

# Appunti Integrativi su Sistemi Distribuiti

Corso di Reti di Calcolatori  
A.A. 2001/2002  
Baldoni, Marchetti, Tucci-Piergiovanni

# Indice

<b>1</b>	<b>Tecnologie</b>	<b>4</b>
1.1	Evoluzione nei Sistemi Distribuiti . . . . .	4
1.1.1	Architetture Distribuite Hardware: dalle SISD al cluster di PC . . . . .	7
1.1.2	Architetture Distribuite Software: dai terminali remoti ai sistemi completamente distribuiti . . . . .	9
1.1.3	Architettura di un Sistema Operativo: dai SO centralizzati al Middleware . . . . .	11
	Sistemi operativi di rete . . . . .	13
	Sistemi operativi distribuiti . . . . .	14
	Group Toolkit . . . . .	15
	Middleware . . . . .	15
<b>2</b>	<b>Modelli di Base</b>	<b>17</b>
2.1	Modello sincrono e asincrono . . . . .	18
2.1.1	Il Problema della Sincronizzazione dei Clock . . . . .	20
	Timestamping . . . . .	21
2.2	Guasti dei Processi . . . . .	24
2.3	Guasti della Comunicazione . . . . .	26
2.4	Topologia della Rete . . . . .	26
2.5	Determinismo contro Randomizzazione . . . . .	26
2.6	Problemi di Agreement . . . . .	27
2.6.1	Primitive di Broadcast . . . . .	27
	Reliable Broadcast . . . . .	27
	FIFO Reliable Broadcast . . . . .	28
	Causal Reliable Broadcast . . . . .	29
	Atomic Broadcast. . . . .	30
	FIFO Atomic Broadcast . . . . .	31
	Causal Atomic Broadcast . . . . .	31
2.6.2	Implementazioni delle Primitive di Broadcast . . . . .	32
	Reliable Broadcast con guasti crash . . . . .	32
	Causal reliable Broadcast in assenza di guasti . . . . .	33
2.6.3	Consenso . . . . .	35

2.6.4	Impossibilità del Consenso in Sistemi Asincroni . . . .	35
	Failure Detectors . . . . .	36
2.6.5	Algoritmi Di Consenso . . . . .	37
	Risolvere il Consenso usando $\mathcal{S}$ . . . . .	38
	Risolvere il Consenso usando $\diamond\mathcal{S}$ . . . . .	39
2.7	Il Problema della Replicazione . . . . .	42
2.7.1	Consistenza del Server Replicato . . . . .	42
	Criteri di Consistenza . . . . .	42
	Linearizzabilità . . . . .	43
2.7.2	Proprietà della Replicazione . . . . .	45
2.7.3	Tecniche di Replicazione a Due Livelli . . . . .	46
	Replicazione Primary-backup . . . . .	46
	Replicazione Attiva . . . . .	48
2.7.4	Group Toolkit . . . . .	49
	Group Communication e Replicazione Attiva . . . . .	50
	Group Communication e Replicazione Primary-backup . . . . .	51
2.7.5	Limiti della Replicazione a Due Livelli . . . . .	52

# Capitolo 1

## Tecnologie

### 1.1 Evoluzione nei Sistemi Distribuiti

Numerose sono state le definizioni di sistema distribuito date a partire dagli anni 70 ma nessuna di queste è pienamente soddisfacente. Ognuna risente delle tecnologie e della visione prevalente (hardware o software) all'epoca in cui è stata formulata. Ad esempio *“un sistema distribuito consiste di un insieme di calcolatori indipendenti che appaiono all'utente del sistema come un singolo calcolatore”* (Tanenbaum 1995) risente della astrazione di sistema operativo distribuito che andava per la maggiore all'epoca e di cui Tanenbaum era un fervido sostenitore<sup>1</sup>. Inoltre essa non coglie il fatto che un sistema distribuito può essere implementato al di sopra di un sistema operativo centralizzato multitasking (per esempio UNIX). In questa sezione noi adotteremo la seguente definizione più astratta:

*“un sistema distribuito è formato da un insieme di entità indipendenti che cooperano per raggiungere un fine comune e tale cooperazione si effettua attraverso scambio di messaggi”*

Se consideriamo le entità come CPU allora la definizione precedente corrisponde a quella di macchina MIMD (Multiple-Instruction-stream-Multiple-Data-stream) data da Flynn nel 1972. Se invece consideriamo le entità oggetti o processi allora troviamo la definizione di sistema distribuito data da Birman nel 1997.

Quindi esempi di sistema distribuito sono: sistemi ad oggetti cooperanti, sistemi client/server e reti di calcolatori. I primi due sono esempi di *sistemi distribuiti software*, l'ultimo rappresenta un *sistema distribuito hardware*.

---

<sup>1</sup>Ecco una serie di altre definizioni: “un sistema distribuito è composto da un insieme di calcolatori che eseguono qualche cosa insieme” (Schroeder in Mullender 1993), “un sistema distribuito è composto da una serie di siti indipendenti che si scambiano messaggi attraverso una rete di interconnessione” (Raynal 1992).

La definizione di Tanenbaum è quindi un caso particolare di quella utilizzata in questa sezione considerando il fine comune quello di fare apparire all'utente il sistema distribuito (insieme di calcolatori) come un singolo calcolatore. Tuttavia il fine comune può essere “la cooperazione tra sistemi per loro natura eterogenei”, “la condivisione di una risorsa”, “una interazione client/server”, “la stesura di un documento condiviso” ecc.

I vantaggi nell'utilizzare sistemi distribuiti hardware e software possono essere sintetizzati nei seguenti punti alcuni dei quali sono legati ai sistemi distribuiti hardware, altri a quelli software ed alcuni ad entrambi:

- **Affidabilità.** Questo è uno dei tipici vantaggi che si attribuisce ad un sistema distribuito. Ovvero la sua capacità di sopravvivere a un guasto di una sua entità (oggetto o processore) grazie alla sua ridondanza intrinseca. Questo, in realtà, è un mito da sfatare. Ci sono casi, *sistema distribuito hardware*, in cui questa capacità si può tradurre in modo semplice in un algoritmo operativo che permette alle entità non guaste di continuare l'elaborazione. Nel caso di *sistemi distribuiti software*, questa capacità diventa più difficile da tradurre, in modo immediato, in algoritmi pratici. In particolare questa difficoltà cresce con l'aumentare della autonomia delle entità. Nei sistemi distribuiti client/server, a seguito di un guasto di un server ci sono diverse politiche che si possono seguire che risultano da un compromesso tra la consistenza che si vuole assicurare al servizio e la velocità di ripristino. Nel caso di un sistema distribuito software in cui gli oggetti o i processi sono entità autonome che condividono uno stato comune, tale capacità è tutt'altro che semplice da tradurre in pratica e necessita di strumenti e servizi che utilizzano tecnologie ed algoritmi molto sofisticati.
- **Integrazione.** Un sistema distribuito assicura che se l'interfaccia delle entità con il sottosistema di comunicazione è standard, tali entità possono cooperare anche se realizzate con diverse tecnologie. Nel caso di sistemi distribuiti hardware, l'utilizzo della tecnologia *ethernet*, ad esempio, permette ad entità eterogenee (sistemi operativi e architetture hardware) di scambiarsi informazioni. Nel caso di sistemi distribuiti software, l'utilizzo di interfacce standard che racchiudono entità eterogenee permettono interazioni client/server tra sistemi software che utilizzano tecnologie incompatibili (*wrapping di legacy system*). La piattaforma *Web* e le piattaforme *middleware* sono solo alcuni degli strumenti che permettono tale integrazione.
- **Rapporto prezzo/prestazioni.** Un insieme di PC connessi da una LAN *ethernet* ha un prezzo di alcuni ordini di grandezza inferiore rispetto a quello di un mainframe e una capacità computazionale almeno paragonabile. Per quanto concerne un sistema distribuito software, le prestazioni sono principalmente da considerare in termini di

capacità di mettere in comunicazione tecnologie altrimenti incompatibili ad un costo molto basso rispetto a quello di dover abbandonare una tecnologia su cui è stato realizzato un certo investimento per migrare verso una nuova tecnologia più recente (Questo è un caso abbastanza raro nell'informatica).

A fronte dei vantaggi abbiamo anche una serie di svantaggi di cui dobbiamo tenere conto ed a cui si sta rispondendo con la nascita di nuove tecnologie che cercano di mitigare tale svantaggi:

- **Produzione di software.** Fino a pochi anni fa (1995), uno dei più seri problemi riguardo lo sviluppo dei sistemi distribuiti era dato dalla mancanza di strumenti software per la realizzazione di applicazioni che potessero mostrare al mercato l'importanza di tali sistemi. Da quel tempo sembra passato un secolo! Questa rivoluzione ha avuto tre passaggi importanti:
  - L'assunzione di TCP/IP (e della relativa interfaccia *socket*) come standard de facto ha definito una base fissa su cui gli sviluppatori di applicazioni si potevano appoggiare senza paura che la tecnologia al di sotto cambiasse ogni dieci minuti e li costringesse a reingegnerizzare le loro applicazioni.
  - L'avvento dell'architetture WEB e di tutte le tecnologie correlate (lato server) come HTML, CGI, servlet, php, shttp ecc. hanno permesso un salto di qualità alla produzione del software.
  - L'affermarsi del linguaggio java che permette a programmi residenti in altre macchine di essere trasferiti in una certo elaboratore e mandati in esecuzione. Tale esecuzione avviene all'interno dell'elaboratore logicamente al di sopra di un ambiente "protetto" chiamato macchina virtuale.
- **Sicurezza.** Nel momento stesso che l'informazione viene decentrata e che c'è un flusso di informazioni sulla rete, emerge il problema della sicurezza. Quando esistevano solo sistemi centralizzati, la sicurezza era per lo più fisica, legata cioè alla prevenzione da accessi indebiti al locale dove erano immagazzinati i dati da proteggere. Oltre a questa sicurezza adesso ci dobbiamo confrontare con problematiche di *sicurezza della comunicazione*, sicurezza rispetto ad accessi indesiderati ai dati attraverso la rete, sicurezza da attacchi che rendono non fruibili dall'esterno servizi critici di una azienda (WEB, Email etc), protezione da programmi che vengono scaricati dalla rete e mandati in esecuzione locale. Questo ha generato una esplosione tale di metodologie e tecnologie da meritare una sezione a parte all'interno del manuale.

- **Comunicazione.** Con lo sviluppo dei sistemi distribuiti, e con l'aumentare degli utenti che utilizzano questi mezzi, c'è la necessità di aumentare le bande trasmissive, di migliorare la qualità del servizio offerta e di fatturare tali servizi. Tecnologie come ATM servono per rispondere al primo punto. Per quanto riguarda gli altri due siamo ancora in una fase di studio poiché non è facile affrontare problemi che non erano stati mai presi in considerazione durante la prima fase dell'evoluzione dei sistemi distribuiti.

Di seguito mostreremo come l'evoluzione dei sistemi distribuiti si sia tradotta in una evoluzione a tutti i livelli architetturali: dall'hardware, al software, ai sistemi operativi, ai linguaggi di programmazione fino alle applicazioni.

### 1.1.1 Architetture Distribuite Hardware: dalle SISD al cluster di PC

Classificare le architetture hardware è un lavoro complesso e non sempre univoco. A seconda dei fattori di classificazione considerati, alcune architetture possono ricadere in una categoria piuttosto che in un'altra. Questo perché non esiste una netta separazione tra diverse classi di elaboratori. Quindi la classificazione deve essere necessariamente basata su fattori più generali. Una delle classificazioni più citate è quella di Flynn (1972) essa si basa sui due flussi di informazioni normalmente presenti nei calcolatori:

- flusso istruzione
- flusso dati

Un elaboratore che ha un solo flusso dati ed un solo flusso istruzioni, come la macchina di Von Neumann, è una macchina Single-Instruction-stream-Single-Data-stream (SISD). In questa categoria ricadono tutte le macchine con una singola Unità Centrale come i Personal Computer, le workstation e i mainframe <sup>2</sup>. Se l'elaboratore prevede più flussi dati ed un singolo flusso istruzioni siamo in presenza di un elaboratore Single-Instruction-stream - Multiple-Data-stream (SIMD). Un elaboratore SIMD permette di eseguire la stessa istruzione in parallelo su un insieme di dati, quindi è particolarmente adatto per realizzare calcoli intensivi vettoriali e matriciali. Elaboratori di questa categoria sono i Vector Processor, Array Processor, alcuni supercomputer e l'italiano APE. Elaboratori che eseguono più istruzioni sullo stesso flusso dati sono denominati Multiple-Instruction-stream-Single-Data-stream

---

<sup>2</sup>Dire che un elaboratore appartiene alla categoria SISD non significa dire che all'interno della CPU non possano essere implementate tecniche di parallelizzazione delle operazioni (come ad esempio tecniche pre-fetching e pipelining), tecniche di parallelizzazione del flusso dati, o processori che eseguono calcoli "ad-hoc" (come ad esempio processori di canale e co-processori matematici).

(MISD). Questa schematizzazione, che di fatto ben rappresenta la nozione di pipelining, non ha però prodotto elaboratori a livello commerciale. Un elaboratore Multiple-Instruction-stream-Multiple-Data-stream (MIMD), infine, è costituito da più Unità Centrali indipendenti che operano su flussi dati che possono essere diversi. Una classificazione più precisa delle macchine MIMD si ottiene facendo riferimento a come è suddivisa la memoria fisica. Macchine MIMD a memoria fisica condivisa e macchine MIMD a memoria privata. Le prime sono anche conosciute con il nome di multiprocessor, mentre le seconde con quello di multicomputer. La Figura 1.1 mostra l'evoluzione delle architetture hardware nel tempo.

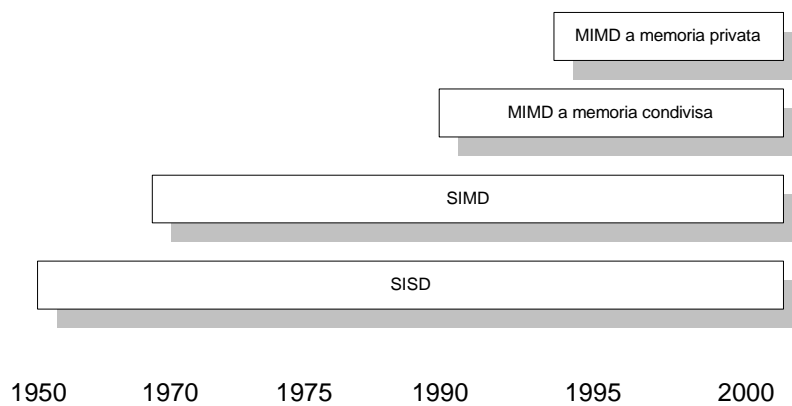


Figura 1.1: Evoluzione delle architetture hardware nel tempo

Multicomputer e multiprocessor possono essere ancora classificati in funzione di come le Unità Centrali sono interconnesse tra loro. L'interconnessione può essere effettuata attraverso un unico mezzo fisico condiviso (Bus, Anello <sup>3</sup>) o attraverso un canale diretto fisico tra coppie di Unità Centrali (vedi Figura 1.2)

Esempi di Macchine MIMD a memoria condivisa basate su canali diretti sono la *Encore* e la *Sequent*. Esempi di macchine MIMD a memoria condivisa basate su canale condiviso sono la *RP3* e la *Ultracomputer*. Le reti di *Personal Computer* (PC) basata su *LAN* sono quindi elaboratori MIMD a memoria privata con interconnessione su mezzo condiviso. Mentre la *hypercube* e le reti di *transputer* sono esempi di multicomputer basati su scambio dati su canali diretti. Recentemente, sono stati introdotti sul mercato sistemi (*crossbar switch*) che permettono l'interconnessione basata su canale diretto anche per gruppi di PC (Rete *Myrinet*). Questo sistema di elaboratori è anche conosciuto come cluster di PC. Un cluster di PC differisce da una rete di PC principalmente (i) per la velocità del trasferimento dati

<sup>3</sup>Ad un livello di astrazione più elevato anche una rete locale o geografica possono essere viste come mezzo condiviso tra due Unità Operative.

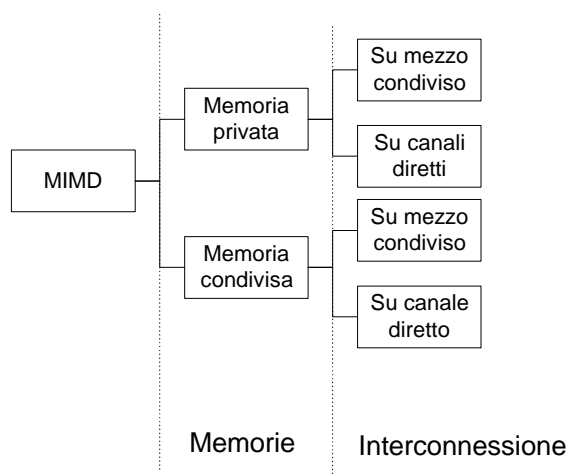


Figura 1.2: Classificazione degli elaboratori MIMD

(oltre 1Gbit/sec), (ii) per la centralizzazione fisica (tutti i PC sono montati sullo stesso rack) e (iii) per la presenza di una applicazione di management, residente su un singolo PC, che permette di lanciare processi su altri PC, monitorare il loro comportamento ecc. Quindi in base alla nostra definizione e considerando le Unità Centrali come entità, un sistema distribuito di tipo hardware corrisponde all'insieme delle macchine MIMD a memoria privata.

