Sistemi affidabili ed in tempo reale

Bruno Ciciani, Francesco Quaglia

1. Introduzione

Comunemente, un sistema viene denominato affidabile se esso è in grado di soddisfare le specifiche di progetto durante la sua utilizzazione. Sistemi affidabili vengono utilizzati in contesti in cui è necessario garantire una serie di attributi quali, per esempio, la sicurezza o la disponibilità di funzionamento, per cui, negli ultimi tempi, il termine *affidabilità* è stato sostituito da una sua generalizzazione identificata in inglese con il termine *dependability*, che in italiano corrisponde a: *fiducia nel corretto funzionamento del sistema*.

La progettazione e l'implementazione di sistemi che abbiano la proprietà di essere "dependable" necessita di apposite metodologie per l'individuazione delle possibili cause di malfunzionamento, denominate comunemente impedimenti, ed anche di appropriate tecniche per rimuovere o almeno limitare gli effetti di queste cause. Di conseguenza, affrontare il problema della dependability implica automaticamente avere una visione chiara di quali possano essere gli impedimenti e le tecniche per evitarne le conseguenze. I sistemi che utilizzano tali tecniche vengono denominati *tolleranti i guasti*.

In questa Sezione verrà presentata la terminologia di base per l'identificazione e la descrizione degli impedimenti, verranno elencati e discussi in maniera sistematica tutti gli attributi della dependability, verranno descritte le principali tecniche utilizzate per affrontare gli impedimenti e garantire quindi gli attributi, ed infine verranno mostrati i principali metodi di valutazione di sistemi dependable.

Prima di questa panoramica, verranno introdotte e descritte le classi di applicazioni che necessitano dei requisiti della dependability. La parte finale della Sezione sarà poi dedicata alla descrizione di una classe particolare di sistemi dependable, ovvero la classe dei sistemi in tempo reale, che oltre a possedere gli attributi classici, devono anche rispondere a vincoli di natura strettamente temporale dipendenti dal particolare contesto applicativo.

2. Applicazioni con requisiti di dependability

Esistono quattro categorie fondamentali di applicazioni che necessitano di essere supportate da sistemi che abbiano la proprietà di essere dependable. Ciascuna di esse verrà brevemente descritta in questo paragrafo.

2.1. Applicazioni di lunga durata

Sono comunemente denominate applicazioni di lunga durata tutte quelle per le quali o è richiesta una grande quantità di tempo di calcolo, al fine di ottenere i risultati desiderati, oppure è richiesto che l'applicazione rimanga attiva per intervalli temporali molto lunghi. Alcuni esempi classici sono i voli spaziali od anche i sistemi satellitari. Requisito classico per applicazioni di questo tipo è quello di rimanere operanti senza manifestare malfunzionamenti per periodi di decine di anni con probabilità pari, o addirittura superiore, a 0.95. Questa necessità deriva dal fatto che per alcune di esse (quali ad esempio sistemi di controllo di sonde spaziali) è impossibile poter apportare riparazioni per rendere il sistema stesso operante in maniera corretta in caso di malfunzionamento.

2.2. Applicazioni critiche

In questa classe ricadono tutte quelle applicazioni in cui malfunzionamenti possono avere delle conseguenze gravi sulla sicurezza delle persone, dell'ambiente o di impianti. Esempi classici sono le applicazioni per il controllo del volo di aeroplani, applicazioni militari o anche certi tipi di applicazioni per il controllo di processi industriali. In applicazioni critiche un malfunzionamento potrebbe causare conseguenze realmente devastanti per cui è tipico richiedere ai sistemi che le supportano la quasi certezza che non si verifichino malfunzionamenti per intervalli temporali adeguatamente lunghi. Il requisito può comunque essere meno stringente a seconda dalla particolare funzionalità dell'applicazione stessa.

2.3. Applicazioni poco mantenibili

Tipicamente, questa classe di applicazioni include tutte quelle ove operazioni di manutenzione sono estremamente costose, o difficili da apportare. Applicazioni di calcolo su sistemi remoti o anche alcune applicazioni spaziali sono classici esempi. In tal caso, l'obiettivo principale è quello di riuscire a rinviare il più possibile la manutenzione garantendo comunque che il sistema funzioni correttamente fino all'istante dell'intervento. Una centrale di commutazione telefonica è un esempio in cui la manutenzione viene effettuata periodicamente e ciò che si vuole è che la centrale stessa funzioni in ogni caso in maniera adeguata nell'intervallo temporale tra due manutenzioni successive. In tale intervallo temporale il sistema deve essere in grado di poter gestire il verificarsi di malfunzionamenti in maniera autonoma.

2.4. Applicazioni disponibili

Le applicazioni disponibili sono tutte quelle che forniscono servizi e per le quali gli utenti richiedenti il servizio si aspettano con alta probabilità che il servizio venga realmente fornito all'atto della richiesta. E' come dire che tali applicazioni devono rendere indisponibili i propri servizi con bassissima probabilità. Esempi classici sono le applicazioni di gestione di transazioni bancarie oppure i sistemi per la gestione di servizi anagrafici.

3. Descrizione degli impedimenti

I tre termini comunemente usati nella descrizione degli impedimenti per la dependability sono: *guasto*, *errore* e *malfunzionamento*. I guasti sono l'origine degli errori i quali poi possono dar luogo ai malfunzionamenti. Tipicamente, un guasto è una imperfezione in qualche componente hardware o software. Esso può o non dar luogo ad un errore, il quale viene comunemente definito come una deviazione dello stato del sistema dai possibili stati previsti. La presenza di un errore non necessariamente causa che il sistema esegua qualche sua funzione in maniera non corretta. In ogni caso, se questo accade allora siamo in presenza di un malfunzionamento. E' da notare che il malfunzionamento può essere visto come la non corretta esecuzione di una funzione sia in termini quantitativi che in termini qualitativi. Nel primo caso il sistema garantisce l'esecuzione della funzionalità per cui è stato progettato ma a regime ridotto, nel secondo caso non la garantisce.

La relazione di causa effetto tra guasti, errori e malfunzionamenti porta come naturale conseguenza alla definizione di due parametri che giocano un ruolo di base nella progettazione di sistemi tolleranti i guasti: la *latenza del guasto* e la *latenza dell'errore*. La prima corrisponde

all'intervallo temporale che intercorre tra l'occorrenza del guasto ed il verificarsi dell'errore; la seconda invece rappresenta il tempo intercorrente tra il verificarsi di un errore ed il conseguente malfunzionamento. Il tempo totale che intercorre tra il verificarsi di un guasto ed il manifestarsi di un malfunzionamento risulta quindi pari alla somma di queste due latenze.

Sistemi malfunzionanti possono essere riportati al corretto funzionamento tramite delle azioni correttive. E' possibile però che il sistema mostri poi un nuovo malfunzionamento. L'intervallo di tempo medio tra due malfunzionamenti è generalmente noto come MTBF (Mean Time Between Failures). Statistiche recenti hanno mostrato che in sistemi convenzionali MTBF assume valori tipici compresi tra le sei e le dieci settimane, con il 50% circa dei malfunzionamenti originati da guasti hardware su processori o dischi.

I guasti traggono origine da una varietà di cause, esistono comunque sei voci primarie per poterli classificare: la fenomenologia, la natura, la fase di occorrenza, l'estensione, il valore e la durata.

La *fenomenologia* porta ad una distinzione tra guasti fisici, ovvero originati da condizioni fisiche estreme, quali per esempio eccessive interferenze elettromagnetiche, e guasti "umani", originati invece da "imperfezioni" dell'uomo. Un errato comando impartito da un operatore umano è un classico esempio di quest'ultimo tipo di guasti.

La *natura del guasto* specifica il suo tipo, ossia se si tratta di un guasto hardware o software, oppure se si tratta di un guasto che coinvolge la parte analogica (per esempio la parte di alimentazione o i componenti elettromeccanici eventualmente presenti) o la parte digitale (per esempio il processore) del sistema.

La *fase di occorrenza* specifica se il guasto è originato durante la realizzazione (progettazione ed implementazione) del sistema, oppure occorre durante il normale funzionamento. Esempi classici di guasti in fase di realizzazione sono le inesattezze nella specifica di progetto, le quali possono dar luogo, per esempio, ad algoritmi od architetture non corretti, l'utilizzo di componenti difettosi, gli errori di codifica dei programmi ed anche le imperfezioni generate durante il processo di produzione dei componenti hardware.

L'*estensione* specifica se un guasto è localizzato in uno specifico componente hardware/software, se coinvolge più componenti e quali.

Il *valore* specifica una sorta di intensità del guasto; esso può essere costante o anche modificarsi nel tempo. Ad esempio, se misuriamo l'intensità di un guasto hardware in termini di numero di chip coinvolti, potremmo avere che all'atto del suo manifestarsi il guasto coinvolge un solo chip e poi, dopo un dato intervallo temporale, esso arriva a coinvolgerne dieci.

Infine, la *durata* identifica l'intervallo temporale in cui il guasto si manifesta; se il guasto è *transiente* esso si manifesterà ad un dato istante temporale e scomparirà poi autonomamente senza che alcuna azione correttiva esterna venga effettuata; al contrario, un guasto *permanente* si manifesterà per sempre a meno che un'azione correttiva non venga apportata; infine esiste il guasto *intermittente* il quale appare e scompare autonomamente in maniera ripetuta. Tipicamente la maggioranza dei guasti, circa l'80%, è transiente o intermittente; solo il rimanente 20% è permanente.

Per quanto riguarda invece gli errori, come già detto in precedenza essi sono il risultato di guasti e la potenziale causa di malfunzionamenti. Per essi non esiste una classificazione specifica come invece esiste per i malfunzionamenti eventualmente originati. In paricolare, questi vengono classificati in base alle voci: dominio, percezione, e conseguenza.

Il *dominio* divide i malfunzionamenti in base al valore e agli effetti temporali. Nel primo caso il malfunzionamento è tale che le funzioni specifiche del sistema non possono essere portate a compimento poiché il risultato del processamento effettuato dal sistema è un valore scorretto; invece nel secondo caso esse non possono essere portate a compimento a causa di errate

temporizzazioni nel processamento. Più in particolare, malfunzionamenti temporali possono essere dovuti sia ad un anticipo che ad un ritardo nel processamento. Inoltre, sia in quelli per valore che in quelli temporali possiamo individuare i cosiddetti malfunzionamenti di tipo *halt*, per i quali nessuna attività del sistema può essere più rilevata.

