



*Lezione 1 -
Introduzione ai
Sistemi Distribuiti e
Problemi di
Consenso*

Prof. Esposito Christian

*Corso di
Sicurezza dei Dati*

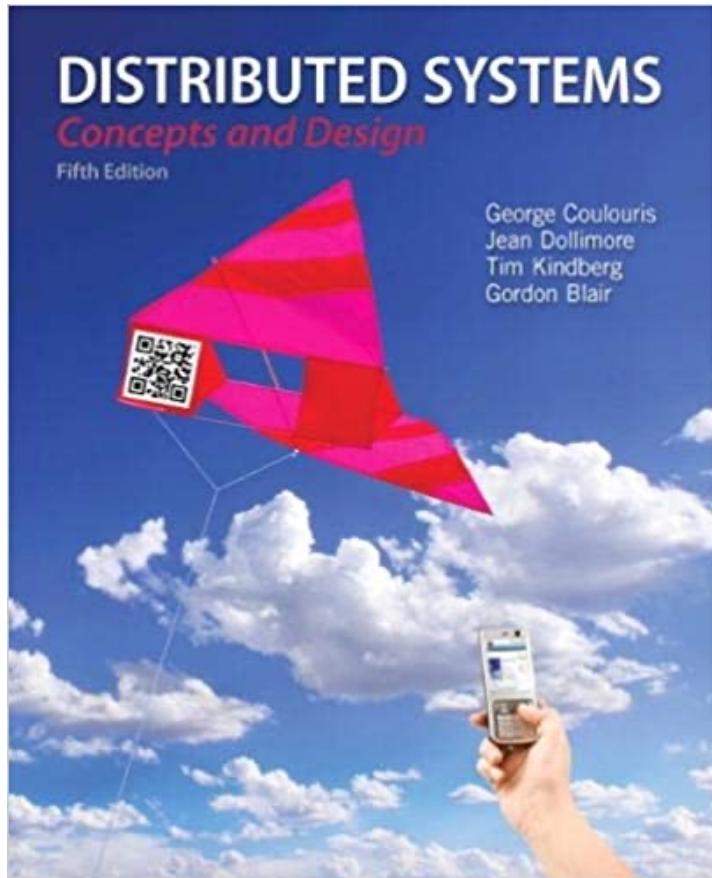


::: Sommario

- Caratterizzazione dei Sistemi Distribuiti
 - Definizioni ed Esempi;
 - Sfide di Progettazione e Proprietà di Trasparenza;
 - Modelli di programmazione, architettura e aspetti fondamentali dei Sistemi Distribuiti;
 - Sincronicità e modi di fallimento.
- Il Consenso nei Sistemi Distribuiti
 - Formulazione del problema del consenso (C);
 - Il problema dei generali bizantini (BG);
 - Il problema della consistenza interattiva (IC);
 - Relazioni tra C, BG, IC;
 - Soluzioni ai problemi C e BG nei sistemi sincroni;
 - Teorema di impossibilità nei sistemi asincroni;
 - Protocollo di Paxos.

::: References

- G. Coulouris et al., “Distributed Systems: Concepts and Design”, V Edizione (Aprile 27, 2011) capitoli 1, 2, 15.



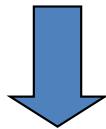


Caratterizzazione dei Sistemi Distribuiti

::: Definizione di sistema distribuito (1/2)

Una definizione di sistema distribuito:

“Il mio programma gira su un sistema distribuito quando non funziona per colpa di una macchina di cui non ho mai sentito parlare” (L. Lamport).

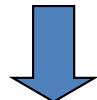


Caratteristiche fondamentali di un sistema distribuito:

- **concorrenza** dei componenti;
- componenti appartenenti a **diversi domini organizzativi/amministrativi**;
- possibilità di **guasti indipendenti** dei componenti;
- le macchine sono **autonome** (hardware), ma l’utente pensa di lavorare su una sola macchina (software).