### 1.1.2 Architetture Distribuite Software: dai terminali remoti ai sistemi completamente distribuiti

Negli anni le architetture distribuite software hanno avuto un forte cambiamento che ha seguito (ed a volte anticipato) l'evoluzione delle architetture hardware e di quelle dei sistemi operativi. Le architetture che noi esaminiamo sono le seguenti: terminali remoti, architetture client-server, architetture cooperative, architetture web-centric e architetture completamente distribuite. La Figura 1.3 mostra una evoluzione temporale di tali architetture.

**Architettura a terminali remoti.** In questa architettura esiste una entità centrale che esegue tutte le operazioni e delle entità remote che interagiscono con l'entità centrale. Queste entità remote sono di fatto molto semplici (ovvero non hanno capacità computazionale) e si limitano a visualizzare ed inviare/ricevere le informazioni alla/dalla entità centrale. Tale entità possiede una parte privata che si occupa delle computazioni necessarie ad ogni singola entità remota. Questo rende necessariamente il sistema omogeneo dal punto di vista hardware e software.

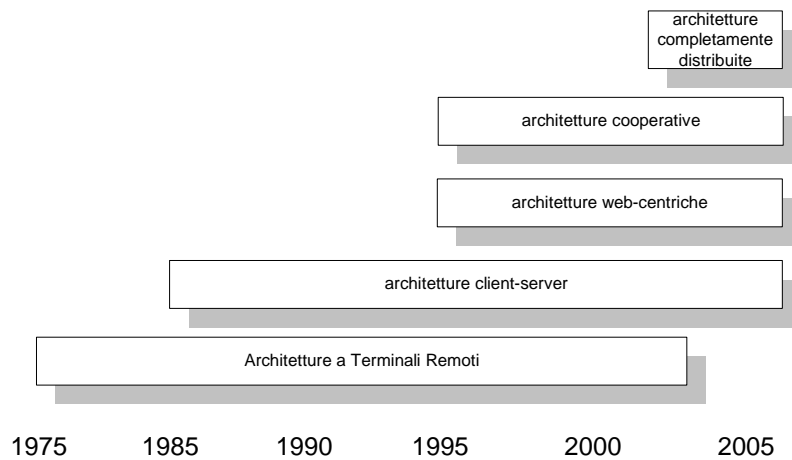


Figura 1.3: Evoluzione delle architetture distribuite software

**Architettura client/server.** Nel paradigma Client/Server i processi applicativi vengono suddivisi in fruitori di servizi (processi client) e fornitori di servizi (processi server). Una interazione tra client e server si svolge nel modo seguente: un client invia una richiesta di servizio al server, il server esegue il servizio ed infine invia una replica al client. In tale sistema un client coopera con altri client attraverso uno o più server che permettono la coordinazione e la condivisione dei dati. Da notare che più server possono fornire lo stesso servizio, che un servizio può essere implementato attraverso la cooperazione di più server e che un server può anche comportarsi come un client fruendo di servizi di altri server. A differenza dell'architettura precedente i client hanno una capacità computazionale propria. L'interazione è, almeno teoricamente, indipendente dal tipo di hardware, sistema operativo e meccanismi base di comunicazione utilizzati dai processi (o oggetti). Questa proprietà permette a client e server di tipo non-omogeneo (hardware e/o software di sistema differenti) di interagire.

**Architettura cooperativa.** Tale architettura rappresenta di fatto l'evoluzione delle architetture client/server e si basa su entità autonome che esportano e richiedono servizi. In particolare questa architettura mira l'evoluzione della società umana, a partire da servizi forniti da entità di basso livello si formano attraverso un processo di integrazione servizi di più alto livello (service integration). Affinché questo processo sia efficace deve essere standardizzata l'interfaccia di tali entità ovvero il modo con cui queste esportano servizi e le modalità per richiederli. Esempi di standardizzazione sono ODP (Open Distributed Processes) e CORBA (Common

Object Request Broker Architecture). Si rimanda alla sezione “Architetture Client/Server” per maggiori dettagli.

**Architettura WEB-Centric.** Negli ultimi anni con l’esplosione del WEB si è riscontrato un curioso ritorno a applicazioni di tipo centralizzato, non troppo diverso da quelle che utilizzavano terminali remoti per quanto concerne il potere computazionale del client. Infatti una architettura WEB-centric mette una entità (WEB server) al centro di un sistema distribuito e le entità client vi accedono per ottenere dei servizi. La stragrande maggioranza di questi servizi sono di fatto eseguiti lato server e vengono restituiti solo i risultati ai client. In questo senso il potere computazionale dei client è molto basso.

**Architettura completamente distribuita.** A differenza della architettura cooperativa e della architettura client/server, nelle architetture completamente distribuite l’astrazione di servizio è realizzata attraverso la cooperazione di entità paritetiche (gruppi) che dialogano usando servizi di comunicazione uno-a-uno o uno-a-molti (multicast) che assicurano diversi gradi di ordinamento ai messaggi (totale, causale, fifo) in funzione delle necessità imposte dalla applicazione. L’uso di primitive uno-a-molti e i ritardi imprevedibili (e distinti) dei canali di comunicazione porta il sistema ad avere una mancanza di informazione globale coerente. Ogni entità percepisce il sistema in modo incompleto. Questo rende arduo (o impossibile) raggiungere accordi tra le entità cooperanti. Il raggiungimento di tali accordi è alla base di servizi fondamentali per le applicazioni come sarà evidenziato nella seconda parte di questa sezione. Tali servizi includono la gestione coerente di dati replicati, la gestione della ridondanza software e della scoperta dei guasti.

### 1.1.3 Architettura di un Sistema Operativo: dai SO centralizzati al Middleware

In modo semplice possiamo dire che un sistema operativo locale è un insieme di moduli software che si interpone tra il software applicativo ed il firmware dell’hardware con l’obiettivo di schermare l’utente (che usa i programmi applicativi), attraverso un processo di virtualizzazione, da quello che è l’hardware sottostante<sup>4</sup>. Questo implica che il sistema operativo deve provvedere

---

<sup>4</sup>Da notare come questa definizione non sia particolarmente rigida. Questo ha portato, con il passare degli anni, a variazioni sostanziali di ciò che è un sistema operativo. Infatti il software applicativo continua a stratificarsi. Quindi sempre più software che ieri era applicativo, domani con molta probabilità sarà integrato all’interno del sistema operativo poiché utilizzato da altri software applicativi sovrastanti. Ad esempio il prodotto Network File System di Sun era una applicazione sul sistema operativo Sun OS. Nella nuova versione del sistema operativo Sun (cfr. Solaris), NFS è diventato parte del sistema operativo.

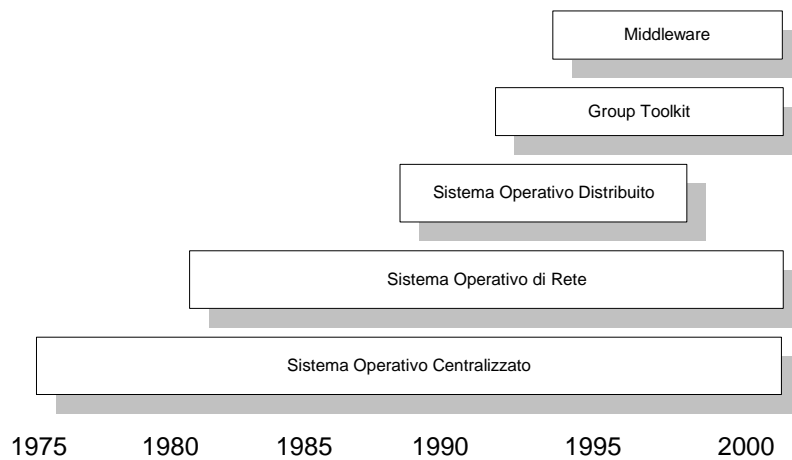


Figura 1.4: Evoluzione delle architetture di un sistema operativo

alla gestione di elementi chiave di una architettura hardware come il processore, i processi, la memoria, l'Input/Output e i file. Con la proliferazione dei personal computer connessi in rete, i sistemi operativi hanno dovuto fare fronte a nuove esigenze legate alla distribuzione come: interoperabilità, trasparenza, autonomia e affidabilità. Questo ha portato in rapida successione alla nascita di *Sistemi Operativi di Rete*, *Sistemi Operativi Distribuiti*, *Group Toolkit* e *Middleware* come mostrato in Figura 1.4.

I sistemi operativi di rete aggiungono ai sistemi operativi centralizzati quei moduli software che permettono la condivisione delle risorse remote. I sistemi operativi distribuiti hanno come obiettivo primario quello della trasparenza unito alla virtualizzazione dei sistemi operativi centralizzati, ovvero una rete di elaboratori vista da un utente come un singolo elaboratore. I sistemi groupware permettono di lavorare a livello di gruppo di processi piuttosto che di singolo processo attraverso una serie di servizi di sistema che rendono particolarmente agevoli la gestione di politiche di affidabilità e di consistenza all'interno del gruppo. Il middleware è una evoluzione dei sistemi operativi distribuiti. Si tratta di una serie di servizi, più leggeri di quelli messi a disposizione da un sistema operativo distribuito, orientati alla cooperazione di entità remote. Obiettivi e caratteristiche delle diverse architetture dei sistemi operativi sono riassunte nella tabella seguente:

<b>Architettura</b>	<b>Caratteristiche Principali</b>	<b>Obiettivi</b>
<i>Sistema operativo centralizzato</i>	Gestione dei processi Comunicazioni tra processi locali Gestione della memoria Gestione Input/Output Gestione del file system	virtualizzazione
<i>Sistema operativo di rete</i>	Comunicazione tra processi remoti Scambio di informazioni remote Ricerca di informazioni remote	Interoperabilità a livello di trasporto OSI (TCP/IP)
<i>Sistema Operativo distribuito</i>	Visione di un insieme di elaboratori connessi in rete come un singolo elaboratore. Visione globale: del file system, dello spazio dei nomi, del tempo, della sicurezza e della potenza computazionale. In alcuni sistemi visione globale anche della memoria.	Trasparenza (interoperabilità a livello di trasporto opzionale)
<i>Group toolkit</i>	Visione di un insieme di processi (gruppo) come una unica entità. Comunicazione tra entità uno-a-molti, gestione dei membri del gruppo.	Affidabilità (attraverso ridondanza software)
<i>Middleware</i>	Visione di una applicazione distribuita come un insieme di entità cooperanti. Comunicazione tra entità remote (uno-a-uno, uno-a-molti, molti-a-molti), servizi per la localizzazione di una entità, servizi per la scoperta delle operazioni eseguibili da una entità, servizi per la sincronizzazione temporale, per la sicurezza etc. (vedi sezione client/server)	Autonomia e cooperazione (interoperabilità a livello di sistemi operativi locali)

Un sistema operativo centralizzato commerciale in realtà ha incluso, nel corso degli anni, alcune funzionalità degli altri sistemi rivelatesi particolarmente utili. Per esempio, la visione globale del file system dai sistemi operativi distribuiti, la ricerca di informazioni remote, la posta elettronica, l'emulazione di terminale dai sistemi operativi di rete e le comunicazioni multicast dai sistemi groupware. Quindi il sistema operativo commerciale ha usufruito di tutti gli studi e le esperienze fatte nei diversi sistemi specifici come illustrato in Figura 1.5.

### **Sistemi operativi di rete**

Il primo obiettivo di un sistema operativo di rete è quello di aggiungere moduli software ai sistemi operativi centralizzati per poter permettere la condivisione delle risorse (programmi, dati, periferiche) presenti nella rete. Questi moduli software devono, come mattone base, permettere una interoperabilità a livello di scambio messaggi tra i processi presenti negli elaboratori della rete. Quindi questo mattone base deve essere utilizzato per realizzare servizi chiave per la condivisione delle risorse tra i processi. Poiché, per quanto concerne le comunicazioni, l'astrazione richiesta dai sistemi operativi di rete è quella *processo-a-processo*, ne segue che un sistema operativo

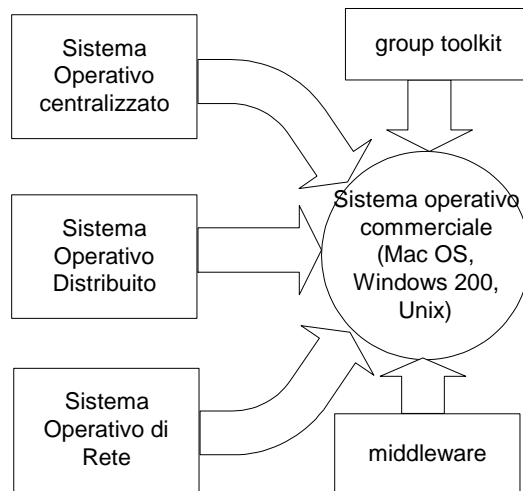


Figura 1.5: Evoluzione di un sistema operativo commerciale

di rete deve includere un livello di trasporto e alcuni servizi creati su questo livello. TCP/IP si è imposto come livello di trasporto standard. Al di sopra di TCP/IP si sono sviluppati negli ultimi anni numerosi servizi di rete, tra cui *remote login*, *file transfer*, *email*, *network browsing* e *remote execution*. Molti di questi sono oramai parte integrante dei sistemi operativi locali.

### Sistemi operativi distribuiti

A cavallo tra gli anni ottanta e novanta c'è stata una grossa spinta, nell'ambito della ricerca, nel campo dei sistemi operativi distribuiti. Questi di fatto esportavano la nozione di trasparenza, tipica dei sistemi operativi centralizzati, alle reti di calcolatori. Di fatto il concetto di trasparenza si associava alla nozione di locazione (dove un programma andrà realmente in esecuzione) e di parallelismo (su quanti processori il programma andrà in esecuzione). I punti cardine di questa ricerca erano: paradigmi di comunicazione, migrazione di codice, coordinamento, gestione di risorse distribuite, protocolli di consistenza di dati replicati e implementazione di algoritmi distribuiti. Lo sviluppo di prototipi seguì due strade distinte: ad-hoc e off-the-shelf. La prima prevedeva la costruzione di sistemi operativi distribuiti direttamente al di sopra del firmware. La seconda al di sopra dei sistemi operativi di rete commerciali. Lo studio imponente fatto in quegli anni ha portato alla definizione di una linea di frontiera che indica quello che si può ragionevolmente implementare su una rete di calcolatori in modo trasparente ad un utente. Da questi studi nacquero i sistemi groupware ed il middleware.

## Group Toolkit

Mentre i sistemi operativi distribuiti hanno come loro obiettivo finale quello di prendere un programma scritto per un sistema operativo centralizzato e di mandarlo in esecuzione al di sopra di una rete di calcolatori, un group toolkit assume che esistono diversi programmi che cooperano attraverso scambio messaggi esplicito avendo come obiettivo finale quello della affidabilità a fronte di guasti che avvengono all'interno del sistema e della trasparenza della singola entità rispetto alla nozione di gruppo. Queste proprietà vengono raggiunte attraverso l'uso di replicazione software. Gruppi di processi/oggetti che eseguono un particolare paradigma di cooperazione (per esempio, replicazione attiva, replicazione passiva, master/slave ecc.). Poiché l'entità base in questi sistemi diventa il gruppo si devono realizzare servizi che possano essere utilizzati dai componenti di un gruppo per mantenere un forte grado di consistenza dei dati al suo interno. Da notare che un gruppo può possedere uno stato e che questo stato è condiviso da tutti i membri. Quindi anche se lo stato visto ad un certo istante di tempo da due membri di un gruppo può essere distinto, essi devono però essere consistenti. I servizi di base sono:

- definizione dei membri attuali del gruppo (*group membership*);
- comunicazione multicast (*one-to-many*) imponendo diversi gradi di ordinamento ai messaggi (*atomic, causal, fifo e reliable multicast*);
- rilevamento dei guasti;
- base dei tempi comune.

Per un approfondimento dei servizi sopraelencati si rimanda alla seconda parte di questa sezione. Da questo approfondimento risulterà chiara la difficoltà di raggiungere l'obiettivo di realizzare applicazioni affidabili al di sopra di una rete di elaboratori. Stanti queste difficoltà, il mercato si sta orientando verso la realizzazione di applicazioni distribuite a basso grado di affidabilità basate su interazione uno-a-uno. Le applicazioni basate su WEB sono un chiaro esempio in questo senso. Tuttavia quando in futuro si presenterà la necessità di realizzare applicazioni con piccoli tempi di caduta del servizio e ripristini veloci, la soluzione basata su gruppi di processi tornerà inevitabilmente ad essere una soluzione appetibile.