La *percezione* sul malfunzionamento origina invece due classi di malfunzionamenti note come: *consistente* ed *inconsistente*. I malfunzionamenti consistenti sono tutti quelli per cui qualsiasi entità interagente con il sistema malfunzionante riesce a catturare la medesima percezione del malfunzionamento. Un esempio è il così detto malfunzionamento di tipo *stop* (noto in inglese come "fail-stop"), in cui il sistema subisce un malfunzionamento di tipo halt ed ogni entità può rivelarlo. I malfunzionamenti inconsistenti, noti anche come *Bizantini*, sono quelli dove non vi è una percezione comune sul fatto che il sistema sia realmente malfunzionante.

Infine, la classificazione basata sulla voce *conseguenze* divide i malfunzionamenti in due tipi: *benigni* e *catastrofici*. I primi sono tali per cui le conseguenze sono paragonabili ai benefici ottenibili in assenza di malfunzionamento; i secondi sono quelli per cui le conseguenze sono di gran lunga differenti.

4. Attributi della dependability

Nel progetto di un sistema dependable non si devono raggiungere solo alcuni obiettivi funzionali e/o prestazionali, ma anche una serie di obiettivi addizionali quali: l'*affidabilità*, la *disponibilità*, la *sicurezza*, la *mantenibilità*, la *collaudabilità* e la *performability*. La *dependability* descrive la qualità di un particolare sistema in termini di questi obiettivi che, come vedremo tra breve, non sono altro che esempi di misure usate per quantificare specifiche qualità del sistema.

4.1. Affidabilità

L'affidabilità è una funzione del tempo *R(t)* definita come la probabilità che il sistema non mostri malfunzionamenti in un intervallo temporale a condizione che nessun malfunzionamento esisteva all'inizio dell'intervallo. In parole più semplici, l'affidabilità è la probabilità che il sistema funzioni correttamente in un dato intervallo temporale. Essa è spesso utilizzata per caratterizzare sistemi per i quali anche un solo periodo momentaneo di malfunzionamento è inaccettabile. Considerando per esempio un sistema di controllo degli alettoni di un aeroplano, ciò che assolutamente non si vorrebbe è che per l'intervallo di durata del volo l'affidabilità del sistema sia intorno allo 0.5. In tal caso esisterebbe una possibilità su due che il sistema si guasti durante il volo stesso. Per tale tipo di applicazione, il valore desiderato per l'affidabilità dovrebbe essere il più vicino possibile a 1.0, il che implica la quasi certezza che il sistema stesso non mostri malfunzionamenti durante il volo.

4.2. Disponibilità

La disponibilità è una funzione del tempo *A(t)* definita come la probabilità che il sistema non mostri malfunzionamenti nell'istante in cui gli è richiesto di operare. Si differenzia dall'affidabilità poiché quest'ultima è una misura di corretto funzionamento in un intervallo, mentre la disponibilità è una misura di corretto funzionamento ad un dato istante temporale. Un sistema può essere altamente disponibile nonostante esso mostri frequenti, ma molto brevi, periodi di malfunzionamento. La disponibilità è una buona misura per caratterizzare quei sistemi in cui sono accettabili malfunzionamenti, purchè nella maggior parte delle circostanze il sistema funzioni in

modo corretto. Considerando per esempio un sistema di emissione di certificati anagrafici, intervalli di malfunzionamento (ad esempio in cui l'emissione dei certificati non può essere effettuata) possono essere comunque accettati purché la grande maggioranza degli utenti all'atto della richiesta del certificato trovi il sistema operante e venga quindi provvista di questo nei tempi di emissione prestabiliti.

4.3. Sicurezza

La sicurezza è la probabilità *S(t)* che il sistema non mostri malfunzionamenti nell'istante in cui gli è richiesto di operare , oppure che, anche se esso mostra un malfunzionamento, questo non comprometta la sicurezza di persone o impianti relazionati al sistema stesso. In altre parole, essa è una misura sia della capacità del sistema di funzionare correttamente che della capacità di non provvedere correttamente alle sue funzionalità ma senza generare conseguenze rilevanti. E' da notare che la sicurezza differisce dall'affidabilità e dalla disponibilità, poiché queste ultime sono misure relative al corretto funzionamento e non includono effetti derivanti da malfunzionamenti.

4.4 Performability

La performability è una funzione del tempo *P(L,t)* definita come la probabilità che il valore del livello di funzionalità del sistema al tempo *t* sia almeno pari ad *L*. Essa gioca un ruolo fondamentale nella progettazione di sistemi nei quali la presenza di guasti non implica la mancata effettuazione di alcune funzionalità, ma solamente una riduzione del livello con il quale esse vengono eseguite. Si consideri un sistema satellitare per l'acquisizione e la trasmissione dei dati nel controllo del traffico aereo. Questo sistema deve provvedere a comunicare posizioni, velocità e traiettorie di aerei in volo entro tempi prestabiliti. Si supponga che si verifichi un guasto a bordo di un satellite. In tal caso è accettabile che il satellite elabori dati in ingresso a frequenza ridotta, purché esso riesca a fornire i dati richiesti alla centrale di controllo nei tempi prestabiliti. Ovviamente questo accadrà se il sistema manterrà almeno una data capacità di elaborazione pur in presenza del guasto, ovvero manterrà almeno un dato valore per la sua performability.

4.5. Mantenibilità

La mantenibilità è una misura della facilità con la quale un sistema può essere riparato una volta manifestatosi il malfunzionamento. In maniera più specifica, la mantenibilità è la probabilità *M(t)* che il sistema malfunzionante possa essere riportato al suo corretto funzionamento entro il periodo *t*. Essa è strettamente correlata con la disponibilità poiché tanto più è breve l'intervallo di ripristino del corretto fuzionamento, tanto più elevata sarà la probabilità di trovare il sistema funzionante ad un dato istante temporale. Per il valore estremo *M(0)* = 1.0, il sistema in oggetto sarà sempre disponibile.

4.6. Collaudabilità

La collauabilità è semplicemente una misura di quanto un sistema rende facile ad un operatore verificare alcuni suoi attributi. E' chiaramente legata alla mantenibilità poiché più facilmente è possibile collaudare un sistema malfunzionante per individuare il componente guasto, più breve sarà l'eventuale intervallo di ripristino del corretto funzionamento.

5. Strumenti per la dependability

Gli obiettivi della dependability possono essere raggiunti tramite tre tecniche classiche: la tolleranza dei guasti, la rimozione dei guasti e la previsione dei guasti. In questo Capitolo verranno descritte tutte e tre queste tecniche.

5.1. La tolleranza dei guasti

La tolleranza dei guasti ha due obiettivi fondamentali. In primo luogo quello di scoprire la presenza di eventuali errori e cercare di rimuoverne gli effetti prima che questi diano poi luogo effettivamente ad un malfunzionamento; questo obiettivo viene raggiunto tramite la tecnica del *processamento dell'errore*. In secondo luogo di impedire che i guasti possano poi originare altri errori; ciò viene comunemente raggiunto tramite la tecnica del *trattamento del guasto*.

Il processamento dell'errore consta usualmente di tre passi fondamentali. In primo luogo vi è la *rivelazione d'errore*, tesa ad individuare, possibilmente in maniera molto precoce, la presenza di errori. La precocità con cui questa fase deve operare è rilevante dal momento che in alcune circostanze esiste la possibilità di propagazione dell'errore che potrebbe poi rendere enormemente complessi i successivi passi. Poi si ha la *diagnosi d'errore*, tesa ad individuare i danni causati dall'errore. Infine si ha la fase di *recupero dall'errore* che può essere di tre tipi: recupero all'indietro, recupero in avanti oppure recupero per compensazione. Il recupero all'indietro consiste nel riportare il sistema in uno stato precedente alla manifestazione dell'errore, e quindi di corretto funzionamento. Questa tecnica necessita della registrazione di un sottoinsieme degli stati attraversati dal sistema per un possibile utilizzo futuro; questi vengono comunemente denominati *checkpoint* (punti di recupero). E' possibile che facendo ripartire il sistema da uno stato precedente non affetto da errori allora l'errore stesso non si rimanifesti in futuro. Questa situazione è classica di errori originati da guasti transienti. Il recupero in avanti consiste nella ricerca di un nuovo stato caratterizzato da assenza di errore a partire dallo stato corrente erroneo. Questo tipo di recupero può comportare una lunga sequenza di passi di trasformazione prima di riuscire ad ottenere uno stato con assenza di errore. Il recupero per compensazione consiste invece nella trasformazione immediata dello stato affetto in uno stato non più affetto da errore. Quest'ultima soluzione richiede in genere una maggiore complessità di progetto e costo finale del sistema.

E' da notare che queste tre tecniche differenti non si escludono mutuamente, per cui è possibile per esempio operare un recupero all'indietro sperando che il guasto sia transiente, e quindi che l'errore non si manifesti più, e poi eventualmente in caso di successiva manifestazione si può operare un recupero in avanti.

La tecnica del trattamento del guasto è tesa a far sì che guasti di natura non transiente non diano origine a nuovi errori. Essa consiste di una fase di *diagnosi del guasto*, tesa ad individuare il o i componenti guasti, una fase di *isolamento del guasto* in cui i componenti guasti vengono disattivati, ed una eventuale fase di *riconfigurazione*. Quest'ultima viene attivata solo qualora, dopo l'isolamento dei componenti guasti, il sistema non riesce più a garantire un funzionamento adeguato, come per esempio un prefissato valore di performability. E' da notare che esiste un caso particolare di isolamento in cui il componente guasto viene lasciato attivo ma si previene la generazione degli errori; questa tecnica è comunemente denominata *mascheramento del guasto*.

Nei seguenti paragrafi verrà analizzata in dettaglio una tecnica di progetto per la tolleranza dei guasti, e quindi per il processamento d'errore ed il trattamento del guasto, basata sulla nozione di ridondanza. Essa consiste nell'aggiunta di componenti addizionali a quelli strettamente necessari per il normale funzionamento del sistema. I componenti addizionali possono essere hardware, software oppure anche copie addizionali di dati. Verrà poi discussa una forma differente di

ridondanza, denominata ridondanza temporale, che non necessita di componenti addizionali ma da luogo a latenze maggiori per il processamento d'errore ed il trattamento del guasto. Infine verrà presentata una panoramica di problematiche aggiuntive per la tolleranza dei guasti in sistemi distributiti.

5.1.1. Ridondanza a livello hardware

La più comune forma di ridondanza hardware consiste nell'avere a disposizione copie multiple dello stesso componente hardware. Date le piccole dimensioni di componenti attuali ed il loro basso costo, questa tecnica viene molto usata in pratica.

Una prima forma di replicazione hardware è la cosiddetta *replicazione passiva*, strettamente collegata alla tecnica del mascheramento dei guasti. Essa permette di prevenire che guasti originino errori senza che vi sia alcun intervento esplicito da parte del sistema stesso o di un operatore esterno per quanto riguarda il rivelamento del guasto ed una eventuale riconfigurazione.