::: Definizione di sistema distribuito (2/2)

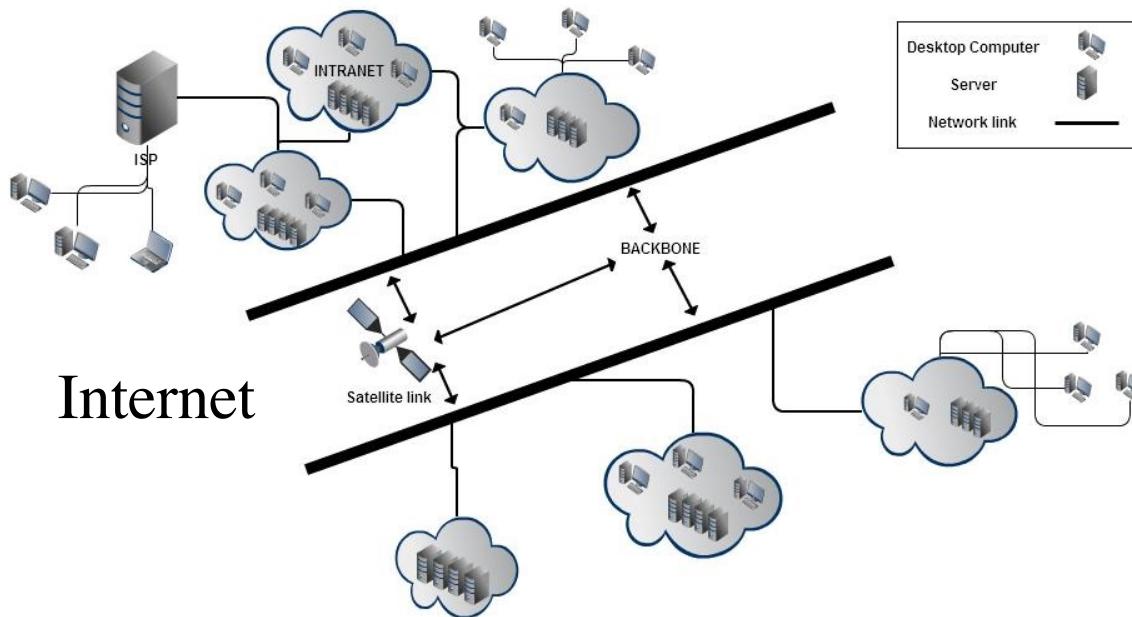
Un sistema distribuito è un sistema i cui componenti, localizzati in computer connessi in rete, comunicano e coordinano le loro azioni solo attraverso scambio di messaggi (*G. Coulouris et al.*).



Caratteristiche fondamentali di un sistema distribuito:

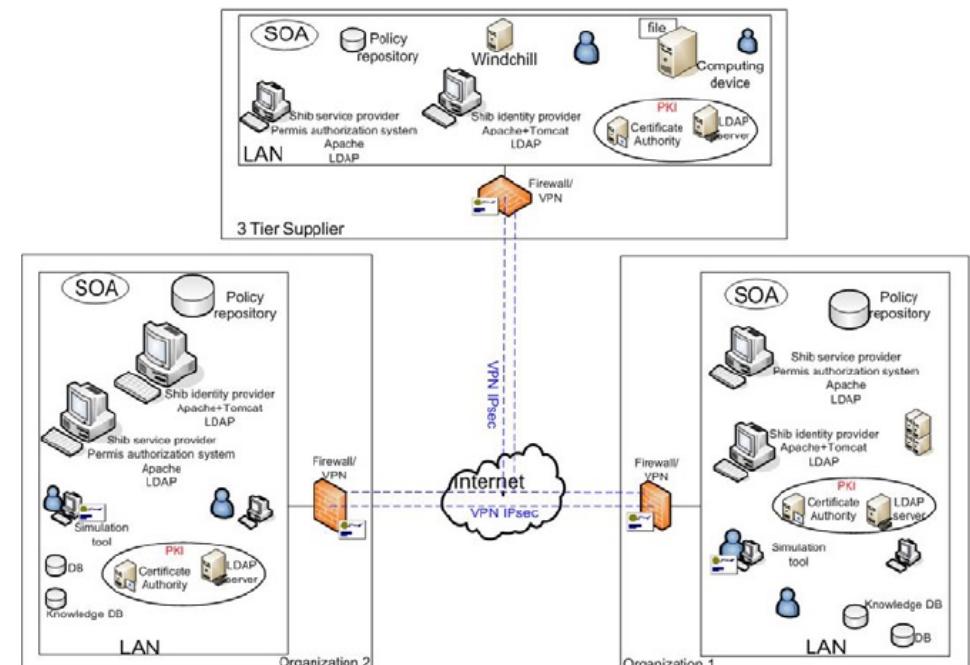
- **concorrenza** dei componenti;
- **assenza di un clock globale**
 - sincronizzazione e interazione via scambio messaggi;
- possibilità di **guasti indipendenti** dei componenti:
 - nei nodi (*crash*, attacchi, ...),
 - nel sottosistema di scambio messaggi (ritardi, perdita, attacchi, ...).