## Middleware

In questi sistemi ogni entità coopera autonomamente esportando e richiedendo servizi. L'idea di base è quella di mantenere un certo grado di trasparenza senza richiedere la trasparenza totale tipica dei sistemi operativi distribuiti. Le entità sono distinte e ben riconoscibili e possono essere eterogenee. Il

ruolo del middleware è quindi quello di connettere queste entità in modo standard. In particolare, esso consiste di un insieme di servizi che permettono a più entità residenti su uno o più elaboratori, di interagire attraverso una rete di interconnessione a dispetto di differenze nei protocolli di comunicazione, architetture dei sistemi locali, sistemi operativi ecc. Tale software si interpone negli elaboratori locali tra le applicazioni ed il sistema operativo locale creando quindi una architettura a tre livelli. Un middleware include una serie di funzionalità tra cui:

- Ambiente di sviluppo applicativo.
- Servizio di Comunicazione.
- Servizi di astrazione e cooperazione. Questi servizi rappresentano il cuore del middleware e comprendono tra gli altri:
  - Directory Service
  - Security Service.
  - Time Service.
- Servizi per le applicazioni.
- Servizi di amministrazione del sistema.

Lo scopo principale di middleware è di aiutare a risolvere problemi di interoperabilità (a livello di sistema operativo) e di connettività riscontrati da applicazioni che devono lavorare su piattaforme distribuite. In questo contesto sono destinati a diventare il collante delle applicazioni utilizzate all'interno di una azienda.

## Capitolo 2

# Modelli di Base

Un modello di sistema è una semplificazione della realtà in cui i problemi devono essere considerati. Un modello per un oggetto è una collezione di attributi e un insieme di regole che governano come questi attributi interagiscono. Un modello si dice *accurato* se si avvicina molto alla realtà dell'oggetto, un modello si dice *trattabile* se è possibile procedere all'analisi dell'oggetto. Un buon modello dovrebbe essere accurato e trattabile al tempo stesso: gli attributi selezionati devono essere solo quelli che influenzano in modo significativo il fenomeno in esame. In assoluto non esiste un buon modello: un modello può essere *adeguato* al tipo di problema e ambiente considerato.

Un sistema distribuito comprende più processi che comunicano su canali a banda stretta e ad alta latenza, con alcuni processi o canali che possono guastarsi. Avere più processi significa avere più potere computazionale ma richiede che essi siano coordinati.

I sistemi distribuiti sono modellati come un insieme di processi  $\Pi \equiv \{p_1, p_2, \dots, p_n\}$  che interagiscono scambiando messaggi tramite canali di comunicazione. Esistono una quantità di modelli che si distinguono tra loro per le restrizioni più o meno forti imposte sui componenti (processi e canali di comunicazione) del sistema. Dato un certo problema in un sistema distribuito la soluzione si raggiunge attraverso un opportuno protocollo (cioè un insieme di opportuni messaggi scambiati tra i processi). Il protocollo, dato un certo problema, è dipendente dal modello di sistema adottato: in generale meno restrizioni si assumono nel modello tanto più un protocollo (se esiste) è complesso.

Quando si descrive un modello di sistema vengono considerati i seguenti parametri: la sincronia dei processi e delle comunicazioni, i tipi di guasti dei processi e della comunicazione, la topologia della rete e se i processi sono deterministici o sono processi randomizzati.

## 2.1 Modello sincrono e asincrono

La sincronia di un modello è un attributo relativo alle assunzioni di timing relative al comportamento di processi e canali di comunicazione. In generale, esistono due grandi categorie di sistemi distribuiti, mappate in due modelli distinti: il modello sincrono e il modello asincrono.

**Modello Sincrono.** In un sistema sincrono, vengono limitati sia gli scostamenti delle velocità di esecuzione dei processi sia i ritardi di consegna dei messaggi, il che consente di ridurre il grado di non-determinismo del sistema e di facilitare la progettazione di applicazioni distribuite. Più formalmente diciamo che un sistema è *sincrono* se soddisfa le seguenti proprietà:

- Esiste un upper bound conosciuto  $\delta$  sul ritardo dei messaggi; consiste nel tempo necessario per mandare, trasportare e ricevere un messaggio su un canale.
- Ogni processo  $p$  ha un clock locale  $C_p$  con un drift rate  $\rho \geq 0$  conosciuto e limitato rispetto al tempo reale. Quindi per tutti i  $p$  e per tutti gli istanti di tempo  $t > t'$ ,

$$(1 + \rho)^{-1} \leq \frac{C_p(t) - C_p(t')}{(t - t')} \leq (1 + \rho)$$

Dove  $C_p(t)$  è la lettura del clock  $C_p$  all'istante di tempo reale  $t$ .

- Esistono upper bound conosciuti sul tempo richiesto da un processo per eseguire un passo elementare.

In un sistema sincrono è possibile misurare i time-out dei messaggi, ciò fornisce un meccanismo per la rilevazione di guasti. Inoltre è possibile implementare *clock approssimativamente sincronizzati*, cioè clock che, oltre ad avere un drift rate limitato, soddisfano anche la seguente condizione: esiste un  $\epsilon$  t.c. per ogni  $t$  presi due qualsiasi processi  $p$  e  $q$  vale  $|C_p(t) - C_q(t)| \leq \epsilon$  [21, 7].

**Modello Asincrono.** In un sistema *asincrono*, al contrario del modello sincrono, i ritardi associati alla consegna dei messaggi non sono superiormente limitati a priori, così come la velocità relativa dei processi del sistema. Quindi un sistema è asincrono se non esiste alcun bound su (i) ritardo dei messaggi, (ii) drift del clock, o (iii) tempo richiesto da un processo per eseguire un passo elementare. In realtà assumere che un sistema sia asincrono è una non-assunzione. Ogni sistema è asincrono: anche un sistema in cui i processi girano in lock-step e in cui la consegna dei messaggi è istantanea soddisfa la definizione di sistema asincrono, quindi un sistema sincrono è un

caso particolare di sistema asincrono. Questo modello è quello più usato per diverse ragioni: ha una semantica semplice; un protocollo pensato per girare in un sistema asincrono può essere usato per girare in qualsiasi sistema distribuito; un sistema distribuito realizzato attraverso l'infrastruttura di rete e sistemi operativi commerciali ad oggi più comuni (per esempio Internet e Windows 2000/NT/98) è riconducibile ad un modello asincrono. Infatti i protocolli della suite IP: UDP e TCP non sono in grado di limitare superiormente il tempo di consegna di un pacchetto e i sistemi operativi commerciali non sono anch'essi in grado di assicurare scostamenti limitati nelle velocità dei processi in esecuzione. Inoltre i sistemi distribuiti aperti, come i sistemi che coprono grandi aree geografiche, o sistemi soggetti a workload variabili e inaspettati imposti dai loro utenti sono fondamentalmente asincroni (in questi sistemi le assunzioni sincrone diventano al massimo probabilistiche).

**Modelli Parzialmente Sincroni.** Sebbene il modello asincrono sia un modello molto attraente perchè non vengono fatte assunzioni di timing <sup>1</sup>, sfortunatamente molti problemi di base nell'ambito dei sistemi distribuiti non possono essere risolti in sistemi asincroni. Per questo motivo sono stati studiati modelli che si pongono tra il modello sincrono e quello asincrono (vedi Fig. 2.1). Questi modelli intermedi sono chiamati sistemi *parzialmente sincroni*. I sistemi parzialmente sincroni si distinguono tra loro a seconda del livello di sincronia assunto. Ad esempio i processi possono avere velocità limitata e clock perfettamente sincronizzati ( $\epsilon = 0$ ) ma ritardo di trasmissione dei messaggi non limitato [10], oppure il ritardo dei messaggi è limitato ma sconosciuto [11].

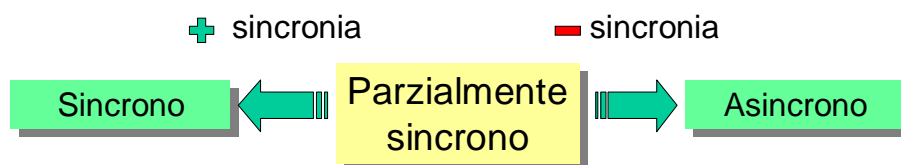


Figura 2.1: Relazione tre Modello Sincrono, Asincrono, Parzialmente Sincrono

<sup>1</sup>Progettare un'applicazione distribuita per un sistema asincrono risulta un vantaggio dal punto di vista del programmatore che potrà disegnare l'applicazione essendo all'oscuro delle reali caratteristiche di sincronia del sistema sottostante: dei tempi di ritardo dei messaggi, del drift rate dei clock, dell' esecuzione di un passo elementare da parte dei processi.

### 2.1.1 Il Problema della Sincronizzazione dei Clock

Abbiamo visto che in un sistema sincrono (o parzialmente sincrono) esistono delle assunzioni relative alla sincronizzazione dei clock. In un sistema distribuito i valori dei clock degli host del sistema possono differire sia a causa della fase, sia a causa della differente velocità. In alcune applicazioni (ad esempio real-time) occorre risolvere due importanti problemi: come sincronizzare tra loro i differenti clock degli host appartenenti al sistema e come sincronizzarli col tempo reale.

Per risolvere il primo problema si possono adottare algoritmi di tipo master-slave o di gossiping. Nel primo caso un processo master invia il proprio valore di clock ai suoi slave e questi ultimi rilevano se i loro clock sono sincronizzati ed in caso contrario aggiornano il valore del proprio clock. Nel secondo ogni processo invia il proprio valore di clock a tutti (o ad un sottoinsieme di tutti processi in gioco) ad intervalli regolari. In questo modo ogni processo può mantenere limitato lo scostamento tra il suo clock e quello degli altri. In entrambi i casi il risultato è un modello astratto fornito di un valore di clock comune a tutti processi caratterizzato da due parametri: la granularità del clock  $G$  (differenza tra due clock tick) e la precisione  $\Pi$  (distanza massima tra i tick in due processi relativi allo stesso valore di clock globale), come illustrato in Figura 2.2. Sistemi distribuiti su rete locale riescono a raggiungere granularità nell'ordine dei centinaia di microsecondi. In area geografica si riesce ad avere una granularità nell'ordine dei millisecondi. Qualora la connessione con il tempo fisico rivesta un ruolo rilevante nella computazione, ovvero sussista la necessità di sincronizzare i clock non solo tra di essi, ma anche rispetto al tempo fisico è possibile sincronizzare i clock di una (o di tutti) stazione con il tempo fisico.

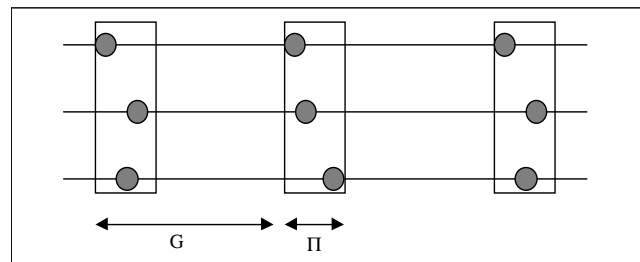


Figura 2.2: Granularità  $G$  e precisione  $\Pi$  in un sistema a clock sincronizzati

Per ottenere ciò si può dotare la stazione o di un ricevitore GPS o di un altro ricevitore a radiofrequenza che restituisca al processo i valori temporali forniti da un servizio di timing riconosciuto a livello internazionale come l'UTC (Universal Coordinated Time) fornito via radio dal National Institute of Standard Time (NIST).

## Timestamping

Nei sistemi asincroni la rete di comunicazione non è sufficientemente affidabile e/o veloce da consentire la sincronizzazione dei clock fisici degli host appartenenti al sistema. Inoltre non sempre è necessario sincronizzare tra loro o col tempo reale i clock degli host del sistema: in un database distribuito, ad esempio, per effettuare il controllo della concorrenza basato su timestamp tra transazioni distribuite è sufficiente stabilire una relazione di ordinamento tra esse, e per stabilire tale relazione non vi è alcuna necessità di condividere una base dei tempi comune. Molto spesso infatti il tempo è utilizzato per ordinare tra loro eventi, ed in tale contesto è sufficiente assegnare ad ogni evento un *timestamp*, ovvero un intero, utilizzando quindi un *tempo logico*. Un insieme di eventi generato dai processi di un sistema distribuito può essere ordinato secondo la relazione cosiddetta *happened before* (o di *precedenza causale*): dati due eventi  $A$  e  $B$ , diciamo che  $A$  precede  $B$ , indicandolo con  $A \rightarrow B$  se:

- $A$  e  $B$  appartengono allo stesso processo e  $A$  accade prima di  $B$ ;
- $A$  e  $B$  appartengono invece a processi distinti,  $A$  è l'evento di invio di un messaggio e  $B$  l'evento di ricezione di tale messaggio (un messaggio non può essere ricevuto prima della sua spedizione, e la sua trasmissione impiega un tempo non nullo);
- se esiste un evento  $C$  tale che  $A \rightarrow B$  e  $B \rightarrow C$  allora  $A \rightarrow C$ .

Dati due eventi  $A$  e  $B$  se  $\neg(A \rightarrow B)$  e  $\neg(B \rightarrow A)$ , i due eventi sono detti *concorrenti*.

In Figura 2.3, ad esempio, si ha ad esempio che  $e_3^2 \rightarrow e_2^2$ ,  $e_2^3 \rightarrow e_1^4$  e quindi che  $e_3^2 \rightarrow e_1^3$ , mentre gli eventi  $e_1^2$  e  $e_3^3$  sono concorrenti.

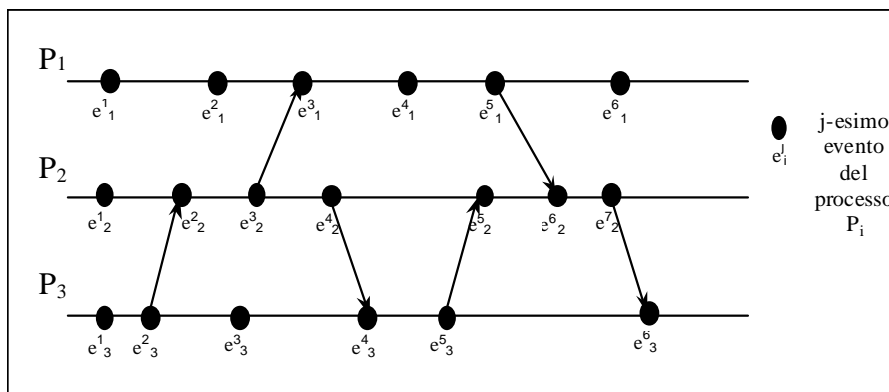


Figura 2.3: Eventi di una computazione distribuita

**Timestamping Scalare.** Come si possono associare timestamp agli eventi di un sistema distribuito in modo che se un evento  $A$  precede causalmente un altro evento  $B$  ( $A \rightarrow B$ ), allora il timestamp di  $A$  sia minore del timestamp di  $B$ ? Lamport ha dato una soluzione molto generale a tale problema, valida in un modello di sistema distribuito asincrono con canali di comunicazione punto-punto. In particolare, a ogni evento  $E$  del sistema viene associato un timestamp  $C(E)$ , tale che:

se  $A \rightarrow B$  allora  $C(A) < C(B)$  (relazione di timestamping scalare)

Per ottenere ciò è sufficiente che ogni processo  $P_i$  mantenga un contatore  $C_i$  inizializzato a 0 e rispetti le seguenti regole di aggiornamento:

1. quando  $P_i$  processa un evento, prima incrementa il contatore  $C_i$  di una unità ( $C_i := C_i + 1$ ) e quindi associa un timestamp  $T_i$  all'evento il cui valore è pari al valore corrente di  $C_i$ ;
2. quando  $P_i$  invia un messaggio, esegue l'evento di trasmissione e allega al messaggio il timestamp  $T_i$  associato a tale evento ricavato dalla regola 1;
3. quando a  $P_i$  arriva un messaggio  $m$  con timestamp  $T$ , esso pone  $C_i = \max(C_i, T)$  e quindi esegue l'evento di ricezione del messaggio (regola 1).