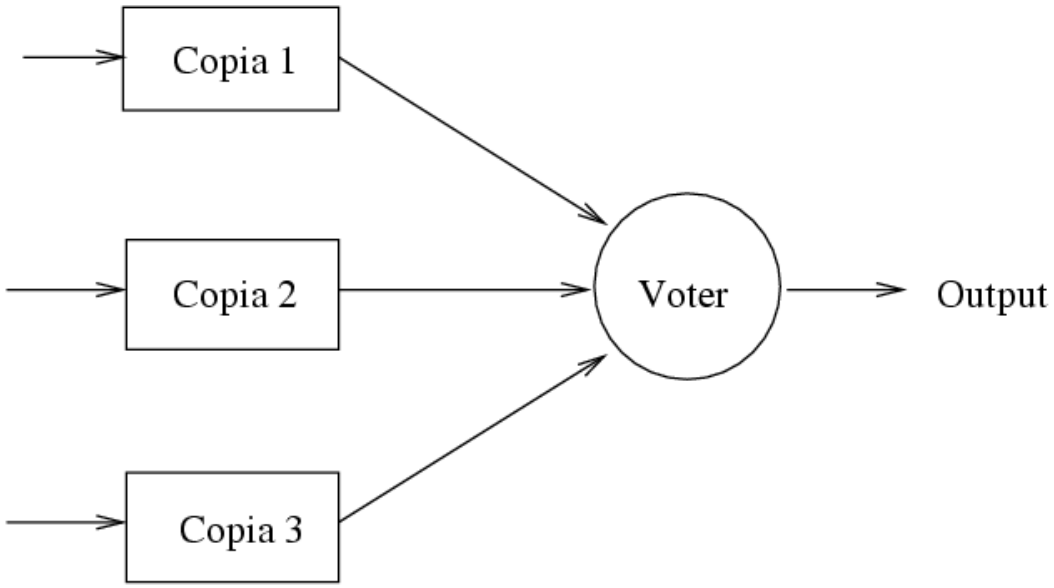

Fig. 5.1 - Ridondanza a tre moduli.

Questo tipo di ridondanza è basato su di un meccanismo di votazione di maggioranza in cui, in generale, esistono almeno tre copie distinte del componente ed un modulo voter. Il caso di tre copie esatte viene comunemente denominato ridondanza a tre moduli (triple modular redundancy - TMR), un esempio del quale viene riportato in figura 5.1. Se una delle copie è affetta da un guasto, le altre due mascherano il risultato della copia guasta tramite il loro valore di output che risulterà concorde. Supponiamo che la copie 1 e 2 non siano guaste, mentre la copia 3 sia affetta da un guasto. In tal caso il valore di output delle copie 1 e 2 sarà identico, per esempio A, mentre il valore di output della copia 3 soggetta a guasto potrebbe essere differente da A, per esempio B. In tal caso, la maggioranza delle copie fornisce il valore di uscita A che verrà interpretato come valore corretto, e quindi associato a copie non guaste.

In questo tipo di schema, vengono mascherati guasti a livello dei componenti, ma non eventuali guasti a livello del voter, per il quale non vi è alcuna ridondanza. Se il voter è soggetto a

guasto allora il suo valore di output potrebbe essere scorretto ed originare quindi errori e poi malfunzionamenti. Per questo tipo di schema, l'affidabilità dell'intero sistema non può mai essere superiore all'affidabilità del voter. Dato un sistema, ogni singolo componente il cui guasto può dar luogo a malfunzionamento del sistema stesso viene denominato *singolo punto di fallimento*. Una tecnica classica per prevenire che il voter diventi un singolo punto di fallimento è quella di replicarlo. In figura 5.2 viene mostrato uno schema con replicazione sia del componente processore, che del componente memoria, che del componente voter. In questo caso, il voto di maggioranza provvede a far si che la memoria contenga i dati corretti anche in presenza di un processore guasto. Inoltre, la replicazione del voter farà si che un guasto in un voter non potrà avere conseguenze su più di una copia dei dati.

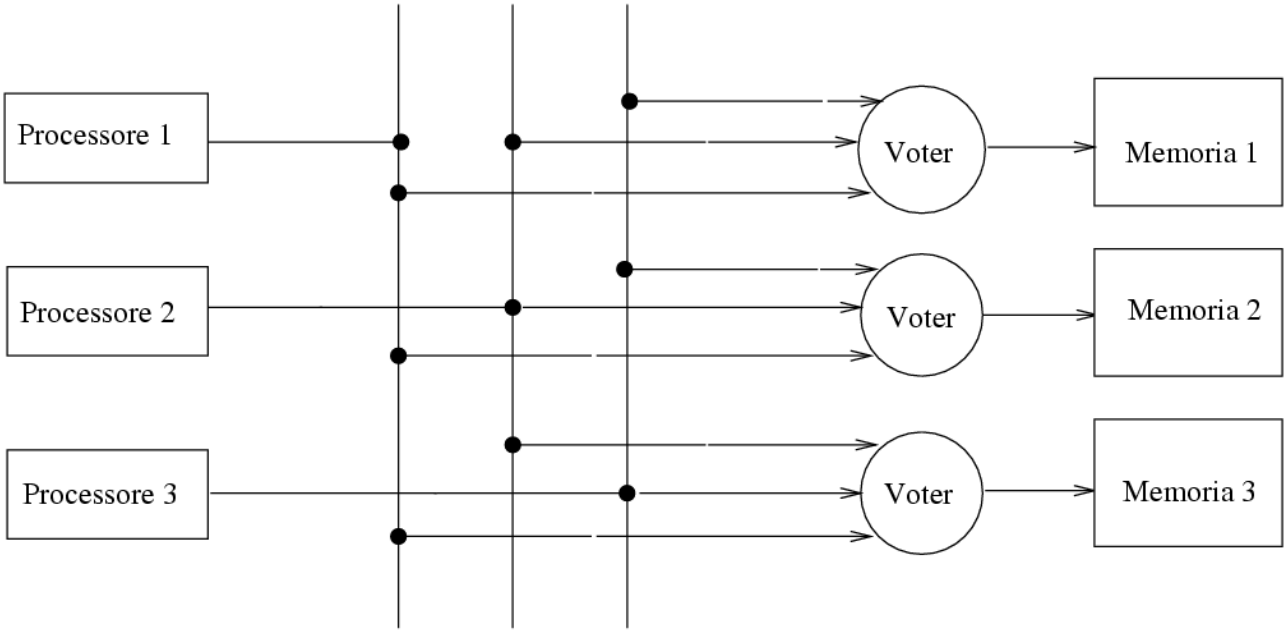

Fig. 5.2 - Replicazione di processore, memoria e voter.

Una generalizzazione dell'approccio TMR è la cosiddetta ridondanza ad N moduli (N-modular redundancy - NMR), la quale differisce da TMR nel fatto che il componente in oggetto viene replicato N volte, con N possibilmente maggiore di 3. Il vantaggio di utilizzare N > 3 moduli risiede nel fatto che più di un singolo guasto può essere tollerato. Ad esempio, il voto di maggioranza permette ad un sistema di tipo 5MR di tollerare il guasto di un massimo di due componenti. Ovviamente lo svantaggio sarà associato al maggiore costo dovuto al più alto numero di copie del modulo per cui si adotta la ridondanza.

Una problematica associata al discorso della ridondanza passiva è quella di identificare una implementazione adeguata per il voter. Quest'ultimo potrebbe infatti essere implementato esclusivamente a livello hardware o a livello software. Un voter implementato a livello software potrebbe sfruttare l'eventuale presenza di un processore nel sistema in modo tale da poter eseguire l'operazione di voto con una minima richiesta di hardware addizionale. Inoltre, un voter implementato via software risulterà più facilmente modificabile rispetto al corrispettivo implementato via hardware. D'altro canto, lo svantaggio è che l'operazione di voto potrebbe comportare maggiori ritardi poiché, in generale, un processore non può eseguire le operazioni associate al voter così velocemente come farebbe un hardware dedicato a tale scopo. Un esempio di

voter implementato via software è mostrato in figura 5.3. Vi è una ridondanza costituita da tre distinti processori i quali eseguono lo stesso programma A. Al termine dell'esecuzione, ciascuno dei processori potrebbe eseguire le operazioni di voto per implementare la ridondanza anche a livello del voter.

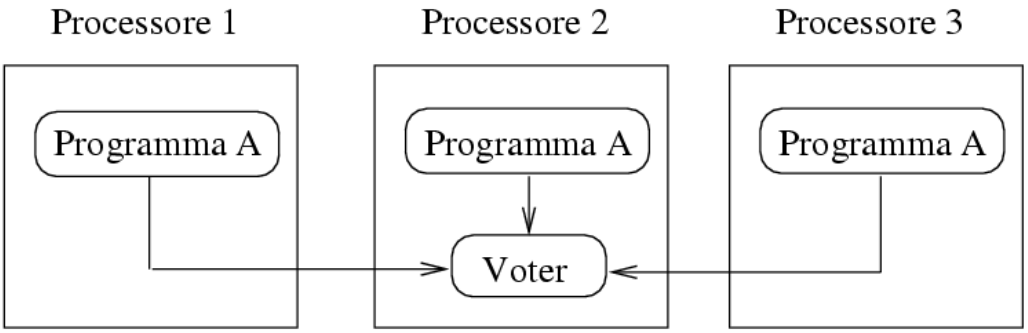

Fig. 5.3 - Voter implementato via software.

Un altro tipo di ridondanza è quella attiva, il cui scopo è di raggiungere l'obiettivo della tolleranza dei guasti non tramite mascheramento. E' da notare che in questo caso non c'è alcun tentativo di prevenzione affinché guasti non diano origine ad errori ed a malfunzionamenti, ma esistono solo azioni di rimedio per riportare il sistema al corretto funzionamento. Quindi tale tipo di ridondanza risulterà più appropriata per tutte quelle applicazioni in cui la presenza, se pur temporanea, di errori e malfunzionamenti è accettabile. Applicazioni tipiche sono per esempio quelle satellitari in cui la ridondanza passiva per il mascheramento di eventuali guasti potrebbe comportare costi enormi, ed in ogni caso brevi periodi di malfunzionamento possono comunque essere accettati.