::: Esempi di sistemi distribuiti



I sistemi di elaborazione sono un'altra applicazione tipica dei sistemi distribuiti, dove un insieme di computer connessi distribuiscono un'elaborazione molto complessa tra i vari computer.

La rete Internet è un esempio, dove diversi processi connessi da reti di vario tipo si scambiano messaggi mediante protocolli di comunicazione.



::: Progettazione di sistemi distribuiti (1/5)

Fattori di complessità:

- **Eterogeneità**: varietà e differenza di reti (tecnologie e protocolli), sistemi operativi, hardware (dei server, dei client, ...), linguaggi di programmazione, ... → Middleware
- **Apertura (*openness*)**: il livello di complessità col quale servizi e risorse condivise possono essere resi accessibili a una varietà di clienti, o resi tra essi interoperanti.

I sistemi aperti richiedono:

- interfacce pubbliche dei componenti
- standard condivisi
- adeguati test di conformità

::: Progettazione di sistemi distribuiti (2/5)

Un sistema distribuito aperto offre servizi secondo regole standard per descrivere la sintassi e la semantica del servizio. Le regole sono di solito specificate attraverso interfacce, che specificano i nomi delle funzioni disponibili e la loro intestazione.

Un'interfaccia dovrebbe essere:

- Completa: specifica tutto quello che è necessario per implementarla,
- Neutra: non dà informazioni su come deve apparire un'implementazione.

Queste caratteristiche influenzano:

- Interoperabilità: due implementazioni costruite diversamente possono coesistere e lavorare insieme,
- Portabilità: un'applicazione sviluppata per un sistema distribuito A può essere eseguita su un sistema B.

::: Progettazione di sistemi distribuiti (3/5)

- **Sicurezza (security):**
 - confidenzialità (protezione dall'accesso da parte di utenti non autorizzati),
 - integrità (protezione dall'alterazione o compromissione),
 - disponibilità (protezione dall'interferenza con i mezzi di accesso alle risorse).
- **Concorrenza:** l'accesso a risorse e servizi condivisi deve essere consentito in maniera virtualmente simultanea a più utenti.
- **Trasparenza:** Un sistema distribuito deve nascondere che i suoi processi e risorse sono fisicamente distribuite. Otto forme di trasparenza identificate nell'ISO Reference Model for Open Distributed Processing (RM-ODP).

::: Progettazione di sistemi distribuiti (4/5)

- Trasparenza di accesso: nascondere le differenze di rappresentazione dei dati e del modo in cui gli utenti accedono alle risorse.
- Trasparenza di locazione: nascondere la locazione fisica di una risorsa.
- Trasparenza di migrazione: permettere il continuo accesso a risorse che possono essere spostate (trasparenza di rilocazionese avviene mentre una risorsa è in uso).
- Trasparenza di duplicazione: nascondere la duplicazione di delle risorse (es. per migliorare le prestazioni).
- Trasparenza di concorrenza: nascondere agli utenti che competono per le medesime risorse.
- Trasparenza ai fallimenti: nascondere all'utente eventuali guasti di risorse.
- Trasparenza alla persistenza: nascondere il tipo di memoria su cui si trova la risorsa (es. volatile o fissa).