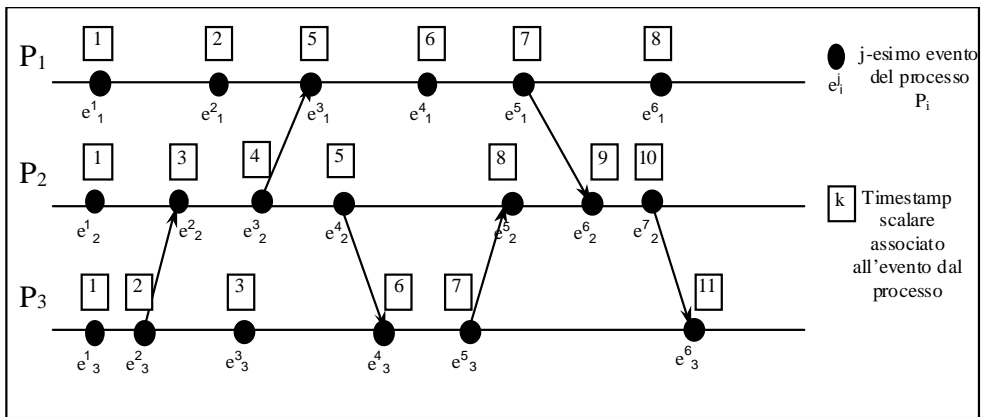


Figura 2.4: Timestamping scalare

In Figura 2.4 sono illustrati i timestamp associati dai processi agli eventi, seguendo le regole espote, nel caso della computazione già illustrata in Figura 2.3. Si noti che se  $A$  e  $B$  sono due eventi concorrenti, utilizzando il protocollo descritto i loro timestamp potranno essere identici o indifferentemente l'uno maggiore dell'altro (come accade, ad esempio, per gli eventi

$e_1^3$  ed  $e_2^4$  di Figura 2.4). In alcune applicazioni occorre tuttavia stabilire un ordine totale su tutti gli eventi, per esempio in un database distribuito con controllo di concorrenza basato su timestamp è necessario assegnare timestamp diversi a tutte le transazioni del sistema e tali timestamp devono formare un ordine totale. Per ottenere ciò, è sufficiente modificare il protocollo appena proposto aggiungendo in coda al timestamp di ogni evento il PID (Process Identifier) del processo in cui l'evento ha luogo. Quindi se due eventi occorrono nel sistema al tempo logico 25 nei processi  $P_1$  e  $P_2$ , allora il primo avrà un timestamp uguale a 25.1 ed il secondo 25.2. Con questo strumento è possibile assegnare timestamp differenti a tutti gli eventi significativi di sistema, rispettando la relazione di timestamping scalare.

**Timestamping vettoriale.** Il timestamping scalare non caratterizza la relazione di precedenza causale. Infatti, come abbiamo visto, per capire se due eventi sono concorrenti non possiamo guardare il loro timestamp scalare. Quindi Mattern nel 1988 ha introdotto la nozione di *vector clock*, che caratterizza la relazione di causalità. Un sistema di vector clock è formato da  $n$  vettori di interi  $Vc$  ad  $n$  componenti, uno per ogni processo. La componente  $Vc_i[x]$  indica il numero di eventi del processo  $P_x$  osservati dal processo  $P_i$ . In particolare, ogni processo  $P_i$  con  $i \in [1, \dots, n]$  gestisce un vettore di interi  $Vc_i[1 \dots n]$  (inizializzato a  $[-, -, \dots, 0, \dots, -]$ ) in base alle seguenti regole:

1. quando  $P_i$  processa un evento  $e$ ,  $Vc_i[i]$  si incrementa di una unità e quindi associa un timestamp  $T_e$  all'evento  $e$  il cui valore è pari al valore corrente di  $Vc_i$ ;
2. quando  $P_i$  esegue un evento di trasmissione di un messaggio, egli allega al messaggio il timestamp di quell'evento ottenuto dalla regola 1;
3. quando arriva un messaggio a  $P_i$  da  $P_j$  con allegato un timestamp  $T$ ,  $P_i$  esegue la seguente operazione:  $\forall x \in [1, \dots, n] : Vc_i[x] := \mathbf{max}(Vc_j[x], T[x])$ , quindi esegue l'evento di ricezione (esegue la regola 1).

Dati due eventi  $A$  e  $B$ , un sistema di vector clock garantisce la seguente proprietà:

$$A \rightarrow B \text{ se e solo se } T_A < T_B \text{ (relazione di timestamping vettoriale)}$$

dove  $Tc < Tc'$  se e solo se  $\forall x \in [1, \dots, n] : Tc'[x] \geq Tc \wedge \exists x \in [1, \dots, n] : Tc'[x] > Tc[x]$ .

In Figura 2.5 sono illustrati i timestamp associati dai processi agli eventi di sistema rispettando le regole appena esposte. Da notare che due eventi  $A$  e  $B$  sono concorrenti (come ad esempio gli eventi  $e_1^3$  ed  $e_2^4$ ) se e solo se  $\neg(Vc_{i,A} < Vc_{j,B})$  e  $\neg(Vc_{j,B} < Vc_{i,A})$ . Catturare la relazione di causalità è importante in molti contesti applicativi quali debugging di applicazioni

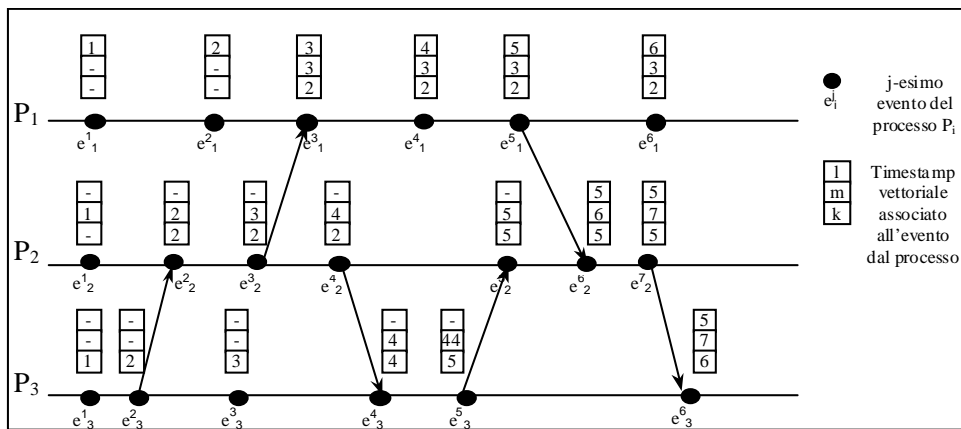


Figura 2.5: Timestamping vettoriale

distribuite, sistemi distribuiti tolleranti ai guasti ecc. dove è importante capire quali eventi possano avere causato altri.

## 2.2 Guasti dei Processi

Un processo è *guasto* in un'esecuzione se il suo comportamento devia da quello prescritto dall'algoritmo che sta eseguendo; altrimenti il processo è *corretto*. Un *failure model* specifica in quale modo un processo guasto può deviare dal suo algoritmo. I *failure model* sono i seguenti [17]:

- *Crash*: un processo guasto termina prematuramente e non esegue alcuna azione da questo punto in poi. Tuttavia prima di fermarsi si è comportato in modo corretto.
- *Send omission*: un processo guasto termina prematuramente, o omette in modo intermittente di inviare dei messaggi che era supposto inviare, o entrambe le cose.
- *Receive omission*: un processo guasto termina prematuramente, o omette in modo intermittente di ricevere dei messaggi che gli sono stati inviati, o entrambe le cose.
- *General omission*: un processo guasto è soggetto a guasti di tipo send omission o receive omission, o entrambi.
- *Arbitrary* (alle volte chiamato *Byzantine* o *malicious*): un processo guasto può esibire qualsiasi comportamento. Ad esempio cambia stato in modo arbitrario.

- *Arbitrary con message authentication*: un processo guasto può esibire qualsiasi comportamento ma è disponibile un meccanismo per l'autenticazione dei messaggi usando *unforgeable signature*. Un processo soggetto ad un guasto arbitrario può asserire di aver ricevuto un particolare messaggio da un processo corretto, anche se non l'ha mai ricevuto. Un meccanismo di autenticazione dei messaggi permette agli altri processi corretti di validare o meno questa asserzione.

I modelli sopra menzionati possono essere applicati sia a sistemi sincroni che asincroni. Tuttavia esistono altri tipi di guasti che sono applicabili solo a sistemi sincroni:

- *Timing failure*: un processo soggetto ad un timing failure è un processo che viola una delle assunzioni di sincronia. Un simile processo può guastarsi in uno o più dei seguenti modi:
  1. commette un general omission failure;
  2. il suo clock locale eccede il bound specificato (clock failure);
  3. viola il bound relativo al tempo richiesto per eseguire un passo elementare (performance failure).

Questi failure model possono essere classificati in termini di severità. Un modello *A* è più *severo* di un modello *B* se l'insieme di comportamenti di guasto permessi da *B* è un sottoinsieme proprio di quello costituito dai comportamenti di guasto permessi da *A* (Figura 2.6). Quindi un algoritmo che tollera guasti di tipo *A* tollera anche guasti di tipo *B*. I guasti di tipo arbitrario sono i più severi mentre i guasti di tipo crash sono i meno severi.

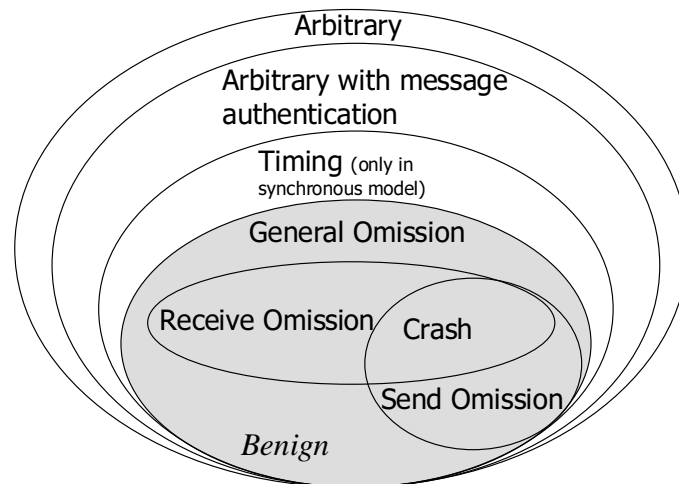


Figura 2.6: Classificazione dei failure model

I *timing failures* sono più severi dei *general omission* e meno severi degli arbitrari con *message authentication*.

Tutti i tipi di guasti che non sono più severi dei *timing* sono chiamati *benign*.

Un sistema è detto *f-fault tolerant* se continua a soddisfare le sue specifiche anche quando  $f$  componenti (in questo caso i processi) sono guasti.

## 2.3 Guasti della Comunicazione

I canali di comunicazione possono essere soggetti ai seguenti tipi di guasti[17]:

- *Crash*: un canale guasto smette di trasportare messaggi. Prima del crash si è comportato in modo corretto.
- *Omission*: un canale guasto omette in modo intermittente di trasportare i messaggi inviati attraverso di esso.
- *Arbitrary* (alle volte chiamato *Byzantine* o *malicious*): un canale guasto può esibire un qualsiasi comportamento. Per esempio può generare messaggi spuri.

Nel caso di sistemi sincroni, si hanno anche:

- *Timing failures*: un canale guasto trasporta messaggi in modo più veloce o più lento rispetto alle sue specifiche.

## 2.4 Topologia della Rete

La rete di comunicazione può essere modellata come un grafo, dove i nodi sono i processi e gli archi sono i canali di comunicazione tra i processi. Questo modello si adatta sia a reti punto-punto che a reti broadcast. Assunzioni precise sulla topologia di rete possono essere fatte se necessarie quando vengono considerati particolari problemi.

## 2.5 Determinismo contro Randomizzazione

Il comportamento di un processo può essere *deterministico* o *randomizzato*. In generale un processo può essere modellato come un automa a stati (anche infinito). La relazione di transizione di stato di un processo deterministico determina in modo *unico* lo stato risultante dall'esecuzione di ogni passo elementare sullo stato corrente. In un processo randomizzato l'esecuzione di un dato passo elementare sullo stato corrente porta in uno

stato facente parte di un insieme di possibili stati, ed ognuna di tali transizioni ha associata una certa probabilità. Informalmente un processo tira una moneta per determinare quale transizione scegliere. La solvibilità di molti problemi è fortemente influenzata dalla caratteristica dei processi di essere deterministici o meno.

## 2.6 Problemi di Agreement

In un sistema distribuito più processi cooperano per il raggiungimento di un obiettivo comune. Questa cooperazione implica che i processi debbano coordinarsi tra di loro. Da qui nascono i problemi di *agreement*: i processi che devono coordinarsi devono “accordarsi” in modo tale da operare tutti la *stessa* scelta. Ad esempio ci sono problemi in cui i processi devono accordarsi per quello che riguarda l’insieme dei messaggi da consegnare, oppure l’ordine in cui questi messaggi debbono essere consegnati. Nel seguito della sezione esamineremo i problemi più comuni che si riscontrano nell’ambito dei sistemi distribuiti.

### 2.6.1 Primitive di Broadcast

Si consideri un sistema distribuito in cui i processi comunichino tramite broadcast. Le primitive fornite dalla rete di comunicazione per inviare e ricevere messaggi di solito non danno garanzie di consegna in presenza di guasti. In particolare se un processo manda in broadcast un messaggio e incorre un guasto durante il broadcast dello stesso, la primitiva di invio offerta dalla rete non garantisce che tutti i processi a cui il messaggio era destinato lo consegnino. Quindi se dei *guasti incorrono durante un broadcast*, è *plausibile che solo un sottoinsieme dei processi consegnino il messaggio precedentemente inviato*. Questa inconsistenza può compromettere l’integrità di un sistema distribuito, quindi tali broadcast *unreliable* non sono tool appropriati per costruire applicazioni tolleranti ai guasti. In pratica è opportuno specificare primitive di comunicazione che soddisfino garanzie di consegna anche in presenza di guasti. Allo scopo di avere primitive di broadcast utili a costruire applicazioni tolleranti ai guasti vengono specificati broadcast con semantiche più forti. In Figura 2.7 sono mostrate le relazioni tra le primitive di Broadcast di seguito presentate.

#### Reliable Broadcast

Informalmente un reliable broadcast garantisce tre proprietà (i) tutti i processi corretti si accordano (fanno agreement) sull’insieme dei messaggi da consegnare, (ii) tutti i messaggi inviati in broadcast da processi corretti sono consegnati, (iii) nessun messaggio spurio viene mai consegnato.

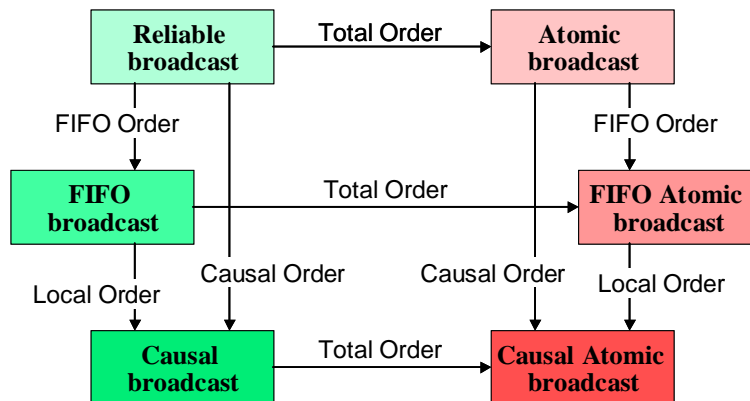


Figura 2.7: Relazioni tra primitive di broadcast

L'idea è che un messaggio o viene consegnato da tutti i processi o da nessun processo.

Formalmente il Reliable Broadcast è definito in termini di due primitive:  $R\text{-broadcast}(m)$  quando un processo invia in broadcast un messaggio  $m$  e  $R\text{-deliver}(m)$  quando un processo consegna un messaggio  $m$ .

La specifica del Reliable Broadcast è la seguente[17] :

- *Validity*: Se un processo corretto invia in broadcast un messaggio  $m$ , allora tutti i processi corretti alla fine consegnano  $m$ .
- *Agreement*: Se un processo corretto consegna un messaggio  $m$ , allora tutti i processi corretti alla fine consegnano  $m$ .
- *Integrity*: Per qualsiasi messaggio  $m$ , ogni processo corretto consegna  $m$  al più una volta, e solo se  $m$  è stato precedentemente inviato in broadcast da un processo mittente.

Si noti che le prime due proprietà sono proprietà di *liveness*, mentre la terza rappresenta una proprietà di *safety*.

### FIFO Reliable Broadcast

Nel Reliable Broadcast, non c'è alcun requisito sull'ordine in cui i messaggi vengono consegnati, "l'accordo", infatti riguarda solo l'insieme dei messaggi da consegnare, poi ogni processo può consegnare questo insieme in un ordine arbitrario. In certe applicazioni ciò può portare ad "anomalie". Si consideri il seguente esempio: una compagnia aerea mette a disposizione degli utenti un server di prenotazioni voli, questo server invia agli utenti interessati la notizia che c'è stato un aumento di tariffa dei voli del 15%. Nel frattempo un certo utente sta prenotando un volo. Ricevuto però il messaggio di aumento delle tariffe decide di cancellare la prenotazione. Quindi i messaggi

“in gioco” sono tre: “Prenota volo”, “Cancella prenotazione” (inviati dall’utente) e “Aumento del 15%” (inviato dal server). La specifica del Reliable Broadcast garantisce “solo” che tutti e tre i messaggi vengano consegnati sia dal processo utente che dal server. Questo significa che si potrebbe verificare lo scenario mostrato in Figura 2.8. In questo caso il server riceve la cancellazione della prenotazione prima di aver registrato la prenotazione stessa!!