L'approccio della ridondanza attiva può essere spiegato tramite il diagramma a stati in figura 5.4. La presenza di un guasto a partire da una situazione di normale funzionamento può dar luogo ad una sequenza di transizioni di stato fino alla manifestazione di un errore o di un malfunzionamento. Quando la manifestazione di un errore/malfunzionamento viene rivelata, vengono attivate sia una fase di riconfigurazione per il trattamento di guasto che una azione di recupero (all'indietro, in avanti o per correzione) tendente a riportare il sistema stesso in uno stato non più affetto da errore. Il sistema riprenderà quindi un corretto funzionamento oppure un funzionamento a livello ridotto, dipendendo dalla entità del guasto e dal livello di ridondanza presente.

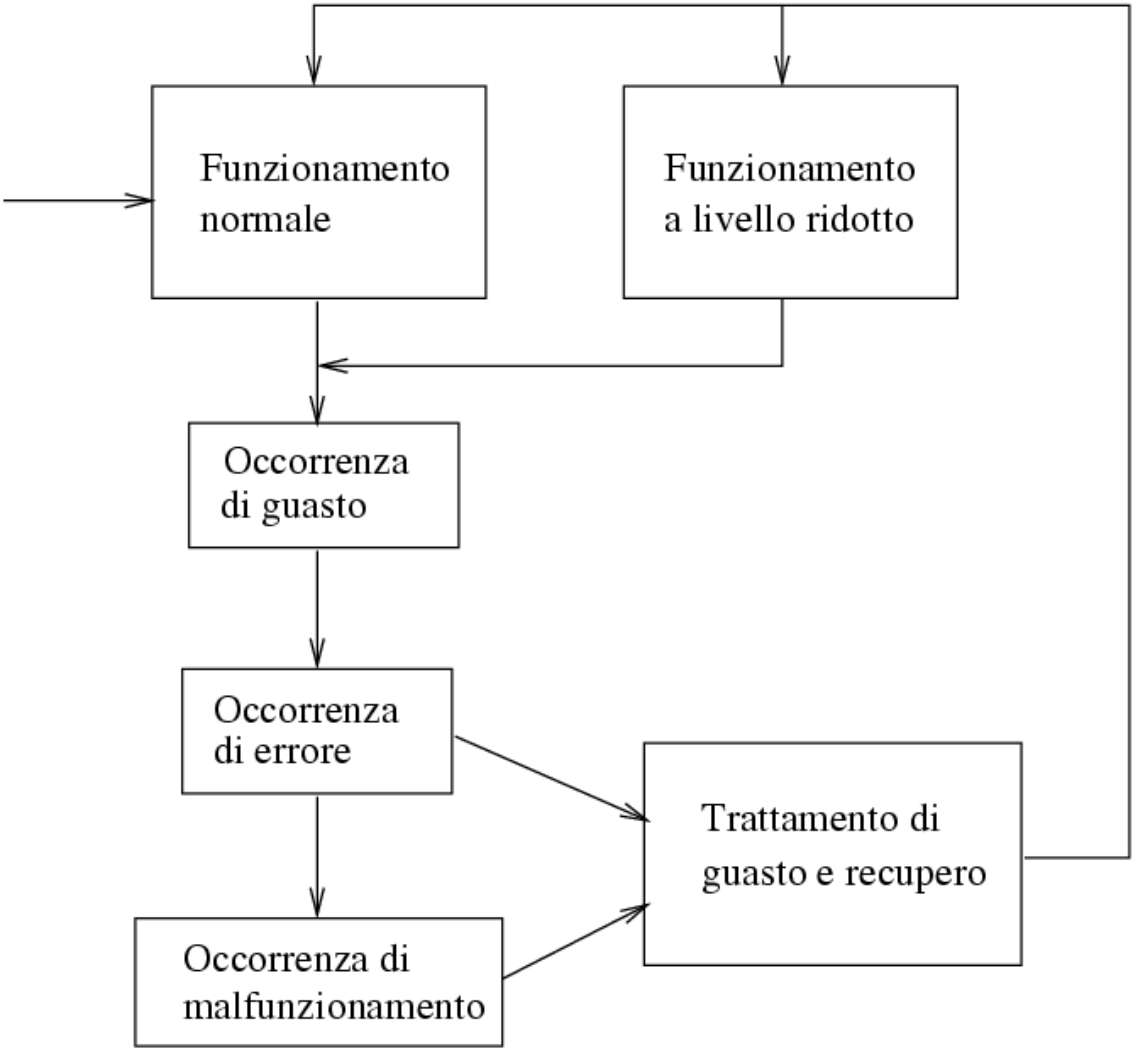

Fig. 5.4 - Diagramma a stati per la ridondanza attiva.

Una tecnica classica di rivelazione di errori/malfunzionamenti utilizzata nella ridondanza attiva è la cosiddetta duplicazione con comparazione, in cui vengono utilizzati due identici componenti per effettuare la stessa operazione. Al termine dell'operazione il risultato viene comparato e, nel caso i due valori siano non concordi, viene rivelata l'occorrenza di un guasto in almeno uno dei due componenti. Questa tecnica non ha la capacità di identificare il componente guasto per cui l'identificazione è demandata alla fase di riconfigurazione.

Una forma particolare di ridondanza attiva è la cosiddetta *riserva fredda*, in cui per ogni modulo soggetto a ridondanza esiste una sua copia non attiva, che viene attivata in caso di guasto della copia originale, da cui il nome di riserva fredda. In tal caso, la riconfigurazione ha lo scopo di sostituire la copia originale con la riserva fredda. E' da notare che, durante il periodo di riconfigurazione necessario all'attivazione della riserva, la funzionalità del sistema viene interrotta. Per minimizzare la durata di questo periodo è possibile utilizzare *riserve calde*, che a differenza di quelle fredde, sono attive contemporaneamente al modulo per cui fungono da ridondanza. Lo svantaggio principale è che queste riserve consumano costantemente energia per rimanere attive, anche quando non forniscono reali servizi.

Infine, un terzo approccio classico alla ridondanza hardware, denominato *ridondanza ibrida*, combina le caratteristiche salienti delle due soluzioni precedenti. In particolare, in questa soluzione

viene adottata sia una tecnica di mascheramento dei guasti per prevenire gli errori, sia una tecnica di diagnosi dei guasti stessi con relative azioni di riconfigurazione per isolare il componente guasto. In generale la ridondanza ibrida è implementata tramite una combinazione di ridondanza NMR in cui sono presenti anche riserve (fredde o calde). Le riserve subentreranno ad eventuali componenti guasti facenti parte dell' insieme originale di N repliche.

5.1.2. Ridondanza a livello software

La ridondanza software può apparire in svariate forme dato che non è strettamente necessario replicare completamente i programmi per ottenerla. Ad esempio, essa può consistere nell'aggiunta di linee di codice utilizzate per eseguire una certa azione di controllo, come per esempio la scrittura e la lettura periodica di locazioni di memoria per controllarne il corretto funzionamento.

Il *controllo della consistenza* è uno dei primi obiettivi della ridondanza software. Esso consiste nell'utilizzo di conoscenza a priori delle caratteristiche dell'informazione che viene gestita dall'applicazione al fine di rivelare eventuali guasti/errori/malfunzionamenti. Si consideri ad esempio un'applicazione in cui è noto che il valore di un determinato parametro non possa mai superare una data soglia. Se questo accade allora vi è un problema in qualche componente del sistema. Per questa applicazione è possibile progettare una specifica routine di controllo che effettui il monitoraggio sul valore del parametro e riveli un eventuale superamento della soglia attivando opportune azioni di ripristino. Un altro esempio classico di controllo della consistenza è il monitoraggio della reale prestazione di un sistema e la comparazione del valore ottenuto con un valore previsto per essa. Se vengono rilevati scostamenti significativi, allora è il caso di attivare procedure di localizzazione di eventuali guasti/errori/malfunzionamenti. E' da notare che il controllo della consistenza potrebbe essere implementato anche via hardware. In tal caso, una pur piccola variazione dell'entità controllata, dovuta per esempio a successive modifiche apportate alla specifica iniziale del sistema, potrebbe richiedere un nuovo progetto per l'hardware che ne effettua il controllo.

Il *controllo di capacità* è un altro degli obiettivi della ridondanza software ed ha lo scopo di collaudare i singoli componenti hardware del sistema per verificarne il corretto funzionamento. Per esempio, se si desidera sapere se tutta la memoria presente è realmente disponibile, è possibile utilizzare del software che acceda alla memoria e ne verifichi il corretto funzionamento. Un altro esempio di controllo di capacità è il test che si effettua sulle ALU, il quale consiste nel fare eseguire alla ALU specifiche istruzioni (addizioni, moltiplicazioni, operazioni logiche e trasferimento) su particolari dati e si compara poi il risultato con valori noti memorizzati per esempio in una memoria a sola lettura (ROM). Infine, un'altra forma di controllo di capacità consiste nel verificare che tutti i processori del sistema sono realmente in grado di comunicare ciascuno con gli altri. Questa può essere implementata tramite opportuni moduli software che periodicamente provvedono a trasferire specifiche informazioni tra coppie di processori.

Gli esempi di ridondanza software considerati fino ad ora utilizzano moduli aggiuntivi per effettuare il controllo di consistenza/capacità. Ciò che ancora non è stato considerato sono le tecniche per scoprire e possibilmente tollerare guasti che possono coinvolgere il software stesso. A differenza dell'hardware, il software non si "rompe", ciò che invece può accadere è che si manifestino guasti originati da progetti software non corretti oppure anche da errori nel processo di codifica degli algoritmi. Quindi ogni tecnica che cerca di scoprire guasti nel software di fatto cerca di scoprire imperfezioni nel processo di progetto o di codifica. Ovviamente, una semplice duplicazione e comparazione tra due moduli software identici non ha la capacità di rivelare tali imperfezioni. Per affrontare questo problema, la ridondanza software più comunemente utilizzata è

quella che fa uso di *N programmi autocontrollanti* (N self-checking programs - NSCP), in cui esistono N versioni distinte dello stesso software. Cascuna di esse contenente i propri test di accettazione, i quali non sono altro che azioni di controllo sui risultati prodotti dal programma. Viene poi utilizzato un modulo che implementa una data logica di selezione per prelevare i risultati prodotti da uno dei programmi in cui tutti i test di accettazione hanno avuto esito positivo. Supponendo, come sia ragionevole, che i possibili guasti in una delle versioni non siano correlati con quelli delle altre, e che gli errori da loro generati siano scoperti tramite i test di accettazione, allora la tecnica NSCP può tollerare fino ad un massino di N-1 guasti.