::: Progettazione di sistemi distribuiti (5/5)

- **Scalabilità**: un sistema è scalabile se resta efficace ed efficiente anche a seguito di un aumento considerevole di utenti o risorse.
La scalabilità di un sistema si può misurare secondo 3 dimensioni:
 - rispetto alla scala: posso aggiungere utenti e risorse,
 - geograficamente: utenti e risorse possono essere fisicamente molto distanti,
 - dal punto di vista amministrativo: facile da gestire anche in presenza di organizzazioni amministrative indipendenti.
- **Flessibilità**: un sistema distribuito flessibile deve rendere semplice la configurazione del Sistema e l'aggiunta di nuove componenti.
- **Guasti**: i guasti nei sistemi distribuiti sono parziali, e vanno gestiti in modo da controllare il livello di servizio offerto in caso di Guasti.

::: Modelli di sistemi distribuiti (1/6)

Un **modello** descrive tutte e sole le caratteristiche essenziali di un sistema distribuito, che è necessario considerare per specificare o analizzare i suoi aspetti strutturali e di funzionamento.

Un modello risponde a domande quali:

- quali sono le principali entità del sistema?
- quali sono le modalità di interazione tra essi?
- quali sono le caratteristiche che determinano il comportamento individuale e collettivo delle entità?

Per la specifica, l'analisi e lo sviluppo di sistemi distribuiti servono:

- modelli architetturali,
- modelli fondamentali,
- modelli di programmazione.

::: Modelli di sistemi distribuiti (2/6)

Descrivono le proprietà delle unità software secondo cui decomporre e programmare i sistemi distribuiti.

Tali proprietà sono astrazioni supportate da sistemi operativi, middleware, linguaggi, ambienti di sviluppo.

Es.: - modello basato su processi
- modello orientato agli oggetti
- modello a componenti
- modello orientato ai servizi

::: Modelli di sistemi distribuiti (3/6)

Descrivono l'organizzazione del sistema distribuito nelle sue parti componenti, le relazioni tra esse, e la loro allocazione sui nodi del sistema di elaborazione.

Es. di architetture:

- *client-server*
- 2-tiers, 3-tiers, n-tiers*
- *peer-to-peer*
- varianti:
 - proxy e cache*
 - server multipli*
 - codice mobile*
 - agenti mobili*
 - thin client*
- *service-oriented* (SOAs)

::: Modelli di sistemi distribuiti (4/6)

Un **modello fondamentale** è un'astrazione delle caratteristiche essenziali di un sistema, necessarie per analizzare e comprenderne il funzionamento, ragionare sul suo comportamento, e dimostrarne formalmente (logicamente, matematicamente) le proprietà.

Scopo di un modello fondamentale è:

- rendere esplicite le ipotesi rilevanti sul sistema;
- consentire di studiare comportamenti/proprietà possibili o non possibili del sistema, date le ipotesi.

Modelli fondamentali:

- modello di interazione
- modello dei guasti
- modello di sicurezza

::: Modelli di sistemi distribuiti (5/6)

Un sistema distribuito è composto da *processi* - entità che eseguono azioni elaborative, in esecuzione su più nodi del sistema di elaborazione, ciascuno dotato di un proprio orologio - che encapsulano risorse, ed interagiscono esclusivamente tramite messaggi scambiati su un canale di comunicazione.

Lo *stato di un processo* è l'insieme dei dati che può leggere o aggiornare, ed è locale e privato – non può essere letto e aggiornato dagli altri processi. Non esiste un *clock* globale.

La velocità dei processi e i tempi di trasmissione dei messaggi tipicamente non possono essere previsti con esattezza.

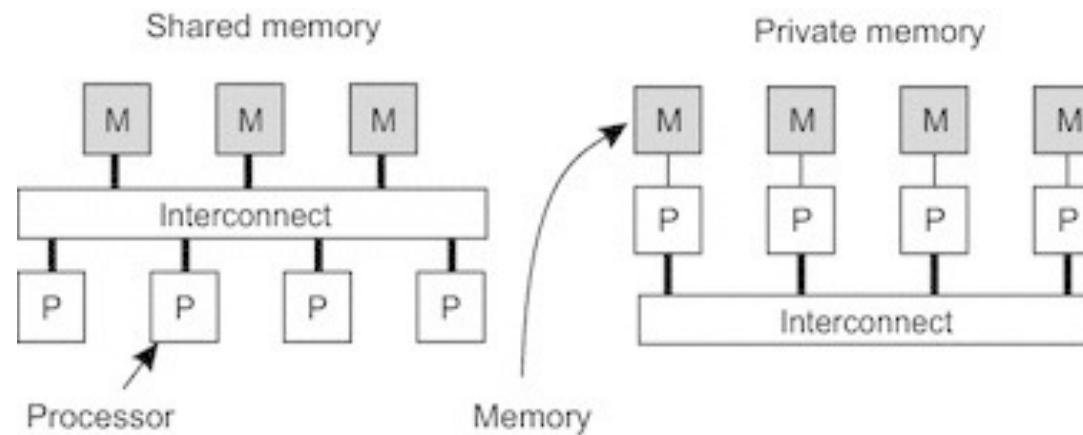
Modelli di interazione:

- sistema sincrono;
- sistema asincrono;
- sistema parzialmente sincrono.

::: Modelli di sistemi distribuiti (6/6)

Un sistema distribuito è formato da più *CPU*, ma queste possono essere organizzate in modo diverso:

- Multiprocessori: un unico spazio di indirizzi fisici è condiviso da tutte le CPU (memoria condivisa).
- Multicomputer: ogni macchina ha il suo indirizzo fisico (memoria privata).



Molti sistemi distribuiti sono composti da multicomputer eterogenei, con diverse componenti e reti dalle caratteristiche molto diverse.

::: Algoritmi distribuiti

Un **algoritmo distribuito** è la specifica delle azioni (elaborazioni, scambio messaggi) che devono essere intraprese dai processi che compongono il sistema, per il raggiungimento di un obiettivo.

È difficile descrivere tutti gli stati possibili di un algoritmo distribuito, anche per via dei malfunzionamenti (indipendenti) cui sono soggetti i processi e la trasmissione dei messaggi.

La progettazione di un algoritmo distribuito deve basarsi sulle ipotesi che è possibile (o non è possibile) fare sulla tempificazione del sistema, e prevedere la gestione dei malfunzionamenti.

::: Sistemi distribuiti sincroni (1/3)

Un sistema distribuito è **sincrono** (Hadzilacos e Toueg, 1994) se esistono e sono noti:

- i limiti inferiori e superiori al tempo di esecuzione di ogni passo di elaborazione;
- il limite superiore al tempo di consegna di un messaggio;
- il limite superiore al tasso di deviazione di ciascun orologio locale (*clock drift rate*) rispetto al tempo reale.

È spesso possibile stimare i limiti al tempo di esecuzione dei processi, al ritardo dei messaggi e alla deriva degli orologi, ma è difficile individuare e garantire i valori reali.

Se non è possibile garantire tali valori limite, qualunque algoritmo basato su di essi non è affidabile.

::: Sistemi distribuiti sincroni (2/3)

Vantaggi:

Per i sistemi sincrono è possibile definire algoritmi distribuiti basati sull'individuazione dei fallimenti tramite *time-out*.

Svantaggi:

È difficile assicurare tali proprietà in un sistema su grande scala e nel tempo.

In pratica, è spesso possibile stimare i limiti al tempo di esecuzione dei processi, al ritardo dei messaggi e alla deriva degli orologi, ma è difficile individuare e garantire i valori reali.

Se non è possibile garantire tali valori limite, qualunque algoritmo basato su di essi non è affidabile.

::: Sistemi distribuiti sincroni (3/3)

Tipicamente i sistemi distribuiti reali non sono sincroni.

È comunque possibile costruire sistemi distribuiti sincroni. Se processi non modellabili come sincroni hanno limiti superiori probabilistici sui tempi di esecuzione e di comunicazione, è possibile utilizzare i timeout per fare in modo che il sistema si comporti come un sistema parzialmente sincrono.

È necessario:

- individuare le risorse richieste dai processi per eseguire le elaborazioni e per scambiare messaggi;
- garantire un numero sufficiente di cicli di processore;
- garantire una sufficiente capacità di rete;
- fornire ai processori clock con deriva limitata.

::: Sistemi distribuiti asincroni

Un sistema distribuito è **asincrono** se non esistono limiti alla velocità di esecuzione dei processi, al ritardo di trasmissione dei messaggi, o alla deviazione degli orologi.

In un sistema asincrono, non è possibile formulare ipotesi temporali relativamente all'elaborazione, allo scambio messaggi, alla sincronizzazione.

Alcuni problemi non hanno soluzione nei sistemi asincroni.