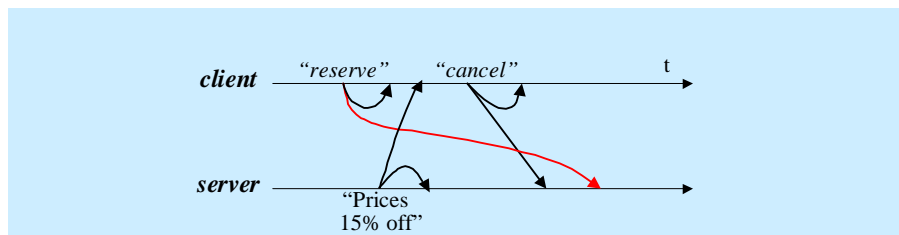


Figura 2.8: Anomalia data da un indesiderato ordine di consegna

Quello che si vorrebbe garantire in questo scenario è che il server riceva i messaggi nell’ordine in cui sono stati mandati dal processo utente. Per assicurare ciò introduciamo un’altra primitiva di broadcast: FIFO Reliable Broadcast.

Formalmente, si ha FIFO Reliable Broadcast quando un Reliable Broadcast soddisfa anche la seguente proprietà:

- *FIFO Order*: Se un processo invia in broadcast un messaggio  $m$  prima di un messaggio  $m'$ , allora nessun processo corretto consegna  $m'$  a meno che non abbia precedentemente consegnato  $m$ .

### Causal Reliable Broadcast

La proprietà di FIFO order è adeguata quando un messaggio  $m$  dipende solo dai messaggi precedentemente inviati dallo stesso mittente. Tuttavia se un messaggio  $m$  dipende anche dai messaggi che il mittente ha consegnato prima del broadcast di  $m$ , l’ordine di consegna stabilito dal FIFO order non è sufficiente. L’esempio in Figura 2.9 mostra come il FIFO order non sia sufficiente quando l’invio di un messaggio dipende dai messaggi consegnati prima. In particolare il messaggio  $m_2 =$ “C’è un party giovedì” dello studente 1 *dipende* dalla consegna (eseguita dallo studente 1) del messaggio  $m_1 =$ “L’esame di Venerdì è stato cancellato” inviato dal professore. Lo studente 2 però consegna prima il messaggio  $m_2$  (si noti che il FIFO order non è stato violato perchè  $m_1$  ed  $m_2$  provengono da due mittenti diversi), quindi ancora non sa che l’esame è stato cancellato. Quello che si vorrebbe garantire in questo scenario è che lo studente 2 consegni  $m_1$  ed  $m_2$

in ordine. A questo scopo introduciamo una primitiva di broadcast chiamata *Causal Broadcast*, basata sulla proprietà di Causal Order.

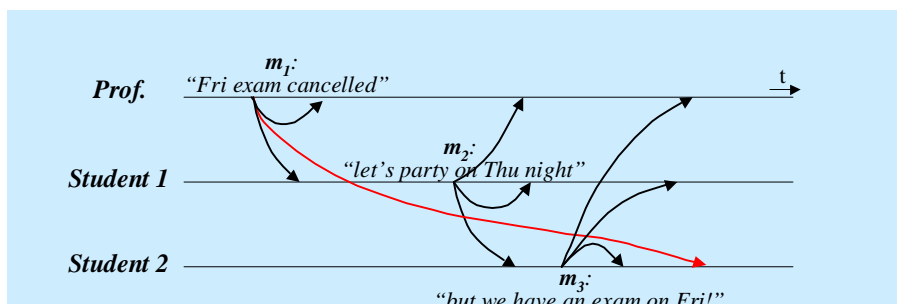


Figura 2.9: Anomalia data da un indesiderato ordine di consegna

Il Causal Order è basato sulla relazione di “accaduto prima” definita da Lamport [19], denotata da “ $\rightarrow$ ”. Siano  $e_i$  ed  $e_j$  due eventi in un sistema distribuito. Diciamo che un evento  $e_i$  precede causalmente  $e_j$  ( $e_i \rightarrow e_j$ ) se e solo se: (i) un processo esegue  $e_i$  prima di  $e_j$ , oppure (ii)  $e_j$  è il broadcast di qualche messaggio  $m$  ed  $e_i$  è la consegna di  $m$ , oppure (iii) esiste un evento  $e_k$  tale che  $e_i \rightarrow e_k$  e  $e_k \rightarrow e_j$ . Un Causal Reliable Broadcast è un Reliable Broadcast che soddisfa la seguente proprietà:

- *Causal Order*: Se il broadcast di un messaggio  $m$  precede causalmente il broadcast di un messaggio  $m'$ , allora nessun processo corretto consegnerà  $m'$  a meno che non abbia precedentemente consegnato  $m$ .

Si noti che avendo a disposizione un FIFO Broadcast si può realizzare un Causal Broadcast soddisfacendo la seguente proprietà di Local Order:

- *Local Order*: Se un processo consegna un messaggio  $m$  prima di inviare in broadcast un messaggio  $m'$ , allora nessun processo corretto consegna  $m'$  a meno che non abbia precedentemente consegnato  $m$ .

### Atomic Broadcast.

Le primitive viste fin qui non assicurano l’assenza di anomalie. Si consideri un conto bancario replicato tramite due repliche  $r_1$  ed  $r_2$ . Le repliche cambiano stato tramite degli eventi di update. In Figura 2.10 sono mostrati due update: “Aumenta del 10% l’interesse” e “deposito di 20\$”. Se i due update non vengono consegnati nello stesso ordine dalle due repliche le repliche diventano *inconsistenti*. Allo scopo di mantenere le due copie identiche introduciamo un’altra primitiva di broadcast: *Atomic Broadcast* o *Total Order Broadcast*.

L’Atomic Broadcast richiede che tutti i processi corretti consegnino tutti i messaggi nel medesimo ordine. Questo Total Order sulla consegna dei

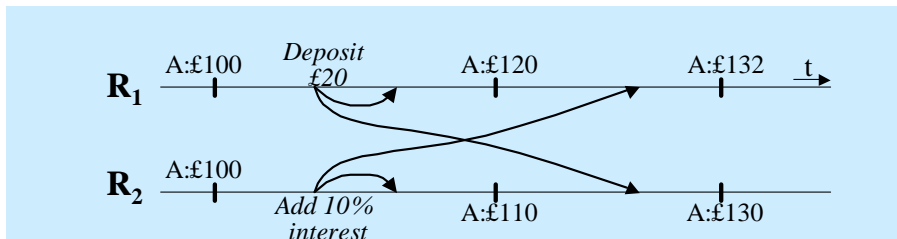


Figura 2.10: Anomalia data da un indesiderato ordine di consegna

messaggi assicura che tutti i processi corretti abbiano la stessa “view” del sistema, quindi possono agire in modo consistente senza altra comunicazione aggiuntiva. Formalmente l’Atomic Broadcast è un Reliable Broadcast che soddisfa la seguente proprietà [17]:

- *Total Order*: Se due processi corretti  $p$  e  $q$  consegnano entrambi  $m$  ed  $m'$ , allora  $p$  consegna  $m$  prima di  $m'$  se e solo se  $q$  consegna  $m$  prima di  $m'$ .

Si noti che la proprietà di Total Order è una proprietà *ortogonale* alle proprietà di FIFO Order e Causal Order. Ciò significa che il Total Order non è una proprietà più forte rispetto alle altre due, quindi se si rispetta il Total Order non è detto che si stia rispettando il FIFO o il Causal Order e viceversa.

### FIFO Atomic Broadcast

L’Atomic Broadcast non richiede che i messaggi siano consegnati in FIFO Order. Per esempio l’Atomic Broadcast permetterebbe il seguente scenario: un processo soffre di un guasto transiente durante il broadcast di un messaggio  $m$ , quindi invia in broadcast un messaggio  $m'$  e i processi corretti consegnano solo  $m'$ . Questo scenario viola la proprietà di FIFO Order.

Quindi un FIFO Atomic Broadcast è un Reliable Broadcast che soddisfa sia la proprietà di Total Order che quella di FIFO order.

### Causal Atomic Broadcast

Il FIFO Atomic Broadcast non richiede che i messaggi vengano consegnati in Causal Order. Ad esempio il FIFO Atomic Broadcast permetterebbe il seguente scenario: un utente guasto A manda in broadcast un articolo; un utente guasto B, che è l’unico a consegnare tale articolo, manda in broadcast un messaggio contenente un commento all’articolo e immediatamente dopo fa crash (prima di consegnare il suo stesso messaggio). Un utente corretto

C consegna il messaggio contenente il commento all'articolo, sebbene non consegnerà mai l'articolo originale. Ciò non soddisfa il Causal Order.

Quindi un Causal Atomic Broadcast è un Reliable Broadcast che soddisfa sia la proprietà di Total Order che quella di Causal Order.

Si noti che il Causal Atomic Broadcast è più forte del FIFO Atomic Broadcast.

## 2.6.2 Implementazioni delle Primitive di Broadcast

In questa sezione illustreremo delle implementazioni di alcune primitive di broadcast. In particolare daremo l'implementazione del Reliable Broadcast in caso di guasti di tipo crash e l'implementazione del Causal Reliable Broadcast in assenza di guasti.

### Reliable Broadcast con guasti crash

Un qualsiasi processo  $P_i$  che vuole inviare un messaggio  $m$  in modo che venga rispettata la specifica del Reliable Broadcast (vedi Sez. 2.6.1) esegue lo pseudo-codice in Figura 2.11. Si noti che la specifica è soddisfatta solo sotto l'assunzione di guasti di tipo crash e di canali di comunicazione che non si guastano mai.

Si supponga inoltre che le primitive di comunicazione, chiamate **send** e **recv**, fornite dalla rete di comunicazione, soddisfino le seguenti proprietà:

- *Safety*. Il processo  $q$  riceve il messaggio  $m$  dal processo  $p$  al più una volta e solo se  $p$  ha precedentemente inviato  $m$  a  $q$ .
- *Liveness*. Se  $p$  invia  $m$  a  $q$  e  $p, q$  sono corretti, allora alla fine  $q$  riceve  $m$  da  $p$ .

```

R-BROADCAST( $m, P_{dest} = \langle P_1, \dots, P_n \rangle$ )
1  send( $m$ ) to  $P_{dest}$ ;

R-DELIVER
1  when ( $P_i$  recv( $m$ ) da  $P_j$ ) do
2    if ( $P_i$  non ha già consegnato  $m$ )
3      then if ( $i \neq j$ )
4        then send( $m$ ) to  $P_{dest}$ ;
5        consegna( $m$ )

```

Figura 2.11: Pseudo-codice eseguito da  $P_i$

Si può intuitivamente verificare la correttezza dell' algoritmo notando che:

- la proprietà di *Validity* è direttamente soddisfatta dalla proprietà di *Liveness* delle primitive **send** e **recv**.

- la proprietà di *Agreement* deriva dall'assunzione di canali che non si guastano e dalla *Liveness* delle primitive **send** e **recv**. Consideriamo il caso in cui un processo corretto  $q$  riceva un messaggio  $m$ . Se il mittente  $p$  del messaggio è corretto allora tutti i processi corretti alla fine consegneranno  $m$  per la proprietà di *Liveness* delle primitive **send** e **recv**. Se, invece, il mittente  $p$  non è corretto allora non è detto che tutti i processi corretti consegneranno  $m$ . Ma secondo l'algoritmo in Figura 2.11 il processo corretto  $q$  una volta ricevuto il messaggio  $m$  lo reinvia in broadcast agli altri processi (linea 4 di R-deliver). Questo assicura che tutti i processi corretti alla fine consegnino  $m$ .
- la proprietà di *Integrity* è soddisfatta grazie alla proprietà di *Safety* delle primitive **send** e **recv**.

### Causal reliable Broadcast in assenza di guasti

Si consideri un'applicazione di invio di notizie in cui il processo  $A$  invia una notizia ai processi  $B$  e  $C$  e quindi  $B$ , ricevuta la notizia da  $A$ , invia una notizia ad  $A$  e a  $C$ . Se la comprensione della notizia inviata da  $B$  si fonda su quella della notizia inviata da  $A$ , è necessario che  $C$  riceva la notizia di  $A$  prima della notizia di  $B$ . Tale situazione è descritta in Figura 2.12 considerando che  $A$ ,  $B$  e  $C$  corrispondono ai processi  $P_1$ ,  $P_2$  e  $P_3$  e le notizie inviate da  $A$  e  $B$  corrispondono ai messaggi  $m_1$  e  $m_2$  rispettivamente.

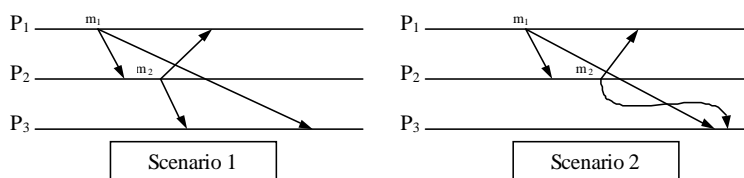


Figura 2.12: Consegna di messaggi causalmente ordinata

In particolare nello scenario 1 il messaggio  $m_2$  viene ricevuto da  $P_3$  prima di  $m_1$  e questo genera una inconsistenza in  $P_3$  che può non capire la causa di quel messaggio. Nello scenario 2,  $P_3$  vede la corretta sequenza di messaggi. Per garantire il verificarsi dello scenario 2 dobbiamo introdurre l'ordinamento causale.

Ricordiamo che l'ordinamento causale impone che per ogni coppia di messaggi tali che:

1. l'evento di trasmissione di  $m_1$  precede causalmente quello di  $m_2$ :  $\text{send}(m_1) \rightarrow \text{send}(m_2)$  e
2.  $m_1$  e  $m_2$  sono diretti allo stesso processo destinazione  $P_x$

allora

- l'evento di consegna del messaggio  $m_1$  ( $\text{deliver}(m_1)$ ) deve precedere quello di  $m_2$  ( $\text{deliver}(m_2)$ ) nel processo  $P_x$

Da un punto di vista realizzativo, una primitiva causale si può implementare come una primitiva FIFO, ovvero aggiungendo informazione di controllo (lato trasmittente) e ritardando opportunamente la consegna dei messaggi arrivati troppo presto attraverso la loro memorizzazione (lato ricezione). Per esempio il messaggio  $m_2$  mostrato nello scenario 1 di Figura 2.12, una volta arrivato, verrebbe memorizzato e consegnato a  $P_3$  subito dopo la consegna di  $m_1$  allo stesso processo. Ciò che cambia rispetto ad una primitiva FIFO riguarda la dimensione dell'informazione di controllo allegata agli eventi significativi, ovvero alle trasmissioni dei messaggi. Nei protocolli FIFO, infatti, si usa un numero di sequenza, mentre nei protocolli causali viene utilizzato un timestamping vettoriale. In Figura 2.13 è mostrato il protocollo per l'ordinamento causale.

```

%Inizializzazione
 $Vc_i := [0, \dots, 0]$ ;

C-BROADCAST( $m, Pdest = \langle P_1, \dots, P_n \rangle$ )
1  $m.Vc := Vc_i$ ;
2 send( $m$ ) to  $Pdest$ ;
3  $Vc_i[i] := Vc_i[i] + 1$ ;

C-DELIVER
1 when ( $P_i$  riceve  $m$  da  $P_j$ ) do
2   ritarda la consegna di  $m$  finchè  $\forall x \in [1, \dots, n] : m.Vc[x] \leq Vc_i[x]$ ;
3   if ( $i \neq j$ )
4     then  $Vc_i[j] := Vc_i[j] + 1$ ;
5   consegna  $m$ 

```

Figura 2.13: Pseudo-codice eseguito da  $P_i$  per l'ordinamento causale

Un processo che vuole inviare un messaggio  $m$  ad un insieme di processi  $Pdest$  invoca la primitiva di causal multicast (si noti che il multicast è una semplice generalizzazione del broadcast in cui la destinazione è costituita da un gruppo di processi). Specificatamente, ogni processo è fornito di un vector clock che considera solo le trasmissioni dei multicast causali come eventi rilevanti. Ad ogni messaggio trasmesso, il processo  $P_i$  allega il proprio vector clock corrente (linea 1) e successivamente alla trasmissione (linea 2) incrementa il proprio contatore di uno (linea 3). In ricezione (istruzione **when**) si ritarda la consegna di un messaggio  $m$  fino a che quest'ultimo non porti nessuna notizia circa messaggi trasmessi da altri processi e non ancora

consegnati a  $P_i$ . Da notare che questi messaggi precederebbero causalmente  $m$ . Questo si traduce all'interno del protocollo nel verificarsi del predicato presente nella seconda linea del codice di ricezione.