Un'altra tecnica di base per la ridondanza software, tesa a tollerare guasti nei moduli software è la cosiddetta *programmazione ad N versioni* (N version programming - NVP). In questa tecnica, ciascun modulo software viene progettato e codificato N volte da gruppi di progettisti/programmatori distinti. Le N versioni vengono poi utilizzate in un classico meccanismo di voto, simile a quello associato alla ridondanza hardware di tipo NMR. L'obiettivo in questo caso è un vero e proprio mascheramento del guasto nel software. Per questo tipo di soluzione esistono due problematiche specifiche. In primo luogo, poiché è statisticamente provato che progettisti/programmatori distinti tendano spesso a fare gli stessi tipi di errori, non vi è mai buona garanzia che due versioni distinte non avranno gli stessi identici guasti. In secondo luogo, le N versioni sono comunque sviluppate a partire dalla stessa specifica, per cui l'approccio NVP non ha la capacità di affrontare il problema delle imperfezioni nella specifica.

Infine, un'ultima tecnica per la tolleranza dei guasti di tipo software è la cosiddetta tecnica dei *blocchi di recupero* (recovery blocks). Essa è concettualmente simile alla tecnica delle N riserve fredde nella ridondanza hardware. In particolare, vengono impiegate N versioni distinte di un dato programma ed un solo insieme di test di accettazione. Una delle versioni è denominata primaria, le altre N-1 sono invece delle riserve fredde. La versione primaria rimane attiva fino a quando i test di accettazione non rivelano un errore. Il tal caso essa viene sostituita da una delle versioni secondarie e così via. Quindi, ogni qual volta che un errore viene scoperto tramite il test di accettazione, la versione correntemente usata viene rimpiazzata da una riserva fredda. Questa soluzione è illustrata in figura 5.5. Assumendo che i test di accettazione abbiano capacità completa di rivelare gli errori e che questi siano indipendenti nelle N versioni, questa tecnica permette di tollerare al più N-1 guasti.

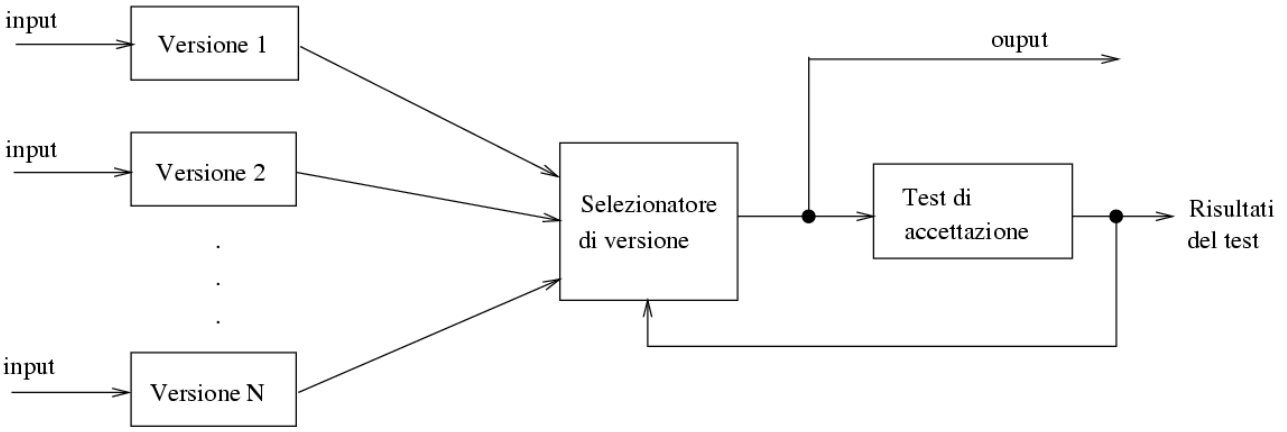

Fig. 5.5 - Tolleranza dei guasti di tipo software tramite blocchi di recupero.

5.1.3. Ridondanza a livello dei dati

La ridondanza a livello dei dati consiste nell'utilizzo di informazioni aggiuntive associate ai dati stessi al fine di poter garantire la correttezza del loro valore. L'aggiunta di questa informazione da luogo a codifiche dei dati cosiddette ridondanti, che permettono appunto o la rivelazione o addirittura la correzione di dati aventi valore corrotto a causa di un qualche guasto. Ogni codice è caratterizzato da una quantità massima di bit di informazione corrotti che esso riesce a rivelare o a correggere. Un codice in grado di rivelare/correggere al più un unico bit di informazione corrotto viene comunemente denominato codice rivelatore/correttore di singolo errore.

Un concetto fondamentale nella caratterizzazione dei codici di rivelazione/correzione degli errori è la *distanza di Hamming* tra due codifiche, definita come il numero di bit differenti di una codifica rispetto all'altra. Per esempio, i valori binari 0000 e 0001 differiscono per un solo bit (il meno significativo), essi hanno quindi distanza di Hamming pari ad uno. I valori binari 0000 e 0101 hanno invece distanza di Hamming pari a due. Ovviamente, se due codifiche hanno distanza di Hamming pari ad uno è possibile che una di esse venga a coincidere con l'altra attraverso la modifica di un singolo bit. Al contrario, se esse hanno distanza di Hamming pari ad almeno due, è impossibile trasformare una codifica nell'altra modificando un singolo bit. Viene definita *distanza di un codice* la minima fra le distanze di Hamming di due qualsiasi codifiche del codice. Se un codice ha distanza pari a due, allora la corruzione di un singolo bit nella rappresentazione di una codifica non potrà mai originare una codifica ammissibile per quel codice.

Altro concetto fondamentale associato ai codici di rivelazione/correzione degli errori è quello di *separabilità*, definita come la capacità di ottenere i dati desiderati a partire dal codice tramite semplice rimozione dell'informazione ridondante presente. Un codice non separabile non permette questo, così che l'estrazione del dato originario richiede operazioni più complesse rispetto alla semplice rimozione dell'informazione ridondante. Esiste una varietà di codici di rivelazione/correzione di errori, sia separabili che non, quali, per esempio, i codici di parità, i codici ciclici od aritmetici, il codice di Berger ed il codice di Hamming. Qui di seguito verranno descritti solo i codici di parità. Il lettore interessato può comunque consultare testi specialistici per informazioni ulteriori sugli altri codici.

Nei codici di *parità a singolo bit*, viene introdotto un unico bit addizionale nella rappresentazione di ciascun dato in modo tale che il numero finale di uni nella codifica del dato sia o un valore pari o un valore dispari. Nel primo caso si parla di codici di parità pari, nel secondo, di codici di parità dispari. Se un valore codificato con un codice di parità pari subisce la corruzione di un bit (ovvero una modifica del bit da 0 ad 1 o viceversa), tale corruzione viene immediatamente rivelata tramite il computo del numero globale di uni, che risulterà avere un valore dispari. L'applicazione più comune dei codici di parità a singolo bit è nella scoperta di guasti nelle memorie dei calcolatori. In particolare, prima che un dato viene scritto in una memoria, esso viene codificato tramite un codice di parità. All'atto della sua lettura, un eventuale guasto/errore nel componente memoria viene rivelato tramite il computo degli uni associati alla codifica del dato. Nei casi in cui il guasto/errore venga realmente rivelato, una procedura di riconfigurazione può essere attivata in modo da rendere non più operante la cella di memoria in oggetto. E' da notare che il valore corretto del dato viene comunque perso; questo deriva dal fatto che i codici di parità a singolo bit hanno solo una capacità di rivelazione, non di mascheramento o correzione. Un altro inconveniente classico dei codici di parità a singolo bit risiede nel fatto che essi non hanno la capacità di rivelare la corruzione di più di un bit, che è comunque un fenomeno possibile. Per esempio, si consideri la codifica 0001, che con l'aggiunta di un bit di parità pari viene poi a diventare 1-0001, e si supponga che vi sia la corruzione di entrambi i bit meno significativi il che da luogo alla codifica 1-010. Quest'ultima è

ammessa secondo il codice di parità pari poichè contiene un numero pari di uni, però non è la codifica corretta del dato in oggetto.

I codici di *parità per sovrapposizione* hanno anche la capacità di individuare quale sia il bit eventualmente errato che quindi può essere corretto tramite una semplice operazione di complementazione del suo valore. In questi codici, i bit propri del dato in oggetto vengono suddivisi in insiemi, possibilmente non disgiunti, e viene introdotto un bit di parità per ciascun insieme. Ciascuno degli insiemi finali includenti il bit di parità viene denominato insieme di parità. L'idea di base è di associare un dato bit ad una combinazione unica di insiemi sovrapposti, così che la corruzione del suo valore ha un effetto finale univocamente identificabile dal punto di vista della parità all'interno dei vari insiemi.

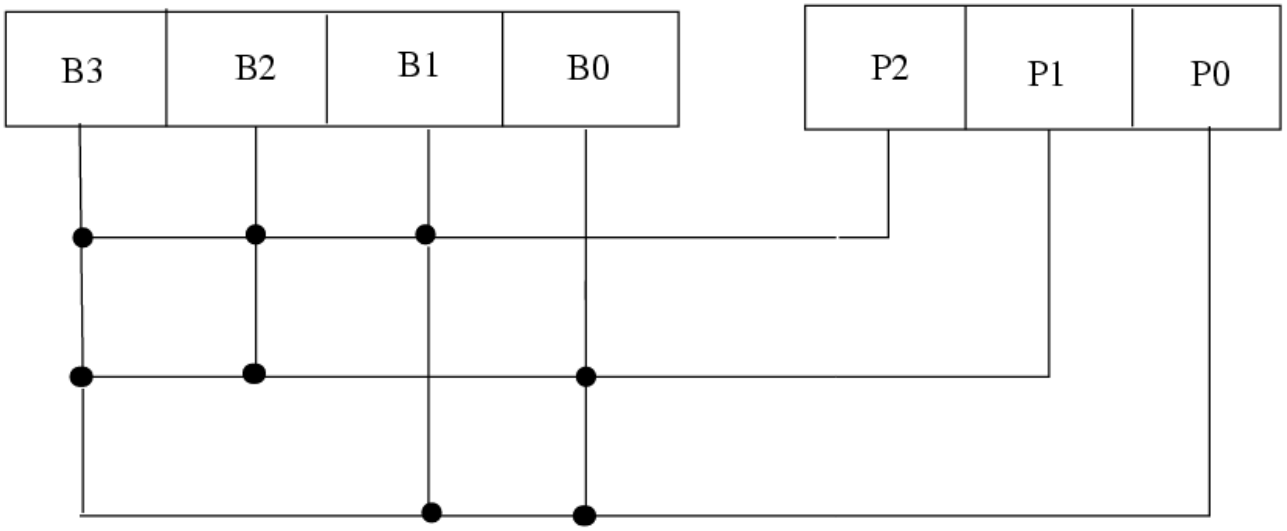

Fig. 5.6 - Un esempio di codice di parità per sovrapposizione.

