input a saturazione:

   modifica lo stato del sistema all'arrivo di un task dall'esterno e restituisce il riferimento al task creato;

3) next_version_notsatur(ind_proc_curr), nel caso di carico di input limitato:

modifica lo stato del sistema perche` si e` liberato un processore e la versione del job appena eseguito sul processore liberato non e` l'ultima; il processore liberato puo` essere riutilizzato immediatamente, se ritenuto non guasto permanente, solo se vi e` almeno un task in attesa di poter eseguire una replica su quel processore, altrimenti esso verra` utilizzato non appena arriva un nuovo task di input;

5) job_arrive_notsatur(j_type), nel caso di carico di input limitato:

   modifica lo stato del sistema all'arrivo di un task dall'esterno e restituisce il riferimento al task creato;

6) go_to_fault():

   gestisce il salto temporale al guasto successivo;

7) fault_occorrence(ind_proc):

   funzione per la gestione dell'evento fault_occorrence.

Ulteriori dettagli possono essere ricavati direttamente dal codice delle funzioni in Appendice, dove per brevita' vengono riportate solo le funzioni go_to_fault e fault_occorrence.

## Conclusioni

In questo lavoro sono state descritte le caratteristiche principali un simulatore ad hoc per l'analisi di strategie di sistemi multiprocessore dependable, sviluppato facendo uso di tecniche di simulazione stocastica orientata agli eventi discreti. Il codice, implementato in linguaggio di programmazione C, rispetta i requisiti di modularità e semplicità, necessari per consentire facili manovre per eventuali estensioni o modifiche. Una caratteristica del simulatore è rappresentata dalla flessibilità con cui è possibile condurre uno studio. Esso permette l'analisi dettagliata di dependability e performance al variare della strategia di fault tolerance adottata, delle caratteristiche fisiche del sistema, del carico di lavoro e delle caratteristiche che definiscono le classi dei guasti. Gli schemi di error processing consentiti sono quelli basati sulla ridondanza temporale e/o spaziale, integrati eventualmente a meccanismi di diagnosi basati sulla ALFA-strategia. Allo scopo di ridurre i tempi di esecuzione, il simulatore e' stato corredato di una tecnica particolare. Essa si basa sulla previsione dell'occorrenza dei guasti e sulla simulazione dettagliata del sistema ad intervalli temporali (quelli che comprendono il verificarsi dei guasti e la terminazione del loro effetto). Tuttavia, questa tecnica risolve solo parzialmente il problema degli eventi rari. Tale problema, di non facile risoluzione, può essere però raggirato se si orienta l'analisi verso sistemi non ultra-dependable (dove i guasti si presentano meno raramente) e se le misure da stimare sono scelte con ragionevolezza rispetto ai parametri che caratterizzano il sistema.

## Riferimenti

[1] A. Avizienis and L. Chen, "On the Implementation of N-Version Programming for Software Fault Tolerance During Program Execution," in Proc. Proc. 1st Int. Conf. on Computing Software Applications (COMPASAC-77), 1977, pp. 149-155.

[2] G. Balbo, "Appunti per il Corso di SIMULAZIONE," Torino, Cooperativa Libraria Universitaria, 1981.

[3] A. Bondavalli, S. Chiaradonna and F. Di Giandomenico, "Efficient Fault Tolerance: an Approach to Deal with Transient Faults in Multiprocessor Architectures," in Proc. ICPADS'94 - International Conference on Parallel and Distributed Systems, Hsinchu, Taiwan, ROC, 1994, pp. 354-359.

[4] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico and L. Strigini, "Rational Design of Multiple-Redundant Systems: Adjudication and Fault Treatment," in "PDCS The Book", B. Randell, J. C. Laprie, H. Kopetz and B. Littlewood Ed., In publication, 1995, pp. 141-154.

[5] A. Bondavalli, F. Di Giandomenico and J. Xu, "A Cost-Effective and Flexible Scheme for Software Fault Tolerance," Journal of Computer Systems Science and Engineering, Vol. 8, pp. 234-244, 1993.