### 2.6.3 Consenso

Il problema del Consenso è un particolare problema di agreement. È l'astrazione di una classe di problemi in cui i processi partono con le loro "opinioni" (forse divergenti) e devono accordarsi su un'opinione comune. Formalmente, nel problema del Consenso, i processi corretti propongono un valore e tutti i processi corretti devono decidere tra i valori proposti un valore comune. Il consenso è definito in termini di due primitive  $propose(v)$  e  $decide(v)$ . Quando un processo esegue  $propose(v)$ , significa che il processo *propone* il valore  $v$ . Similmente quando un processo esegue  $decide(v)$  significa che il processo *decide* il valore  $v$ . La specifica del problema è la seguente [17]:

- *Termination*: Ogni processo corretto alla fine decide un valore.
- *Validity*: Se tutti i processi corretti che propongono un valore, propongono  $v$ , allora tutti i processi corretti alla fine decidono  $v$ .
- *Agreement*: Se un processo corretto decide  $v$ , allora tutti i processi corretti alla fine decidono  $v$ .
- *Integrity*: Ogni processo decide al più una volta, e se decide per  $v$  allora qualche processo ha proposto  $v$ .

La prima proprietà è una proprietà di *liveness*, mentre le altre tre proprietà sono proprietà di *safety*.

### 2.6.4 Impossibilità del Consenso in Sistemi Asincroni

Un risultato fondamentale sul problema del Consenso è stato provato da Fischer, Lynch e Patterson [13]. Questo risultato asserisce che non esiste alcun algoritmo deterministico che risolve il problema del Consenso in un sistema asincrono soggetto *anche ad un singolo* guasto di processo di tipo *crash*.

Il problema fondamentale è che in un sistema asincrono soggetto a guasti di tipo crash, un processo guasto non può essere distinto da un processo estremamente lento. L'impossibilità da parte di processi corretti di determinare con accuratezza quali processi sono realmente guasti è il cuore dell'impossibilità di raggiungere Consenso.

A questo scopo si riporta spesso l'esempio dei quattro eserciti alleati, ognuno dei quali è comandato da un generale. I quattro eserciti devono decidere tra "Attacco" o "Ritirata". L'obiettivo è conquistare un castello,

per avere successo tutti e quattro gli eserciti devono decidere di attaccare. I generali tra di loro comunicano attraverso messaggeri affidabili (messaggeri che non muoiono e che non perdono/danneggiano i messaggi) che però non si può prevedere quanto impiegheranno a trasportare il messaggio. Inoltre i generali possono essere uccisi. In questa situazione *non* si può decidere tra “Attacco” o “Ritirata”. Per convincersene basti pensare alla situazione in cui un generale ha chiesto agli altri di decidere di attaccare ma ha ricevuto risposta solo da due generali. Il problema è che il fatto di non aver ricevuto ancora risposta dal terzo generale non dice niente sullo suo stato. Il terzo generale potrebbe essere morto (nel qual caso bisognerebbe decidere per la “Ritirata”) oppure solo pigro a prendere la decisione o il messaggero lento (nel caso di generale vivo invece si dovrebbe decidere per “Attacco”). Questa indeterminatezza porta all'impossibilità di prendere un decisione.

Tale risultato di impossibilità ha spinto molti ricercatori a trovare un insieme di assunzioni minimali che, quando soddisfatte da un sistema distribuito asincrono, rendono il problema del Consenso risolvibile in tale sistema. Una possibilità è risolvere il problema in sistemi parzialmente sincroni. Si sono studiati molti modelli che hanno cercato di restringere le assunzioni di sincronia a quelle necessarie per risolvere il problema [10, 11]. Un altro modo per risolvere il problema del Consenso è estendere il sistema asincrono con *failure detector* [5] (che verranno illustrati nella prossima sezione).

## Failure Detectors

La maggiorparte dei sistemi reali esibiscono delle caratteristiche di sincronia parziale: e.g. ritardo di msg e/o tempo di transizione di stato dei processi *limitato* nella vasta maggioranza dei casi. Per risolvere il problema del Consenso si potrebbe sfruttare queste assunzioni di timing, tuttavia progettare un algoritmo che deve integrare queste assunzioni di timing non è affatto semplice. L'idea è allora offrire un *servizio* che in qualche modo incapsuli queste assunzioni di timing e dia la possibilità di sviluppare un'applicazione come se dovesse girare in un sistema asincrono. Questo servizio viene offerto attraverso i *Failure Detector*[5].

Informalmente un failure detector è un oracolo distribuito che dà suggerimenti (anche non corretti) a proposito dei processi che hanno fatto crash fino a quel momento. Ogni processo  $p_i$  ha accesso ad un modulo locale  $FD_i$  che monitora gli altri processi nel sistema e mantiene l'insieme di quelli che *sospetta* aver fatto crash. Ogni processo periodicamente consulta il modulo locale di failure detector e può usare l'insieme dei sospetti ritornato per risolvere il Consenso. L'informazione ritornata da un modulo di failure detector  $FD_i$  può non essere corretta (ad esempio viene sospettato un processo che non si è guastato) e può essere inconsistente (ad esempio  $FD_i$  sospetta al tempo  $t$  un processo  $p_k$  mentre nello stesso istante di tempo  $t$  un altro modulo  $FD_j$  non sospetta  $p_k$ ).

Un failure detector è caratterizzato a livello astratto da due proprietà chiamate *completeness* e *accuracy*.

**Completeness.** La completeness di un failure detector è relativa alla sua abilità nel rilevare processi che hanno fatto crash.

- *Strong Completeness.* Ogni processo che fa crash alla fine è per sempre sospettato da *tutti* i processi corretti.
- *Weak Completeness.* Ogni processo che fa crash alla fine è per sempre sospettato da *qualche* processo corretto.

**Accuracy.** La accuracy di un failure detector restringe i sospetti non corretti che un failure detector può fare.

- *Strong Accuracy.* Nessun processo è sospettato prima che faccia crash.
- *Weak Accuracy.* Qualche processo corretto non è mai sospettato.
- *Eventual Strong Accuracy.* Esiste un tempo dopo il quale *nessun* processo corretto è mai sospettato da un processo corretto.
- *Eventual Weak Accuracy.* Esiste un tempo dopo il quale *qualche* processo corretto non è mai sospettato da un processo corretto.

**Classi di Failure Detector.** Le otto combinazioni di completeness e accuracy definiscono altrettante classi di failure detector. Ogni classe ha un nome come illustrato nella Tabella 2.1[5]. La classe di failure detector  $\diamond S$  è importante poiché qualsiasi failure detector di questa classe permette di risolvere il problema del Consenso in un sistema asincrono con una maggioranza di processi corretti (solo una minoranza di processi può fare crash). L'algoritmo che risolve il Consenso sotto queste assunzioni è descritto più avanti. E' stato dimostrato che  $\diamond S$  è la classe più debole che rende possibile risolvere il Consenso in un sistema asincrono con una maggioranza di processi corretti [26]<sup>2</sup>.

### 2.6.5 Algoritmi Di Consenso

In questa sezione vengono presentati due algoritmi, proposti da Chandra e Toueg [5], che risolvono il problema del Consenso in un sistema asincrono soggetto a guasti di tipo crash, facendo ricorso ai failure detector.

---

<sup>2</sup>In realtà il risultato è provato per failure detector di classe  $\diamond W$ . Tuttavia i failure detector delle classi  $\diamond S$  e  $\diamond W$  sono equivalenti.

completeness	accuracy			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	<i>Perfect</i> $\mathcal{P}$	<i>Strong</i> $\mathcal{S}$	<i>Eventually Perfect</i> $\diamond\mathcal{P}$	<i>Eventually Strong</i> $\diamond\mathcal{S}$
Weak	$\mathcal{Q}$	<i>Weak</i> $\mathcal{W}$	$\diamond\mathcal{Q}$	<i>Eventually Weak</i> $\diamond\mathcal{W}$

Tabella 2.1: classi di failure detector

### Risolvere il Consenso usando $\mathcal{S}$

In questa sezione viene proposto un algoritmo che risolve il problema del Consenso usando un failure detector di tipo  $\mathcal{S}$ . Questo failure detector gode delle proprietà di *Strong Completeness* e *Weak Accuracy*: alla fine ogni processo che ha fatto crash viene permanentemente sospettato da tutti i processi corretti e qualche processo corretto non è mai sospettato. Questo algoritmo tollera al più  $n - 1$  processi guasti (in un sistema asincrono con  $n$  processi).

Questo algoritmo esegue tre fasi (si veda Figura 2.14).

```

PROPOSE( $v_p$ )
1  ARRAY  $V_p := \langle \perp, \perp, \dots, \perp \rangle$ ;
2  VALUE  $V_p[p] := v_p$ ;
3  ARRAY  $\Delta_p := V_p$ ;
4  F.1 : for each  $r_p := 1, 1 \leq r_p \leq n - 1$ 
5      send  $[r_p, \Delta_p, p]$  to all;
6      wait until (deliver  $[r_p, \Delta_q, q]$  from  $\forall q \notin FD_p$ )
7       $msgs_p[r_p] := \{(r_p, \Delta_q, q) \mid delivered(r_p, \Delta_q, q)\}$ ;
8       $\Delta_p := \langle \perp, \perp, \dots, \perp \rangle$ ;
9      for each  $k := 1, 1 \leq k \leq n$ 
10         if ( $V_p[k] = \perp$  and  $\exists (r_p, \Delta_q, q) \in msgs_p[r_p] \mid \Delta_q[k] \neq \perp$ )
11             then  $V_p[k] := \Delta_q[k]$ ;
12                  $\Delta_p[k] := \Delta_q[k]$ ;
13 F.2 : send  $[V_p]$  to all;
14         wait until (deliver  $[V_q]$  from  $\forall q \notin FD_p$ )
15          $lastmsgs_p := \{V_q \mid delivered(V_q)\}$ ;
16         for each  $k := 1, 1 \leq k \leq n$ 
17             if ( $\exists V_q \in lastmsgs_p \mid V_q[k] = \perp$ )
18                 then  $V_p[k] := \Delta_q[k]$ ;
19                      $V_p[k] := \perp$ ;
20 F.3 : DECIDE(primo non -  $\perp$  componente di  $V_p$ );

```

Figura 2.14: Algoritmo che risolve il consenso con  $\mathcal{S}$

- Fase 1: un processo  $p$  esegue  $n - 1$  round asincroni ( $r_p$  denota il numero

di round corrente per il processo  $p$ ). Durante un round il processo invia in broadcast un vettore di valori proposti (uno per ogni processo, eventualmente alcune entry sono vuote) quindi riceve i valori proposti dagli altri. Più in dettaglio ogni processo  $p$  propone il valore  $v_p$ , contenuto in  $V_p[p]$ , inviandolo col numero di round  $r$  a tutti i processi. Quindi aspetta di ricevere i valori proposti dai processi che non sono contenuti nella lista dei sospettati fornita dal modulo locale  $FD_p$  del failure detector, aggiorna il vettore dei valori proposti e passa al round  $r + 1$ .

- Fase 2: Viene inviato il vettore dei valori proposti a tutti i processi. Quindi  $p$  aspetta il vettore dei valori proposti da ogni processo non contenuto nella lista dei sospettati fornita dal modulo locale  $FD_p$  del failure detector e aggiorna il suo vettore con i valori ricevuti. Alla fine di questa fase il vettore dei valori proposti è lo stesso in ogni processo corretto, l' $i$ -simo elemento o contiene il valore proposto dal processo  $p_i$  o contiene un valore null. Dall'assunzione di weak accuracy (almeno un processo corretto non sarà mai sospettato) il vettore conterrà il valore proposto di almeno un processo.
- Fase 3: in questa fase i processi corretti decidono il primo valore non-null presente nel vettore dei valori proposti.

### Risolvere il Consenso usando $\diamond\mathcal{S}$

In questa sezione viene proposto un algoritmo che risolve il problema del Consenso usando un failure detector di tipo  $\diamond\mathcal{S}$ . Questo failure detector gode delle proprietà di *Strong Completeness* ed *Eventual Weak Accuracy*: alla fine ogni processo che ha fatto crash viene permanentemente sospettato da tutti i processi corretti ed esiste un istante di tempo dopo il quale qualche processo corretto non è mai sospettato. Questo algoritmo tollera al più  $\lceil \frac{n}{2} \rceil - 1$  processi guasti (in un sistema asincrono con  $n$  processi) e assume che  $\lceil \frac{(n+1)}{2} \rceil$  processi siano corretti. L'*Eventual Weak Accuracy* permette al failure detector di sospettare erroneamente qualsiasi processo corretto, solo dopo un certo istante di tempo qualche processo corretto non sarà più sospettato.

Questo algoritmo usa il paradigma *rotating coordinator* [6, 11] e procede in “round” asincroni. Si assume che tutti i processi sappiano che durante il round  $r$ , il coordinatore è il processo  $c = (r \bmod n) + 1$ . Tutti i messaggi sono inviati al, o prevencono dal, coordinatore “corrente”. Ogni volta che un processo diventa coordinatore, prova a decidere un valore consistente. Se il processo coordinatore è corretto e non è sospettato dai processi vivi, allora avrà successo e invierà tramite un *R-broadcast* il valore deciso.

L'algoritmo di Figura 2.15 passa attraverso tre epoche asincrone, ognuna delle quali può generare diversi round asincroni. Nella prima epoca

sono possibili diversi valori decisi. Nella seconda epoca un valore diventa “locked”: nessun altro valore può essere deciso. Nella terza epoca i processi decidono il valore “locked”.

Ogni round dell’algoritmo è diviso in quattro fasi asincrone.

- Fase 1: durante questa fase ogni processo invia la stima corrente del valore deciso, etichettato con un timestamp pari al valore dell’ultimo round in cui il valore stimato è stato aggiornato, al processo coordinatore  $c$ .
- Fase 2: durante questa fase il processo coordinatore  $c$  raccoglie  $\lceil \frac{(n+1)}{2} \rceil$  di tali valori stimati, seleziona quello con il timestamp più alto, e lo invia a tutti i processi come nuova stima  $estimate_c$ .
- Fase 3: durante questa fase per ogni processo corretto ci sono due possibilità:
  - $p$  riceve  $estimate_c$  dal processo coordinatore  $c$  e invia un  $ack$  a  $c$  per indicare che ha adottato  $estimate_c$  come nuova stima; oppure
  - in seguito all’aver consultato il modulo locale di failure detector  $FD_p$ ,  $p$  sospetta che  $c$  abbia fatto crash, e invia un  $nack$  a  $c$ .
- Fase 4: durante questa fase il coordinatore  $c$  aspetta  $\lceil \frac{(n+1)}{2} \rceil$  (sia  $ack$  che  $nack$ ) risposte. Se tutte le risposte sono  $ack$ , allora  $c$  sa che una maggioranza di processi ha cambiato la stima a  $estimate_c$ , e quindi che  $estimate_c$  è locked. Conseguentemente,  $c$  invia tramite  $R$ -broadcast una richiesta per decidere  $estimate_c$ . In ogni momento, se un processo  $R$ -deliver tale richiesta allora decide il valore in accordo col messaggio ricevuto.

```

PROPOSE( $v_p$ )
1  VALUE  $estimate_p := v_p$ ;
2  BOOLEAN  $undecided_p := \top$ ;
3  INTEGER  $r_p := 0$ ;
4  INTEGER  $ts_p := 0$ ;
5  while  $undecided_p = \top$  do
6       $r_p := r_p + 1$ ;
7       $c_p := (r_p \bmod n) + 1$ ;
8      F.1 : send [ $p, r_p, estimate_p, ts_p$ ] to  $c_p$ ;
9      F.2 : if ( $p = c_p$ )
10         then wait until (for  $\lceil \frac{(n+1)}{2} \rceil$  (deliver [ $q, r_p, estimate_q, ts_q$ ] from  $q$ ))
11              $msgs_p[r_p] := \{(q, r_p, estimate_q, ts_q) \mid delivered(q, r_p, estimate_q, ts_q)\}$ ;
12              $t := \text{massimo } ts \mid (q, r_p, estimate_q, ts) \in msgs_p[r_p]$ ;
13              $estimate_p := estimate_q \mid (q, r_p, estimate_q, t) \in msgs_p[r_p]$ ;
14             send [ $p, r_p, estimate_p$ ] to all;
15         F.3 : wait until ((deliver [ $c_p, r_p, estimate_{c_p}$ ] from  $c_p$ ) or  $c_p \in FD_p$ )
16             if ( $delivered(c_p, r_p, estimate_{c_p})$  from  $c_p$ )
17                 then  $estimate_p := estimate_{c_p}$ ;
18                  $ts_p := r_p$ ;
19                 send [ $p, r_p, ack$ ] to  $c_p$ ;
20             else send [ $p, r_p, nack$ ] to  $c_p$ ;
21         F.4 : if ( $p = c_p$ )
22             then wait until (for  $\lceil \frac{(n+1)}{2} \rceil$  (deliver [ $q, r_p, ack/nack$ ] from  $q$ ))
23                 if (for  $\lceil \frac{(n+1)}{2} \rceil$   $delivered(q, r_p, ack)$ )
24                     then  $R - broadcast(p, r_p, estimate_p, decide)$ ;
25         when ( $R - deliver(q, r_q, estimate_q, decide)$ ) do
26             if ( $undecided$ )
27                 then  $DECIDE(estimate_q)$ ;
28                  $undecided := \perp$ ;

```

Figura 2.15: Algoritmo che risolve il consenso con  $\diamond\mathcal{S}$

L'idea che sta alla base dell'algoritmo proposto è che la rete di comunicazione alterna periodi di stabilità a periodi di instabilità (e.g. carico inspettato). Nei periodi di stabilità il Failure Detector non genera falsi sospetti, si comporta quindi dando un buon livello di accuratezza. Questo significa che il coordinatore corretto non verrà sospettato dalla maggioranza dei processi (magari da una maggioranza!) e quindi l'algoritmo termina perchè il coordinatore manda in R-broadcast il valore deciso. Si noti che se nella fase tre un processo sospetta il coordinatore allora non adotta la stima ricevuta (si consideri che l'invio della stima potrebbe essere avvenuto appena prima il crash del coordinatore), mentre nella fase 4 ogni processo che riceve la stima la adotta *anche* se sospetta il coordinatore perchè in questa fase il valore da decidere è già *locked* (presente in una maggioranza di processi).