Questa caratteristica dei codici di parità per sovrapposizione può essere descritta in maniera molto semplice attraverso l'esempio in figura 5.6 che mostra un caso particolare di codice di parità per sovrapposizione utilizzato per dati a 4 bit. Vengono costruiti tre insiemi di parità distinti: B3B2B1P2, B3B2B0P1 e B3B1B0P0. Il bit B3 appartiene a tutti e tre i gruppi di parità. In caso il suo valore sia corrotto, allora la parità di tutti e tre gli insiemi verrà alterata e univocamente esso verrà identificato come fonte del problema poichè nessun altro bit appartiene a tutti e tre gli insiemi di parità. Utilizzare un codice di parità per sovrapposizione per un dato di quattro bit comporta una penalità molto alta dal punto di vista della ridondanza (3 bit addizionali, quindi il 75% di informazione aggiuntiva). E' dimostrabile comunque che al crescere del numero di bit originali, la percentuale di bit addizionali decresce. In particolare, sia m il numero di bit originali e sia k il numero di bit di parità (cioè il numero di insiemi di parità) per garantire una parità per sovrapposizione. Dato che ogni bit errato deve produrre una singola configurazione per quanto riguarda gli insiemi di parità coinvolti allora il processo di controllo di parità deve essere i grado di produrre un insieme di possibilità che sia almeno pari a $m+k$ (cioè al numero di bit totali) più 1. La configurazione addizionale è associata al caso che nessuno tra gli $m+k$ bit ha un valore corrotto. Quindi otteniamo la seguente relazione:

$$2^k \geq m + k + 1$$

che ci permette di identificare il valore minimo di *k* affinché essa sia soddisfatta.

5.1.4. La ridondanza temporale

Le tecniche di ridondanza viste fino ad ora hanno lo svantaggio di dover utilizzare copie addizionali di componenti hardware e/o software. Per far fronte a questo problema, esiste una diversa forma di ridondanza, denominata *ridondanza temporale*. Questo tipo di ridondanza è consigliabile in tutte quelle applicazioni in cui il tempo non è un fattore critico.

Il concetto di base della ridondanza temporale è quello di eseguire un dato calcolo in sequenza più di una volta, e comparare i risultati ottenuti in ciascuna ripetizione per verificare se esista qualche discrepanza. In caso affermativo, la discrepanza è sintomatica di un errore e l'azione classica è quella di ripetere il calcolo tramite una nuova sequenza per verificare se tale discrepanza persiste o meno.

Originariamente la ridondanza temporale è stata impiegata soprattutto per il trattamento degli errori dovuti a guasti transienti, ove, una volta rivelato l'errore, successive ripetizioni dovrebbero rivelare assenza di discrepanza dovuta al fatto che il guasto scompare autonomamente. Comunque questo tipo di ridondanza ha la capacità di far scoprire anche guasti permanenti in componenti hardware a patto che si faccia uso di una minima quantità di hardware aggiuntivo. Lo schema in figura 5.7 riporta un esempio di ridondanza temporale per la scoperta di guasti permanenti. Al tempo T0 viene effettuata una prima computazione, poi al tempo T1 viene effettuata una seconda computazione in cui i dati di ingresso vengono codificati in ingresso al sistema di calcolo e poi decodificati in uscita. I risultati delle due computazioni vengono poi comparati per rivelare eventuali discrepanze. Le funzioni di codifica/decodifica sono selezionate in modo da poter rivelare guasti nel sistema di calcolo, classiche funzioni di decodifica utilizzate sono gli operatori di complementazione e gli shift aritmetici/logici.

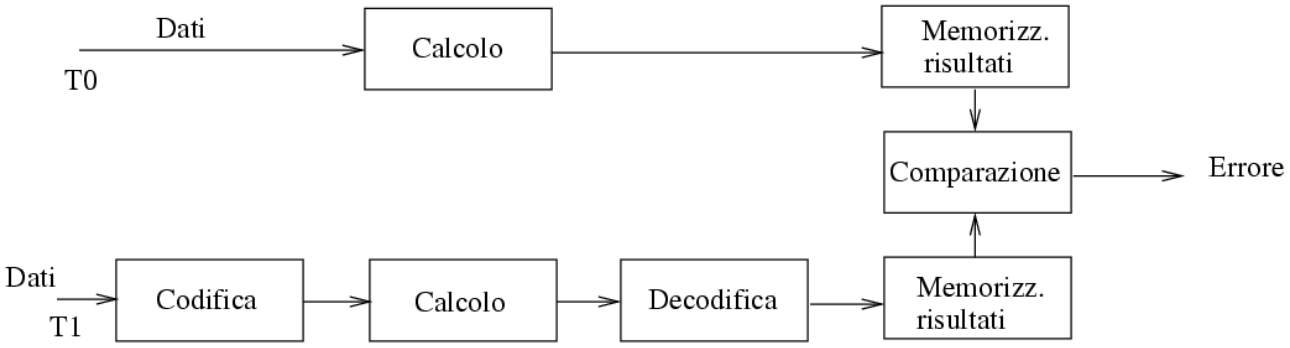

Fig. 5.7 - Ridondanza temporale per guasti permanenti.

Gli operatori di complementazione possono essere utilizzati in tutti i casi in cui il sistema combinatorio hardware per il calcolo abbia la proprietà di essere auto-duale (self-dual), ovvero se la funzione *F(X)* calcolata sull'ingresso *X* sia tale per cui *F(X)= not(F(not(X)))*. Per un circuito di tipo auto-duale la successiva applicazione di *X* ed *not(X)* come input produrrà una sequenza di uscite con valori alternati tra 0 ed 1. Se esiste almeno un insieme di valori di ingresso per cui la presenza di un guasto non origina una sequenza di uscite con valori alternati allora è possibile rivelare guasti. Il vantaggio degli operatori di complementazione risiede nel fatto che ciascun circuito combinatorio con *n* ingressi può essere trasformato in un circuito di tipo auto-duale con al più *n+1* variabili di ingresso.

Come detto in precedenza, la codifica può essere effettuata anche tramite ricalcolo con operatori di shift (recomputing with shift operators - RESO). In questo caso la funzione di codifica è uno shift a sinistra, quella di decodifica è uno shift a destra. Questo tipo di codifica è utilizzata qualora il circuito hardware sia di tipo bit-slice. In questo caso, supponendo che l'*i*-esimo slice sia guasto, allora avremo che nel calcolo con ingresso non codificato, il bit di uscita errato sara' l'*i*-esimo, invece nel caso di calcolo con ingresso codificato il bit di uscita errato sarà l'*i*-1-esimo. Le due uscite differiranno nel *i*-esimo o nel *i*-1-esimo bit o in entrambi. Quindi in caso di un singolo bit-slice guasto, supponendo che il guasto non abbia effetto su altri bit-slice, allora un singolo shift sarà in grado di rivelare il guasto. Nel caso in cui un guasto su un bit-slice possa avere ripercussioni sui bit-slice adiacenti allora la rivelazione del guasto necessita di shift di più di una singola posizione sugli ingressi e sulle uscite.

5.1.5. Il problema del ripristino nei sistemi distribuiti

Un sistema distribuito può essere schematizzato come un insieme di nodi di elaborazione caratterizzati da determinati componenti hardware/software ed una rete di comunicazione per permettere lo scambio di informazioni tra di essi. I concetti di base per la tolleranza dei guasti descritti in precedenza, in particolare, la scoperta e la diagnosi d'errore (due dei tre passi fondamentali del processamento d'errore), così come la diagnosi di guasto e la riconfigurazione (i due passi fondamentali del trattamento di guasto), hanno validità anche nel contesto di questi sistemi. Inoltre, è importante rilevare che un sistema distribuito ha delle qualità di ridondanza intrinseche nella sua struttura, data la replicazione dei nodi di elaborazione, che permettono quindi l'attuazione di una qualsiasi delle tecniche precedentemente descritte.

Ciò che realmente è più complesso nei sistemi distribuiti è la fase finale del trattamento di errore, ovvero la fase di recupero, questo perché lo stato globale del sistema risulta partizionato in stati locali dei singoli nodi di elaborazione, ed è possibile che esistano dipendenze tra i valori di stato di nodi distinti originate dallo scambio di informazioni. Dal punto di vista delle tecniche per la tolleranza dei guasti le dipendenze, dovute a scambio di informazione, introducono complicazioni notevoli, dato che esse rendono possibile il propagarsi degli errori.

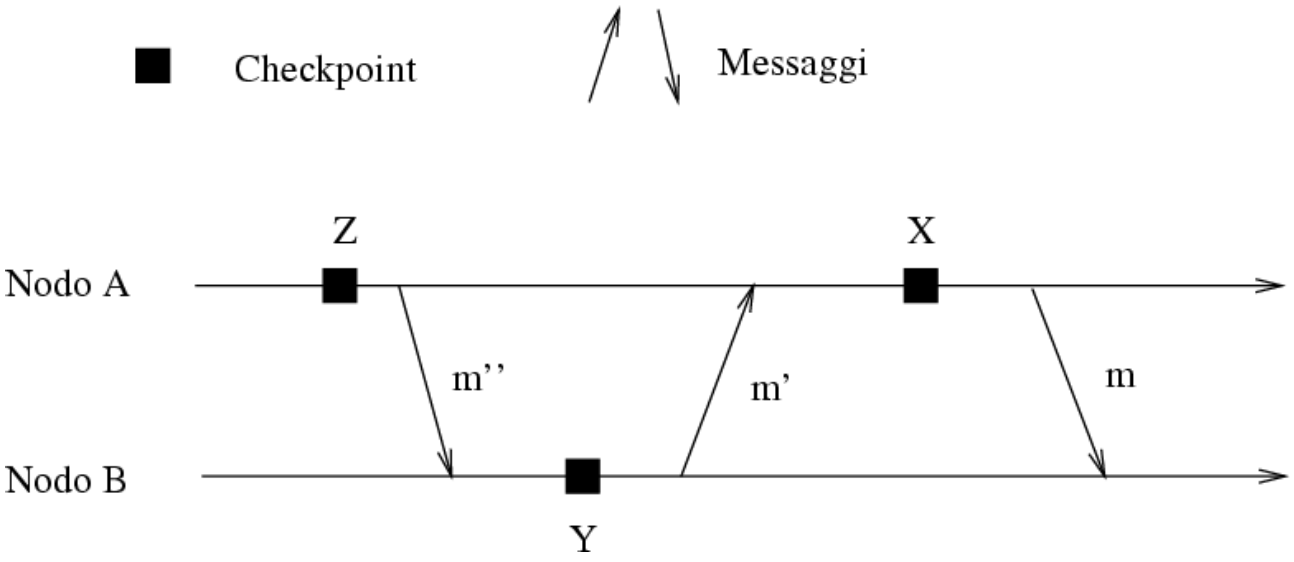

Fig. 5.8 - Propagazione d'errore in sistemi distribuiti.