[6] A. Grnarov, J. Arlat and A. Avizienis, "On the Performance of Software Fault-Tolerance Strategies," in Proc. Proc. 10th Int. Symp. on Fault-Tolerant Computing (FTCS-10), Kyoto, Japan, 1980, pp. 251-253.

[7] G. Iazeolla, "Introduzione alla Simulazione Discreta," Torino, Boringhieri, 1978.

[8] IEEE-TR, "Special Issue on Fault-Tolerant Sofware," IEEE Transactions on Reliability, Vol. R-42, pp. 177-258, 1993.

[9] J. C. Laprie, "Dependability: a Unifying Concept for Reliable Computing and Fault Tolerance," in "Dependability of Resilient Computers", T. Anderson Ed., BSP Professional Books, 1989, pp. 1-28.

[10] J. C. Laprie, J. Arlat, C. Beounes and K. Kanoun, "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures," IEEE Computer, Vol. 23, pp. 39-51, 1990.

[11] J. F. Meyer, "On Evaluating the Performability of Degradable Computing Systems," IEEE Transactions on Computers, Vol. C-29, pp. 720-731, 1980.

[12] B. Randell, "System Structure for Software Fault Tolerance," IEEE Transactions on Software Engineering, Vol. SE-1, pp. 220-232, 1975.

[13] D. P. Siewiorek and R. S. Swarz, "Reliable Computer System - Design and Evaluation," Digital Press, 1992.

[14] A. T. Tai, A. Avizienis and J. F. Meyer, "Evaluation of Fault Tolerant Software: a Performability Modeling Approach," in "Dependable Computing for Critical Applications 3", C. E. Landwehr, B. Randell and L. Simoncini Ed., Springer-Verlag, 1993, pp. 113-135.

[15] A. T. Tai, A. Avizienis and J. F. Meyer, "Performability Enhancement of Fault-Tolerant Software," IEEE Transactions on Reliability, Special Issue on Fault-Tolerant Sofware, Vol. R-42, pp. 227-237, 1993.

[16] K. S. Trivedi, "Probability & Statistics with Reliability, Queuing, and Computer Science Applications," London, Prentice-Hall, 1982.