## 2.7 Il Problema della Replicazione

Le applicazioni distribuite sono accessibili ad un numero sempre crescente di client, che accedono a questi servizi tramite Internet. Ciò implica che il requisito di alta disponibilità e affidabilità del software è sempre più importante in ambiti come la finanza, booking-reservation, controllo industriale, telecomunicazioni, ecc. Una soluzione per aumentare la disponibilità e l'affidabilità dei servizi consiste nello sviluppare software su hardware replicato tollerante ai guasti. Sebbene questa soluzione sia adatta per alcune classi di applicazioni e sia stata perseguita con successo da alcune compagnie come Tandem e Stratus, fattori economici hanno spinto a cercare una soluzione meno costosa basata sul software. L'idea è di avere più copie del servizio, chiamate repliche, distribuite su diversi host. Il client deve però avere l'impressione di interagire con una singola entità non replicata, da ciò nasce il problema di mantenere le diverse copie del servizio consistenti. Se supponiamo che una replica sia un processo che cambia stato tramite un'operazione di update è necessario che la sequenza di update eseguita da ogni replica sia la stessa. In questo modo lo stato delle repliche evolve in modo consistente. Assicurare la consistenza del server replicato non è l'unico scopo della replicazione. E' importante definire cosa bisogna garantire al client. Questo significa, ad esempio, che un client deve ricevere un risultato generato da un update eseguito per la sua richiesta, o che non esistano più update eseguiti per la stessa richiesta.

Nel seguito si tratterà dapprima il problema della consistenza del server replicato: si definiranno dei criteri di consistenza e le proprietà da soddisfare per mantenere le repliche consistenti. Quindi si prenderà in considerazione l'interazione client/server: si definiranno le proprietà della replicazione che una volta soddisfatte assicurano ai client risposte consistenti.

### 2.7.1 Consistenza del Server Replicato

#### Criteri di Consistenza

Un criterio di consistenza caratterizza il risultato ritornato da un'operazione. Esistono tre principali criteri di consistenza definiti in letteratura: *linearizzabilità*[18], *sequential consistency*[20], *causal consistency*[1]. In tutti e tre i casi un'operazione è eseguita sullo *stato più recente* di un processo. La linearizzabilità è, tra i tre, il criterio di consistenza più restrittivo, mentre la causal consistency definisce il criterio di consistenza più debole. Sia linearizzabilità che sequential consistency definiscono ciò che è informalmente chiamato un criterio di consistenza *forte*, mentre la causal consistency definisce un criterio di consistenza *debole*. La causal consistency è inclusa nella sequential consistency (un'esecuzione che soddisfa sequential consistency soddisfa anche causal consistency), e la sequential consistency è inclusa nella lineariz-

zabilità (un'esecuzione che soddisfa linearizzabilità soddisfa anche sequential consistency).

Molte applicazioni richiedono consistenza forte, i.e. linearizzabilità e sequential consistency, in quanto fornisce l'illusione di interagire con processi non replicati. Si considererà solo la linearizzabilità. La ragione per cui si considera linearizzabilità anziché sequential consistency nasce da ragioni pratiche. La linearizzabilità è infatti più facile da implementare rispetto alla sequential consistency. In altre parole molte implementazioni di consistenza forte si riducono ad assicurare la linearizzabilità.

### Linearizzabilità

Informalmente si può dire che assicurare la linearizzabilità significa avere un servizio replicato che risponde al client come fosse una singola entità logica.

**Esecuzione linearizzabile.** Per fare un esempio, si consideri una coda FIFO  $x$  che, allo scopo di essere fault-tolerant, è implementata da due repliche  $x_1$  e  $x_2$ . Si consideri la seguente esecuzione:

- al tempo  $t=1$  il processo  $p_i$  invoca l'operazione di accodamento  $enq(a)$ .
- al tempo  $t=2$  il processo  $p_j$  invoca l'operazione di accodamento  $enq(b)$ .
- al tempo  $t=3$  il processo  $p_j$  riceve la risposta, da parte di  $x$ ,  $ok()$ .
- al tempo  $t=4$  il processo  $p_i$  riceve la risposta, da parte di  $x$ ,  $ok()$ .
- al tempo  $t=5$  il processo  $p_i$  invoca l'operazione di estrazione  $deq()$ .
- al tempo  $t=6$  il processo  $p_j$  invoca l'operazione di estrazione  $deq()$ .
- al tempo  $t=7$  il processo  $p_i$  riceve la risposta, da parte di  $x$ ,  $ok(b)$ .
- al tempo  $t=8$  il processo  $p_j$  riceve la risposta, da parte di  $x$ ,  $ok(a)$ .

L' esecuzione consiste di quattro operazioni (denotate con  $O_1, O_2, O_3, O_4$ ): l'operazione di accodamento  $O_1$  invocata al tempo globale  $t = 1$  e completata al tempo globale  $t = 3$ ; l'operazione di accodamento  $O_2$ , invocata al tempo globale  $t = 1$  e completata al tempo globale  $t = 3$ ; l'operazione di estrazione  $O_3$ , invocata al tempo globale  $t = 5$  e completata al tempo globale  $t = 7$ ; l'operazione di estrazione  $O_4$  invocata al tempo globale  $t = 6$  e completata al tempo globale  $t = 8$ .

Trascurando per un momento i risultati ritornati da questa esecuzione ( $b$  al processo  $p_i$  ed  $a$  al processo  $p_j$ ), si può notare che l'operazione  $O_1$ , sebbene inizi al tempo globale  $t = 1$  viene completata *dopo* ( $t = 3$ ) l'invocazione dell'operazione  $O_2$  ( $t = 2$ ). Due operazioni così "interfogliate" vengono chiamate *concorrenti*. Anche le operazioni  $O_3$  e  $O_4$  sono concorrenti. Le

operazioni concorrenti possono esser state servite in ordine arbitrario:  $x$  potrebbe avere eseguito prima  $O_1$  e poi  $O_2$  oppure prima  $O_2$  e poi  $O_1$ . La stessa cosa vale per  $O_3$  e  $O_4$ .

Tuttavia le operazioni  $O_1$  e  $O_2$  sono entrambe completate *prima* di  $O_3$  e  $O_4$ . Ciò significa che  $x$  ha sicuramente eseguito  $O_1$  e  $O_2$  prima di  $O_3$  e  $O_4$ .

Quindi le possibili sequenze *legali* di operazioni eseguite da  $x$  sono:

- $S_1 = (O_1, O_2, O_3, O_4)$ :
- $S_2 = (O_2, O_1, O_3, O_4)$
- $S_3 = (O_1, O_2, O_4, O_3)$
- $S_4 = (O_2, O_1, O_4, O_3)$

Poiché i risultati delle operazioni  $O_3$  e  $O_4$  sono rispettivamente  $b$  ed  $a$ , una sequenza legale che *segue la specifica della coda FIFO* è  $S_2 = (O_2, O_1, O_3, O_4)$ . L'esecuzione in questo caso si dice *linearizzabile*.

Si noti che se l'operazione  $O_4$  avesse ritornato anch'essa  $b$  al tempo  $t = 8$  l'esecuzione non sarebbe stata linearizzabile perché *non esiste una sequenza legale che rispetta la specifica della coda FIFO*.

Ora sia assuma che le due repliche  $x_1$  ed  $x_2$  osservino la seguente sequenza di eventi:

- la replica  $x_1$  riceve le invocazioni nel seguente ordine:
  1.  $enq(b)$  da parte del processo  $p_j$ ;
  2.  $enq(a)$  da parte del processo  $p_i$ ;
  3.  $deq()$  da parte del processo  $p_i$ ;
  4.  $deq()$  da parte del processo  $p_j$ ;
- la replica  $x_2$  riceve le invocazioni nel seguente ordine:
  1.  $enq(a)$  da parte del processo  $p_i$ ;
  2.  $enq(b)$  da parte del processo  $p_j$ ;
  3.  $deq()$  da parte del processo  $p_i$ ;
  4.  $deq()$  da parte del processo  $p_j$ ;

Ogni replica gestisce le invocazioni in modo sequenziale, nell'ordine in cui le ha ricevute. Ciò significa che la replica  $x_1$  invia le risposte  $ok(b)$  a  $p_i$  e  $ok(a)$  a  $p_j$ , mentre la replica  $x_2$  invia le risposte  $ok(a)$  a  $p_i$  e  $ok(b)$  a  $p_j$ .

Se  $p_i$  e  $p_j$  considerano entrambi le risposte prodotte dalla replica  $x_1$  allora si ottiene l'esecuzione linearizzabile descritta sopra. Tuttavia se  $p_i$  considera la risposta di  $x_1$  mentre  $p_j$  considera la risposta di  $x_2$ , allora tutt'e due i processi ottengono la risposta  $b$ , quindi l'esecuzione non è linearizzabile.

Il problema in questo scenario è che le due repliche non ricevono le invocazioni nello stesso ordine. Un problema analogo si verifica se a causa di un crash del client, una replica gestisce l'invocazione mentre un'altra non la gestisce.

**Condizioni sufficienti per assicurare linearizzabilità.** Nell'esempio precedente si è visto che il concetto di linearizzabilità è legato alla specifica del servizio (nel caso discusso una coda FIFO). Tuttavia bisogna assicurare che tutte le possibili run di un generico servizio replicato siano linearizzabili anche quando non se ne conosce la specifica. A tal fine vengono introdotte due condizioni che una volta soddisfatte assicurano la linearizzabilità.

Se definiamo l'evento  $update(req)$  come la modifica dello stato di una replica, in accordo con la richiesta  $req$  mandata da un client, allora per assicurare linearizzabilità è sufficiente che le repliche eseguino tutte lo stesso insieme di update e che li eseguino nello stesso ordine. Queste condizioni possono essere espresse come segue:

- *Atomicity*: Se una replica esegue un update  $update(req)$  tutte le repliche corrette devono eseguire l'update  $update(req)$ .
- *Order*: Dati due diversi eventi di update  $update$  e  $update'$ , se due differenti repliche li eseguono entrambi, allora li eseguono nel medesimo ordine.

## 2.7.2 Proprietà della Replicazione

Si è detto che una tecnica di replicazione deve necessariamente mantenere la consistenza del server replicato. Per farlo è sufficiente che la tecnica di replicazione soddisfi le proprietà di *Atomicity* e *Order* definite nella precedente sezione.

Tuttavia il soddisfacimento di queste due proprietà non definisce completamente una *corretta* interazione client/server. Ad esempio il seguente scenario si potrebbe verificare: tre client diversi  $c_1, c_2, c_3$  inviano tre richieste diverse  $req_1, req_2$  e  $req_3$ , e l'insieme di repliche eseguono tutte  $update(req_1)$  e  $update(req_2)$  nel medesimo ordine e non eseguono mai  $update(req_3)$ . Le proprietà di Atomicity e Order sono soddisfatte ma il client  $c_3$  non riceverà mai un risultato per la sua richiesta. Un altro scenario che si potrebbe verificare è l'esecuzione da parte delle repliche di una sequenza di update in cui sono presenti due update per la stessa richiesta o un update per una richiesta che nessun client ha inviato. Tutti questi scenari violano una corretta interazione client/server. Le seguenti proprietà definiscono ciò che una tecnica di replicazione deve soddisfare per assicurare una corretta interazione client/server[8]:

**Termination.** Se un client invia una richiesta, a meno che non faccia crash, alla fine riceve una risposta.

**Uniform Agreed Order.** Se si verifica un evento  $update(req)$  tale che una replica esegue  $update(req)$  come  $i$ -simo evento di update, allora anche tutte le repliche che eseguono l' $i$ -simo update eseguono  $update(req)$  come  $i$ -simo evento.

**Update Integrity.** Per qualsiasi richiesta  $req$ , ogni replica esegue  $update(req)$  al più una volta, e solo se un client ha inviato la richiesta  $req$ .

**Response Integrity.** Se un client invia una richiesta  $req$  e consegna un risultato  $res$ , allora l'evento  $update(req)$  è stato eseguito da qualche replica.

Il soddisfacimento di queste proprietà garantisce il mantenimento della consistenza del server replicato relativamente a ciò che deve essere garantito al client. In particolare *Uniform Agreed Order* è una riformulazione della proprietà di Order. La proprietà di *Termination* assicura la terminazione di un'interazione client/server (i.e., la liveness del servizio replicato). La proprietà di *Update Integrity* assicura che le repliche eseguano gli update solo in seguito alla prima ricezione di una richiesta. Infine, la proprietà di *Response Integrity* garantisce che le risposte fornite ai client siano state generate dal servizio.

Si noti che una volta risolto il problema della consistenza del servizio replicato (una volta soddisfatte le proprietà di Order e Atomicity) soddisfare questa specifica è relativamente semplice (ad esempio si può soddisfare assumendo canali affidabili, equipaggiando i client con semplici meccanismi di ritrasmissione ed equipaggiando le repliche server con un meccanismo di filtraggio dei duplicati).

### 2.7.3 Tecniche di Replicazione a Due Livelli

Una tecnica di replicazione a due livelli prende in considerazione solo due tipi di giocatori: i client e le repliche server [25, 4, 14, 22, 12, 9]. In questa sezione vengono presentate due fondamentali classi di tecniche: (1) la tecnica di replicazione *primary-backup*, e (2) la tecnica di *replicazione attiva*. Nella prima tecnica, un processo, chiamato *primary*, assicura un controllo centralizzato. Tale controllo centralizzato non è invece presente nella seconda tecnica.

#### Replicazione Primary-backup

Nell'approccio *primary-backup* [4, 16], una delle repliche, chiamata *primary* gioca un ruolo particolare: riceve le invocazioni dai processi client, e invia il risultato indietro. Le altre repliche che non sono *primary* sono chiamate *backup*. I *backup* interagiscono con il *primary* e non interagiscono con i

client. Consideriamo una richiesta di un client  $req$  inviata da un client  $c$ . In assenza di guasti l'invocazione è gestita come segue (vedi Figura 2.16):

- Il primary ( $p_1$ ) riceve la richiesta  $req$  e la esegue. Alla fine dell'esecuzione la risposta  $res$  è disponibile e lo stato di  $p_1$  risulta aggiornato.
- Il primary invia dei messaggi di update  $upd$  contenenti l'identificatore  $reqID$  della richiesta  $req$ , la risposta  $res$  e l'aggiornamento del suo stato. Sulla ricezione dei messaggi di update  $upd$  i backup aggiornano il loro stato ed inviano un messaggio di acknowledgement  $ack$  al primary.
- Una volta che il primary riceve il messaggio  $ack$  da tutte le repliche non guaste, invia la risposta  $res$  al client  $C$ .

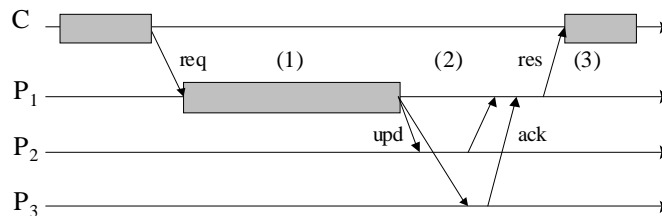


Figura 2.16: Tecnica primary-backup

Se il primary non si guasta per crash, allora lo schema appena descritto assicura certamente la consistenza del server replicato: è il primary che definisce l'insieme e l'ordine degli update (non ci sono altri processi che ricevono le richieste dei client).