Per far chiarezza su questa problematica si consideri l'esempio in figura 5.8, in cui è schematizzato un sistema costituito da due nodi di elaborazione A e B. Si supponga che, durante l'evoluzione della computazione, si verifichi un errore sul nodo A, il quale da origine poi alla spedizione del messaggio errato m verso il nodo B. All'atto della ricezione del messaggio e del processamento del suo contenuto anche lo stato del nodo B viene affetto dall'errore. Si supponga poi che l'errore venga rivelato sul nodo A, e che si tenti un processamento d'errore con ripristino all'indietro. Come spiegato in precedenza, questa soluzione è basata sui checkpoint (punti di recupero), ovvero registrazioni dello stato del sistema durante la sua evoluzione. Supponiamo che il nodo sul quale viene rivelato l'errore, ovvero A, attui un recupero all'indietro utilizzando il checkpoint X antecedente all'occorrenza dell'errore. In tal caso tutte le azioni sul nodo A successive al checkpoint X vengono revocate, inclusa la spedizione del messaggio m di trasferimento dati verso il nodo B. Ovviamente, la revoca del messaggio m implica che anche le transizioni di stato sul nodo B, avvenute successivamente alla ricezione di m, devono essere annullate. Per far questo si può attuare un ripristino all'indietro del nodo B al checkpoint Y. Quest'ultimo ripristino causa però la revoca della spedizione del messaggio m' verso il nodo A e quindi invalida anche le azioni del nodo A successive alla ricezione di m'. Questo implica che A dovrà effettuare un ulteriore ripristino all'indietro utilizzando il checkpoint Z. Questo implicherà la revoca del messaggio m'' che originerà un nuovo ripristino all'indietro del nodo B e così via. E' possibile quindi che si origini una sequenza indefinita di ripristini all'indietro causati da successivi annullamenti di spedizioni di messaggio, nota come *effetto-domino*, il quale può provocare il ripristino di ciascun nodo al proprio stato iniziale, perdendo quindi tutti i risultati della computazione prodotti fino alla rivelazione dell'errore.

L'effetto-domino trae origine dal fatto che esistono delle dipendenze tra i checkpoint dei singoli nodi dovute allo scambio di informazione trasportata dai messaggi. Nell'esempio precedente esiste una dipendenza tra X ed Y, dovuta al messaggio m; esiste una dipendenza tra Z ed Y, dovuta al messaggio m', e così via. Per evitare le dipendenze è necessario, quindi, adottare delle tecniche di coordinamento che facciano sì che i singoli nodi registrino il proprio stato in modo da prevenire dipendenze. Tipicamente questo viene realizzato tramite algoritmi distribuiti che usano o messaggi di controllo addizionali oppure informazione di controllo aggiunta sui messaggi propri dell'applicazione. Un algoritmo distribuito molto semplice basato sul secondo tipo di soluzione è il seguente: ciascun nodo mantiene una variabile locale denominata *timestamp* il cui valore iniziale è zero; il valore di questa variabile viene incrementato di uno ogni volta che il nodo decide di prendere un checkpoint; tale valore viene aggiunto come informazione di controllo su ciascun messaggio spedito; quando viene ricevuto un messaggio con informazione di controllo associata ad un valore superiore al timestamp locale del nodo ricevente, allora prima di processare il messaggio viene preso un checkpoint e viene portato il valore del timestamp locale al valore dell'informazione di controllo. Vediamo l'esempio di applicazione mostrato in figura 5.9. Il nodo A prende il checkpoint X e porta il suo timestamp al valore 2; questo valore viene poi aggiunto sul messaggio m. Quando il nodo B riceve m si accorge che il suo timestamp è inferiore a 2; questo induce B a prendere il checkpoint Y prima di processare m. In questo modo non esistono dipendenze tra X ed Y, quindi questi due checkpoint possono essere usati in un recupero all'indietro senza rischio di effetto-domino.

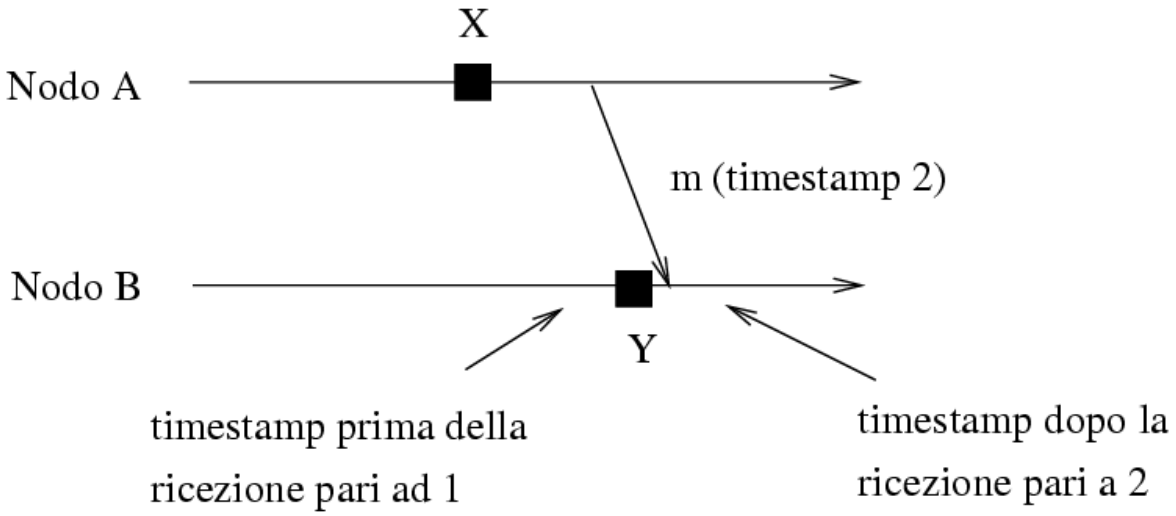

Fig. 5.9 - Un esempio di coordinamento tramite timestamp.

5.2. La rimozione dei guasti

La rimozione dei guasti è, insieme alla tolleranza dei guasti, un'altra tecnica per assicurare gli attributi della dependability. Essa consiste di tre passi fondamentali: verifica, diagnosi e correzione. La verifica è il processo di controllo che il sistema aderisca a delle proprietà comunemente denominate condizioni di verifica. Qualora queste proprietà non siano assicurate allora vengono attivati i due passi successivi. La verifica può essere di due tipi fondamentali:

- verifica di condizioni generali caratteristiche di specifiche classi di sistemi, per esempio l'assenza di deadlock;

- verifica di condizioni specifiche, ovvero strettamente associate alla specifica del sistema in oggetto.

Le tecniche di verifica si dividono poi anche in base al fatto che esse vengono effettuate con sistema attivo o meno. Verificare un sistema senza che esso sia attivo da luogo alle cosiddette *verifiche statiche*, le quali includono analisi di complessità o anche analisi formali. Invece, le verifiche attuate con il sistema attivo vengono dette *dinamiche*. In tal caso, esiste il problema della scelta delle configurazioni di ingresso al sistema da utilizzare nella fase di verifica, poiché verificare il sistema rispetto a tutte le possibili configurazioni di ingresso risulta, nella quasi totalità dei casi, una soluzione impraticabile. In genere le configurazioni di ingresso utilizzate nella verifica sono selezionate in base allo specifico obiettivo della verifica (ovvero se la verifica è strutturale o funzionale) ad anche in base ai tipi di guasto che possono verificarsi nei componenti in oggetto. Il problema di osservare i risultati di uscita della fase di verifica dinamica e di decidere se siano soddisfatte o meno le condizioni di verifica è comunemente noto come *problema dell'oracolo*. In genere, il processo di decisione è coadiuvato dall'uso della simulazione (si effettua quindi una comparazione tra i valori di uscita ottenuti e quelli originati da un simulatore del sistema), ed anche dall'uso di valori di uscita ottenuti dai cosiddetti "sistemi di riferimento", che sono sistemi precedentemente verificati e che non hanno manifestato guasti in fase di esercizio.

Se la fase di verifica evidenzia la presenza di guasti allora entrano in gioco le fasi di diagnosi e correzione del guasto, che devono avere la proprietà di non originare "regresso" nel sistema, ovvero effetti collaterali che possano dar poi luogo a guasti distinti da quello che si cerca di correggere.

E' da notare che la tecnica della rimozione dei guasti, qualora applicata durante il normale funzionamento del sistema viene comunemente denominata come *mantenimento correttivo*, che può essere sia di tipo curativo, ovvero tesa a rimuovere guasti che hanno realmente originato errori che siano stati rivelati, o di tipo preventivo, ovvero tesa a rimuovere i guasti prima che essi originino realmente degli errori.

5.3. La previsione dei guasti

La previsione dei guasti è la terza delle tecniche comunemente adottate per garantire gli attributi della dependability. Questa tecnica si basa sulla valutazione del comportamento dei sistemi in funzione dell'occorrenza di guasti e della loro attivazione. In altre parole, la modalità operativa della previsione dei guasti consiste nella valutazione dell'impatto dei guasti sugli attributi della dependability. La valutazione può essere di due tipi: *ordinale* e *probabilistico*. Nel caso di valutazione ordinale, l'obiettivo è quello di identificare e classificare i guasti. Nel caso di valutazione probabilistica, l'obiettivo è invece quello di stabilire in termini probabilistici il grado con cui i singoli attributi della dependability restano soddisfatti in presenza dei guasti. Esempi di metodologie per quest'ultimo obiettivo verranno trattate nel Capitolo seguente.

6. Tecniche di valutazione della dependability

Come descritto precedentemente, le qualità di un sistema di elaborazione possono essere valutate tramite le prestazioni e il livello di fiducia con cui esso soddisfa le specifiche di progetto. La valutazione del comportamento di un sistema può essere fatta in modo sperimentale o analitico. Nel primo caso si può far uso di un prototipo del sistema e le grandezze d'interesse sono stimate tramite dati statistici. Nel secondo caso tali grandezze sono ricavate direttamente da un modello matematico del sistema stesso.

Le valutazioni euristiche sono di gran lunga più costose e più complesse da ottenere rispetto a quelle basate su modelli matematici. E' da notare che alcune volte è impossibile avere a disposizione un prototipo del sistema e, comunque, pur avendolo, alcune misure d'interesse, come quelle della dependability, necessitano di lunghi tempi d'osservazioni, che rendono impossibile la valutazione. D'altro canto, i modelli matematici possono non rappresentare adeguatamente la struttura reale del sistema o il comportamento dei suoi componenti. In questi casi è possibile usare modelli simulativi che possono essere facilmente programmati ed eseguiti su sistemi di elaborazione. Anche in questo caso le grandezze possono essere stimate tramite valutazioni statistiche del comportamento del sistema. Inoltre la loro facilità d'uso li rende una valida alternativa alla valutazione analitica.