## Appendice

In questa Appendice, per brevita', viene riportato solo il codice che implementa le principali funzionalita' del paragrafo 4.2.

```
/*****************************************************************/
/*    FUNZIONE DI SALTO TEMPORALE AL GUASTO SUCCESSIVO           */
/*                                                               */
/* Il salto avviene solo se l`evento "fault_occorrence"          */
/* attualmente e` l`ultimo elemento dell'event_list              */
/* o se il successivo  e` un evento di fine missione             */
/*****************************************************************/

void go_to_fault()
{
 double dump;
 PTR_EVNOTICE rif_ev;
 int b=FALSE;
```

```
 int i;
 int exp;
 double incr_alfa=1;

 if (event->next==Null) b=TRUE;
 else if ((event->next)->type==END_MISSION) b=TRUE;
 if (b==TRUE)
      {
       exp=event->sched_time-time/mean_service;
       if (exp>0)
           for (i=1;i<=exp;i++) incr_alfa*=K;
       for (i=0;i<N_PROC;i++)
/* aggiorna il parametro alfa di ogni processore dato che esso */
/* ha funzionato correttamente per "exp" esecuzioni di repliche */
           {
            if (proc[i].perm_fault_proc==0)
                {
                 proc[i].alfa=proc[i].alfa*incr_alfa;
                }
           }
       dump=event->sched_time-event_list->sched_time;

/* riordina l'event_list in modo da effetuare un salto temporale */
/* Cio` vuol dire che la sottolista di eventi che precede l'evento */
/* "fault_occorrence" viene spostata al di la di questo aggiornando*/
/* il campo "sched_time" delle event_notice che la compongono */

       while (event_list->type!=END_MISSION &&
              event_list->type!=FAIL) {
                                  rif_ev=event_extract(&event_list);
                           rif_ev->sched_time+=dump;
                           if (event->sched_time>rif_ev->sched_time)
                             rif_ev->sched_time=event->sched_time;
                             event_list=schedula(event_list,rif_ev->sched_time,rif_ev);
                                  }

      }
}


/*****************************************************************/
/*    FUNZIONE PER LA GESTIONE DELL'EVENTO FAULT_OCCORRENCE      */
/*                                                               */
/* Al verificarsi dell'evento si stabilisce la natura del guasto */
/* che e` occorso nel processore ind_proc.                       */
/* Lo stato del processore viene aggiornato, la variabile        */
/* "attesa" incrementata di 1 dato che vi e` un processore       */
/* guasto  che dovra` essere diagnosticato. Si generano          */
/* quindi l'indice  del processore prossimo a guastarsi e il     */
/* tempo al guasto successivo, e si rischedula l'evento          */
/*****************************************************************/

void fault_occorrence(ind_proc)
 int ind_proc;
 {
  double casual;
  double par2=0.0;
  double par3=0.0;
  double new_time_sched=time;
  int b=FALSE;
  int new_ind_proc;
  int i=0;

  if (proc[ind_proc].perm_fault_proc!=TRUE &&
       proc[ind_proc].remov_fault_proc!=TRUE)
     {
       casual=randomnumber(&seed_fail);
       if (casual<perm_fault_prob)
                 {
```

```
                    proc[ind_proc].perm_fault_proc=TRUE;
                    fault_processor++;
                    proc[ind_proc].time_fault_proc=time;
                    if (proc[ind_proc].attesa==0)
                     {
                      proc[ind_proc].attesa=1;
                      attesa++;
                     }
                   }
          else if (casual<intermitt_fault_prob)
                    {
                     proc[ind_proc].perm_fault_proc=TRUE;
                     proc[ind_proc].intermitt_fault_proc=TRUE;
                     fault_processor++;
                     proc[ind_proc].time_fault_proc=time;
                     if (proc[ind_proc].attesa==0)
                      {
                       proc[ind_proc].attesa=1;
                       attesa++;
                      }
                    /* genera il tipo del guasto intermittente */
                    casual=randomnumber(&seed_fail);
                    while (casual>type_intermitt[i].prob) i++;
                    proc[ind_proc].intermitt_fault_freq=type_intermitt[i].freq;
                   }

          else {
                proc[ind_proc].trans_fault_proc=TRUE;
                proc[ind_proc].time_trans_proc=time;
                if (proc[ind_proc].attesa==0)
                        {
                         proc[ind_proc].attesa=1;
                         attesa++;
                        }
                }
       }
    else b=TRUE;

/* Se vi e` nel sistema qualche processore funzionante, genera */
/* 'indice genera l'indice del processore soggetto al prossimo */
/* guasto e l'istante in cui questo avviene */
 if ( N_PROC-remproc> fault_processor-permfprocrem)
  {
 new_ind_proc=uniform(&seed_proc_guasto,0,N_PROC,par3);
 new_time_sched+=exponential(&seed_guasto,fail_rate,par2,par3);
 while (proc[new_ind_proc].remov_fault_proc==TRUE ||
        proc[new_ind_proc].perm_fault_proc==TRUE)
         {
          new_ind_proc=uniform(&seed_proc_guasto,0,N_PROC,par3);
          new_time_sched+=exponential(&seed_guasto,fail_rate,par2,par3);
         }

 rif_event_curr->ind_free_proc=new_ind_proc;
 rif_event_curr->rif_job=Null;
 rif_event_curr->sched_time=new_time_sched;
 event_list=schedula1(event_list,rif_event_curr->sched_time,rif_event_curr);
   }

/* Se il verificarsi del guasto non ha alcun effetto (il processore*/
/* "ind_proc" e' gia' guasto - b==TRUE), se nessun processore */
/* guasto e` in attesa di essere diagnosticato ( attesa==0 ), se */
/* non si aspetta la scomparsa dell'effetto di un guasto che gia` */
/* ha prodotto errori (cioe` se non si sta` simulando per passi */
/* - go_step==FALSE) allora si puo` richiedere un salto nel tempo */
/* all'istante del successivo guasto */
 if (b==TRUE && go_step==FALSE && attesa==0) go_to_fault();
 }
```