Assicurare la linearizzabilità a fronte di guasto (crash) del primary è più difficile. Innanzitutto un nuovo primary deve essere selezionato, quindi si possono distinguere tre casi:

1. Il primary fa crash prima dell'invio dei messaggi di update  $udp$  ai backup ((1) in Figura 2.16). Il client non riceve risposta alla sua richiesta, e sospetterà un guasto. Dopo aver conosciuto l'identità del nuovo primary, il client reinvierà la richiesta  $req$ . L'invocazione è considerata come una nuova invocazione dal nuovo primary.
2. Il primary fa crash dopo l'invio dei messaggi di update  $udp$  ai backup ma prima che il client riceva la risposta  $res$  ((2) in Figura 2.16). Questo è il caso più difficile da gestire. L'atomicità deve essere assicurata: o tutti i backup ricevono  $udp$  o nessuno lo riceve. Se nessun backup riceve  $udp$  si ricade nel caso precedente. Se tutti lo ricevono, allora lo stato di tutti i backup è stato aggiornato dalla richiesta  $req$  ma il client non riceve risposta quindi reinvierà la richiesta. In questo

caso l'informazione  $\langle reqId, res \rangle$  è necessaria perché permette al nuovo primary di non eseguire due volte la stessa richiesta (che violerebbe la proprietà di *Update Integrity*) ma di inviare direttamente *res* al client.

3. Il primary fa crash dopo che il client ha ricevuto risposta. Non è un problema, il nuovo primary gestirà le nuove richieste.

Se si assume un meccanismo di failure detection *perfetto*, implementare questa tecnica è relativamente semplice. L'implementazione diventa molto più complicata nel caso di un modello di sistema asincrono, nel quale il meccanismo di failure detection può non essere affidabile. Il paradigma di comunicazione *view synchronous*, presentato nella Sezione 2.7.4, definisce la semantica di comunicazione che assicura la correttezza della tecnica di primary-backup nel caso di un meccanismo di failure detection non affidabile.

Uno dei principali vantaggi della tecnica primary-backup è permettere operazioni *non-deterministiche*. Questo non è il caso della replicazione attiva descritta nella seguente sezione.

### Replicazione Attiva

Nella tecnica di replicazione attiva, anche chiamata “state-machine approach” [16, 25], tutte le repliche giocano lo stesso ruolo: non esiste un controllo centralizzato come nel primary-backup. Si consideri un client *c* che invia una richiesta *req*, allora (si veda Figura 2.17):

- La richiesta viene inviata a tutte le repliche (nella figura  $p_1, p_2, p_3$ ).
- Ogni replica processa la richiesta, aggiorna il suo stato e invia il risultato *res* indietro al client.
- Il client aspetta fino a che (1) non riceve la prima risposta, oppure (2) fino a che non riceve una maggioranza di risposte identiche.

Se le repliche non si comportano in modo malizioso (se sono esclusi i guasti di tipo bizantino) allora il client aspetta solo la prima risposta. Se le repliche possono comportarsi in modo malizioso (guasti bizantini), allora  $2f + 1$  repliche sono necessarie per tollerare al più  $f$  repliche guaste [25]. In questo caso il client aspetta di ricevere  $f + 1$  risposte identiche.

La tecnica di replicazione attiva richiede che le richieste dei client siano ricevute dalle repliche non guaste nello stesso ordine. Ciò richiede l'uso di un'adeguata primitiva di comunicazione che assicuri le proprietà di atomicità e ordine. Questa primitiva è chiamata *total order multicast* o *atomic multicast* la cui semantica è presentata nella Sezione 2.7.4.

Escludendo i guasti bizantini, i compromessi tra replicazione attiva e replicazione primary-backup sono i seguenti:

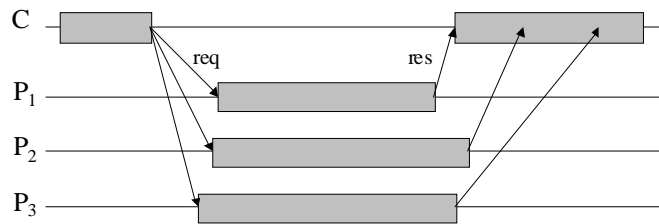


Figura 2.17: Tecnica di replicaizione attiva

- La replicaizione attiva richiede che le repliche siano deterministiche, mentre la tecnica di replicaizione primary-backup non lo richiede.
- Con la replicaizione attiva il crash di una replica è trasparente al client: il client non deve reinviare la stessa richiesta. Con la tecnica primary-backup il crash dei backup è trasparente al client, ma non il crash del primary. Nel caso di crash del primary il ritardo subito dal client (il tempo che passa tra l'invio della richiesta e la ricezione della risposta) può aumentare in modo significativo. Ciò rende la tecnica primary-backup non adatta in un contesto di applicazioni real-time.
- La replicaizione attiva usa più risorse rispetto alla replicaizione primary-backup, poiché ogni richiesta è processata da tutte le repliche.

#### 2.7.4 Group Toolkit

Nelle comuni applicazioni client/server le entità che partecipano alla comunicazione sono solitamente due (o catene composte da entità interagenti a coppie client/server in cui il server diviene client di altri server). In molte applicazioni la comunicazione anziché avvenire tra due entità può riguardare un'entità client ed un insieme di processi dislocati sugli host di una rete di calcolatori, costituenti logicamente un *gruppo*. I gruppi sono stati inizialmente introdotti come un conveniente meccanismo di indirizzamento: i gruppi hanno infatti un'identità che permette al programmatore di inviare i messaggi al gruppo senza menzionare ogni processo ad esso appartenente (detto *membro*). Successivamente sono stati estesi per gestire la replicaizione [2]. In questo caso il gruppo è costituito dalle repliche del servizio. Un gruppo può essere *statico* o *dinamico*:

- *gruppo statico*: un gruppo statico è un gruppo la cui membership non cambia durante l'intera vita del sistema. Questo non significa che i membri di un gruppo non possano fare crash, significa solo dire che la membership non viene cambiata per riflettere un crash di uno dei suoi membri: una replica, dopo aver fatto crash e prima di un possibile recovery, rimane membro del gruppo. I gruppi statici sono adeguati nell'ambito della replicaizione attiva, in quanto questo tipo di

replicazione non richiede di intraprendere nessuna azione specifica nel caso di guasto di una replica. Ciò non è vero nel caso della tecnica primary-backup: se il primary fa crash la membership deve cambiare, allo scopo di eleggere un nuovo primary.

- *gruppo dinamico*: un gruppo dinamico è un gruppo la cui membership cambia durante la vita del sistema. La membership cambia per esempio per riflettere il crash di uno dei suoi membri: una replica che fa crash viene rimossa dal gruppo. Se successivamente la stessa replica fa recovery viene reinclusa nel gruppo. La nozione di *view* è usata per modellare l'evoluzione della membership di un gruppo. La view iniziale contiene tutti i membri del gruppo, gli eventuali cambi di membership si riflettono in una nuova view.

I *group toolkit* sono strumenti che forniscono funzionalità per gestire i gruppi, come ad esempio servizi per gestire la membership di un gruppo o lo state transfer quando un membro viene reincluso in un gruppo dopo aver fatto recovery (*group membership service*, *state transfer service*). Inoltre comprendono primitive di comunicazione, chiamate *group communication*, che consentono di indirizzare messaggi ai gruppi in modo che rispettino condizioni di ordinamento e atomicità (*total order multicast*, *view synchronous multicast*).

## Group Communication e Replicazione Attiva

Nella sezione 2.7.3 si è visto che la replicazione attiva richiede una primitiva di total order multicast. Nella sezione 2.6.1 si è data la specifica del Total Order Broadcast che viene utilizzata tra i membri di un gruppo. In questo contesto, invece, la primitiva è utilizzata da processi esterni al gruppo (ad esempio un client). La specifica quindi, denotando con  $TOCAST(m, S)$  il total order multicast di un messaggio  $m$  ad un gruppo di repliche server  $S$ , viene facilmente adattata come segue:

- *Total Order*: si considerino due primitive  $TOCAST(m_1, S)$  e  $TOCAST(m_2, S)$ , se due repliche  $r_i$  ed  $r_j$  appartenenti ad  $S$  consegnano  $m_1, m_2$ , allora entrambe consegnano i messaggi nello stesso ordine.
- *Agreement*: si consideri la primitiva  $TOCAST(m, S)$ , se una replica  $r_i$  appartenente ad  $S$  consegna  $m$ , allora tutte le repliche corrette di  $S$  consegnano il messaggio  $m$ .
- *Termination*: si consideri la primitiva  $TOCAST(m, S)$  eseguita da qualche processo  $p_i$ . Se  $p_i$  è corretto, allora tutte le repliche corrette di  $S$  alla fine consegnano il messaggio  $m$ .

## Group Communication e Replicazione Primary-backup

Lo schema di replicazione primary-backup non richiede una primitiva di total order multicast, in quanto il primary definisce quali e in che ordine gli update devono esser fatti. Tuttavia in questo contesto è necessario che il gruppo sia dinamico allo scopo di definire un nuovo primary quando il primary corrente fa crash. Ciò richiede un *group membership service* [23] che definisce una nuova view ogniqualvolta la membership del gruppo cambia. Ogni view viene consegnata a tutte le repliche in modo che le repliche vengano a conoscenza dell'identità del primary (viene ad esempio considerato primary il primo elemento della view). Chiaramente è necessario un meccanismo che garantisca una consegna ordinata delle view. Tuttavia se anche questo meccanismo fosse presente ciò non sarebbe sufficiente ad assicurare la correttezza della tecnica di replicazione primary-backup. Per illustrare il problema, si consideri il seguente esempio, considerando la prima view composta da  $\{r_1, r_2, r_3\} \equiv S$  e quindi  $r_1$  come primary corrente:

- Il primary corrente riceve una richiesta dal client e fa crash mentre invia il messaggio di update ai backup  $r_2$  e  $r_3$ . Il messaggio di update viene ricevuto solo da  $r_2$ .
- Una nuova view viene consegnata alle repliche composta da  $\{r_2, r_3\}$ , quindi  $r_2$  diventa il nuovo primary. Gli stati di  $r_2$  ed  $r_3$  sono però inconsistenti.

L'inconsistenza è dovuta alla non atomicità dell'update inviato dal primary ai suoi backup. L'inconsistenza è evitata se, ogniqualvolta il primary invia il messaggio di update ai backup, o tutti o nessuno dei backup corretti riceve il messaggio. Questa semantica di atomicità nel contesto dei gruppi dinamici è chiamata *view synchronous multicast* [3, 24], denotato *VSCAST*. Per capirne il funzionamento, consideriamo  $t_k$  il tempo locale in cui una replica  $r_k$  consegna l' $i$ -sima view. Da questo istante di tempo in poi la replica invierà messaggi alle altre repliche, etichettandoli con il numero di view (in questo caso  $i$ ), tramite *VSCAST*. Supponiamo che  $r_k$  invochi *VSCAST*( $m$ ). Si consideri che venga definita e consegnata una nuova view  $i + 1$ , allora *VSCAST* assicura che  $m$  o viene consegnato da tutte le repliche (in realtà da ogni replica presente sia nella  $i$ -sima che nella  $i + 1$ -sima view) prima della consegna della  $i + 1$ -sima view o da nessuna replica. Quindi se  $r_k$  fosse il primary che fa crash, ed un nuovo primary venisse eletto, allora o tutte le repliche della nuova vista o nessuna di loro ha consegnato l'ultimo messaggio di update. Tutte le repliche nella nuova vista condividono lo stesso stato, ciò assicura consistenza.

### 2.7.5 Limiti della Replicazione a Due Livelli

Abbiamo visto che nell'ambito della replicazione attiva viene utilizzata una primitiva di total order multicast, mentre nel caso del primary-backup si fa uso di un view synchronous multicast.

Esiste un'importante relazione tra il problema del total order multicast, del view synchronous multicast e il problema del Consenso descritto nella Sezione 2.6.3. La relazione si basa sul concetto di riduzione [17]: un problema  $\mathcal{B}$  si riduce al problema  $\mathcal{A}$ , se esiste un algoritmo  $\mathcal{T}_{\mathcal{A} \rightarrow \mathcal{B}}$  che trasforma qualsiasi algoritmo per  $\mathcal{A}$  in un algoritmo per  $\mathcal{B}$ . Ciò significa che se  $\mathcal{A}$  può essere risolto allora può essere risolto anche  $\mathcal{B}$ . Informalmente si può dire che  $\mathcal{B}$  non è più difficile di  $\mathcal{A}$ . I problemi sono *equivalenti* se possono ridursi l'uno all'altro.

Si può dimostrare che il problema del Consenso può essere ridotto sia al problema del total order multicast [17] che al problema del view synchronous multicast [16]. Ciò significa che il Consenso non è più difficile né del total order multicast né del view synchronous multicast. Poiché vale il risultato di impossibilità [13] che asserisce che il Consenso non può essere risolto in un sistema asincrono con anche un solo processo che si può guastare per crash (si veda Sezione 2.6.4), allora anche il total order multicast e il view synchronous multicast non possono essere risolti in un sistema siffatto.

Tuttavia si può dimostrare che sia il problema del total order multicast [5] che del view synchronous multicast [15] possono essere ridotti al Consenso. Ciò significa che sia il total order multicast che il view synchronous multicast possono essere risolti dove è risolto il Consenso: in sistemi parzialmente sincroni o in sistemi asincroni estesi con failure detector, come detto nella Sezione 2.6.4.

Questo implica che le tecniche di replicazione a due livelli descritte in precedenza possono assicurare la consistenza del server replicato solo se implementate in sistemi che esibiscono un grado di sincronia sufficiente a risolvere il problema del Consenso. Ciò significa che client e repliche server devono essere distribuiti su reti come LAN o CAN. In pratica non possono essere distribuiti su reti asincrone come Internet.

# Bibliografia

- [1] M. Ahamad, P.W. Hutto, G. Neiger, J.E. Burns, and P. Kohli. Causal Memory: Definitions, Implementations and Programming. Technical Report TR GIT-CC-93/55, Georgia Institute of Technology, 1994.
- [2] K. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):37–53, December 1993.
- [3] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.
- [4] N. Budhiraja, F.B. Schneider, S. Toueg, and K. Marzullo. The Primary-Backup Approach. In S. Mullender, editor, *Distributed Systems*, chapter 8, pages 199–216. Addison Wesley, 1993.
- [5] T. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, pages 225–267, Mar. 1996.
- [6] J. Chang and N. Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.
- [7] F. Cristian. Probabilistic Clock Synchronization. *Distributed Computing*, 3:146–158, 1989.
- [8] X. Défago. *Agreement-Related Problems: From Semi Passive Replication to Totally Ordered Broadcast*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 2000. PhD thesis no. 2229.
- [9] X. Défago, A. Schiper, and N. Sergent. Semi-passive replication. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 43–50, West Lafayette, IN, USA, October 1998.
- [10] D. Dolev, C. Dwork, and L. Stockmeyer. On the Minimal Synchronism Needed for Distributed Consensus. *Journal of the ACM*, 34(1):77–97, January 1987.
- [11] C. Dwork and N.A. Lynch L. Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.
- [12] Pascal Felber and André Schiper. Optimistic active replication. In *Proceedings of 21st International Conference on Distributed Computing Systems (ICDCS'2001)*, Phoenix, Arizona, USA, April 2001. IEEE Computer Society.
- [13] M. Fischer, N. Lynch, and M. Patterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

- [14] D. Gifford. Weighted voting for replicated data. In *Proc. of 7th ACM Symposium on Operating System Principles*, pages 150–162, 1979.
- [15] R. Guerraoui. Transaction model vs Virtual Synchrony model: bridging the gap. In K. Birman, F. Cristian, F. Mattern, and A. Schiper, editors, *Distributed Systems: From Theory to Practice*, Lecture Notes on Computer Science. Springer-Verlag, 1995. To appear.
- [16] R. Guerraoui and A. Shiper. Software-Based Replication for Fault Tolerance. *IEEE Computer - Special Issue on Fault Tolerance*, 30:68–74, April 1997.
- [17] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcast and Related Problems. In S. Mullender, editor, *Distributed Systems*, chapter 16. Addison Wesley, 1993.
- [18] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.
- [19] L. Lamport. Time, Clocks and the Ordering of Event in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.
- [20] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.
- [21] L. Lamport and P. M. Melliar-Smith. Synchronizing Clocks in the Presence of Faults. *Journal of the ACM*, 32(1):52–78, 1985.
- [22] Dahlia Malkhi. Quorum Systems. In Joseph Urban and Partha Dasgupta, editors, *The Encyclopedia of Distributed Computing*. Kluwer Academic Publishers, 2000.
- [23] A. Ricciardi and K. P. Birman. Consistent Process Membership in Asynchronous Environments. In K. Birman and R. van Renesse, editors, *Reliable Distributed Computing With The ISIS Toolkit*, chapter 13. IEEE Computer Society Press, Los Alamitos, 1993.
- [24] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *Proc. of the 13th International Conference on Distributed Computing Systems*, pages 561–568, May 1993.
- [25] F.B. Schneider. Replication Management using state-machine approach. In S. Mullender, editor, *Distributed Systems*, chapter 7, pages 169–198. Addison Wesley, 1993.
- [26] V. Hadzilacos T. D. Chandra and S. Toueg. The Weakest Failure Detector for Solving Consensus. *Journal of the ACM*, pages 685–722, July 1996.