Di seguito si introdurranno alcuni approcci analitici per la valutazione quantitativa degli attributi più significativi della dependability: affidabilità (reliability), disponibilità (availability), sicurezza (safety) e performability. Prima di presentarli si introdurranno alcune definizioni basilari.

6.1. Nozioni basilari

La probabilità che un'entità si guasti per la prima volta nell'intervallo di tempo *(0,t)* è detta *funzione di guasto* e viene espressa con *Q(t)*, essa è una funzione di distribuzione cumulativa che gode delle seguenti proprietà:

$Q(t) = 0$ per $t = 0$

$0 \leq Q(t) \leq Q(t + \Delta t)$ per $\Delta t \geq 0$

$Q(t) = 1$ per $t = +\infty$

La probabilità che un'entità funzioni correttamente nell'intervallo di tempo *(0,t)*, detta *affidabilità*, viene espressa con *R(t)*, ed è pari a:

$R(t) = 1 - Q(t)$

e quindi:

$R(t) = 1$ per $t = 0$

$1 \geq R(t) \geq R(t + \Delta t)$ per $\Delta t \geq 0$

$R(t) = 0$ per $t = +\infty$

Se *Q(t)* è una funzione continua, allora esiste la sua derivata:

$$q(t) = \frac{dQ(t)}{dt}$$

ed è pari alla *densità di probabilità di guasto*, inoltre è continua anche *R(t)* e la sua derivata nel tempo *r(t)* è pari a:

$$r(t) = \frac{dR(t)}{dt} = \frac{d(1-Q(t))}{dt} = -\frac{dQ(t)}{dt} = -q(t)$$

Poiché le grandezze *R(t)* e *Q(t)* sono delle probabilità su di un singolo evento (entità sana, entità guasta) esse non si possono valutare analizzando la storia di una singola entità. In particolare è necessario studiare prima il comportamento di una popolazione sufficientemente grande d'entità dello stesso tipo e misurare la frequenza con cui si verificano i guasti, dopodiché si deve correlare tale frequenza con le suddette grandezze. In particolare, si supponga di avere una popolazione di *N* entità all'istante $t = 0$ e di osservare il numero d'entità sane nel tempo. Sia *n(t)* tale numero all'istante *t*, data la definizione di affidabilità abbiamo che:

$$n(t) = NR(t)$$

La frequenza media di guasti durante l'intervallo di tempo $(t, t + \Delta t)$ è pari a:

$$\frac{n(t)-n(t+\Delta t)}{\Delta t}$$

quindi ogni singola entità all'istante *t* sarà soggetta nell'intervallo *(t, t + Δt)* ad una frequenza media di guasto pari a:

$$\frac{1}{n(t)}\frac{n(t)-n(t+\Delta t)}{\Delta t}$$

Facendo tendere *Δt* a zero si ha che ogni singola entità è soggetta all'istante *t* ad una *frequenza istantanea di guasto* pari a:

$$h(t)=\lim_{\Delta t\to 0}\frac{1}{n(t)}\frac{n(t)-n(t+\Delta t)}{\Delta t}=\frac{1}{n(t)}\left(-\frac{dn(t)}{dt}\right)=$$

$$=\frac{1}{NR(t)}\left(-\frac{dNR(t)}{dt}\right)=-\frac{N}{NR(t)}\frac{dR(t)}{dt}=-\frac{dR(t)}{R(t)}\frac{1}{dt}$$

Notando che:

$$-h(t)dt=\frac{dR(t)}{R(t)}$$

si ha che integrando questa quantità, si ottiene l'affidabilità:

$$R(t)=e^{-\int_0^t h(\tau)d\tau}$$

Come già osservato, la frequenza di guasto può essere determinata sperimentalmente e quindi è possibile determinare *R(t)* o *Q(t)*. Mentre la frequenza di guasto di una singola entità può essere misurata sperimentalmente o, comunque, è possibile averne una stima estrapolando il comportamento d'entità similari, la determinazione della frequenza di guasto, e quindi dell'affidabilità, di un sistema complesso costituito da più entità distinte basandosi sull'osservazione sperimentale è molto costosa, se non addirittura impossibile. Per questo motivo nei paragrafi successivi s'introdurranno delle tecniche per la valutazione dell'affidabilità e di altre grandezze della dependability, conoscendo le caratteristiche di ogni singola entità.

Spesso è utile disporre di indici che possano dare un'idea immediata della qualità di un'entità; a tal fine si è rivelato utile il tempo medio di guasto o *MTTF* (Mean Time To Failure), formalmente definito come il valore atteso del tempo medio di manifestazione del guasto:

$$MTTF=\int_0^\infty tq(t)dt$$

che può anche essere espresso come:

$$MTTF = -\int_0^\infty t\frac{dR(t)}{dt}dt = -\left[tR(t)\right]_0^\infty + \int_0^\infty R(t)dt = \int_0^\infty R(t)dt$$

essendo

$$\lim_{t\to\infty} tR(t) = \lim_{t\to\infty} te^{-\int_0^\infty h(\tau)d\tau} = 0$$

dato che *h(t)* è costante o crescente con *t*.

Fig.6.1 – Andamento caratteristico della frequenza istantanea di guasto in funzione del tempo.

L'andamento caratteristico della frequenza istantanea di guasto è riportato in figura 6.1. La prima regione ha un andamento decrescente nel tempo, ciò è dovuto alla cosiddetta mortalità infantile delle entità; questo fenomeno è legato al fatto che durante i primi periodi di funzionamento dell'entità vengono messe in risalto le sue imperfezioni di fabbricazione. La seconda regione ha un andamento sostanzialmente uniforme, dovuta a guasti che si presentano casualmente ed indipendentemente dalle imperfezioni di fabbricazione, identificate nella prima regione. Infine la terza regione ha un andamento crescente nel tempo, dato che con l'invecchiamento è sempre maggiore la probabilità che si verifichino fenomeni irreversibili che danneggino le entità stesse. Poiché la prima regione la si può eliminare facendo un controllo preventivo delle entità da utilizzare, così come la terza regione, se si ipotizza di utilizzare le entità per un periodo di tempo inferiore al loro tempo di invecchiamento, allora è ragionevole supporre che la frequenza istantanea di guasto delle entità sia costante nel tempo, cioè:

$$h(t) = \lambda$$

Quindi, in queste ipotesi, le grandezze precedenti assumono le seguenti espressioni:

$$R(t) = e^{-\int_0^t h(\tau)d\tau} = e^{-\lambda t}$$

$$Q(t) = 1 - e^{-\lambda t}$$

$$r(t) = -\lambda e^{-\lambda t}$$

$$q(t) = \lambda e^{-\lambda t}$$

Fig.6.2 – Andamento della densità di probabilità di guasto in funzione del tempo.

In figura 6.2 è schematizzato l'andamento della densità di probabilità di guasto in funzione del tempo e le sue relazioni con la funzione di guasto e l'affidabilità.

Nel caso di sistemi riparabili, oltre all'evento "occorrenza di guasto" è necessario considerare l'evento riparazione o sostituzione di entità guaste. In tal caso il sistema è caratterizzato da periodi di buon funzionamento e da periodi d'inattività. Mentre i periodi di buon funzionamento dipendono dall'affidabilità delle singole entità del sistema, i periodi d'inattività dipendono dalla politica di intervento dei tecnici responsabili della manutenzione e dal tempo necessario per la riparazione o sostituzione delle entità guaste. Indicando con *MTTF* il tempo medio di guasto del sistema, e con *MTTR* (Mean Time To Repair) il tempo medio per riparare o sostituire una entità guasta, si definisce disponibilità a regime permanente (*Availability*) del sistema la quantità:

$$A = \frac{MTTF}{MTTF + MTTR}$$

Quindi, per migliorare la disponibilità di un sistema o s'interviene sulla sua affidabilità o sui tempi di riparazione.

La somma tra *MTTF* e *MTTR* è pari al tempo medio tra due malfunzionamenti o *MTBF* (Mean Time Between Failure), un altro indice molto usato per esprimere la qualità dei sistemi riparabili e già introdotto nei Capitoli precedenti. Molto spesso si confonde il *MTTF* con *MTBF*, ciò è dovuto al fatto che i valori dei due indici sono normalmente simili (perché di norma il valore di *MTTR* è piccolo rispetto a quello del *MTTF*), però mentre il primo indice si riferisce al periodo di tempo in cui il sistema è operante, il secondo tiene conto anche del periodo di tempo in cui il sistema è in manutenzione. Nelle considerazioni fatte si è supposto che il sistema sia visto in modo monolitico, successivamente faremo vedere come è possibile calcolare la disponibilità di un sistema complesso costituito di entità di cui si conoscono le caratteristiche di affidabilità e di disponibilità.

Un parametro molto importante nel progetto e nell'analisi dei sistemi tolleranti i guasti è il *fattore di copertura*, definito come la probabilità condizionata che, dopo il verificarsi di un guasto, il sistema torni a funzionare correttamente. Questo parametro da un indice della qualità del sistema a tollerare un guasto. Intuitivamente è la misura dell'abilità del sistema a rivelare un guasto, a localizzarlo, a contenerne le conseguenze e a recuperare uno stato consistente e libero da errori. Il problema fondamentale del fattore di copertura è che esso è molto difficile da valutare. Per la sua stima è necessario prevedere tutti i possibili guasti e, per ogni guasto identificato, è necessario stimarne la frequenza di verifica e il relativo fattore di copertura. Non solo è difficile identificare in modo puntuale la probabilità con cui si verifica un dato guasto, ma molto spesso non è neanche realistico riuscire a prevedere tutti i possibili guasti. Inoltre in genere il fattore di copertura è valutato considerando la presenza di un singolo guasto alla volta, mentre si dovrebbe tenere conto anche della possibilità che più guasti si possano verificare contemporaneamente. Inoltre il fattore di

copertura è tipicamente assunto costante nel tempo, mentre andrebbe valutato tenendo conto del suo andamento nel tempo; per esempio la probabilità di rivelare un guasto cresce all'aumentare del tempo dopo l'occorrenza del guasto stesso.

6.2. Valutazione dell'affidabilità e della disponibilità

La valutazione dell'affidabilità e della disponibilità di un sistema complesso costituito da entità di cui si conoscono le grandezze relative può essere fatta tramite diversi approcci analitici. Di seguito presenteremo quelli più comunemente utilizzati: la *modellizzazione combinatoria* e quella *markoviana*.