Molti sistemi distribuiti reali sono asincroni (es.: Internet).

I modelli sincrono e asincrono sono gli **estremi** di uno spettro di possibilità con cui modellare i sistemi reali.

::: Modello dei fallimenti (1/6)

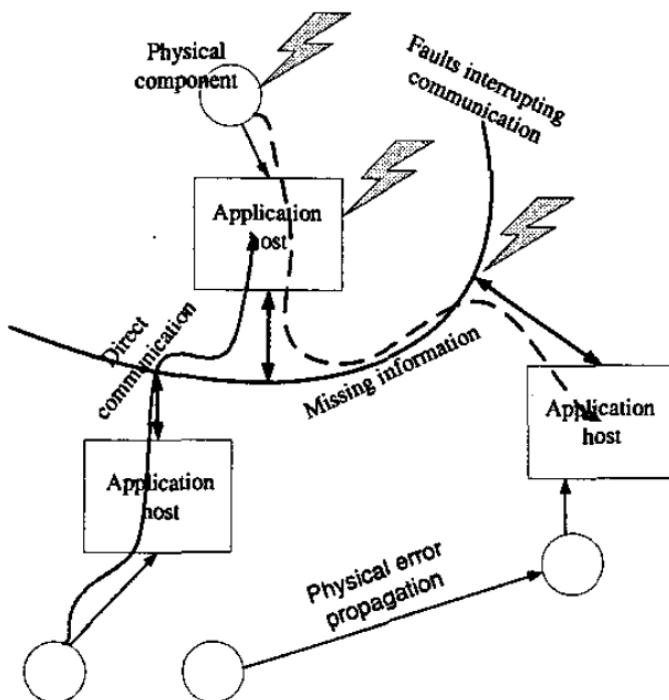
Fallimento (*failure*): scostamento da un comportamento considerato corretto o desiderabile.

Guasto (*fault*): causa originaria del fallimento (es: un “baco” in una istruzione del programma); anche se presente, il *fault* può non essere attivato in una esecuzione, per es. in funzione dei dati di ingresso (es: l’istruzione col “baco” non viene eseguita); quando il *fault* degenera in errore a causa dell’attivazione, esso si dice attivo (*active*), altrimenti è dormiente (*dormant*).

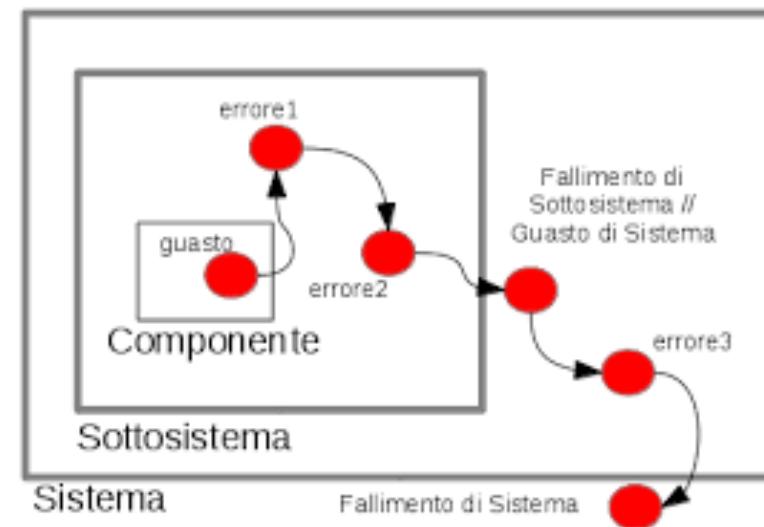
Errore (*error*): lo stato in cui transita il sistema a seguito dell’attivazione di un guasto, nel quale dunque il suo comportamento si discosta da quello considerato corretto (*correct behaviour*).

::: Modello dei fallimenti (2/6)

Si parla di “**catena fault – error – failure**”.



Errori e fallimenti si propagano da un nodo all’altro.



::: Modello dei fallimenti (3/6)

Un modello dei fallimenti per un sistema distribuito definisce le modalità con cui si possono verificare guasti in:

- processi,
- canali di comunicazione.

In pratica, si possono avere fallimenti:

- per *crash*,
- per omissione,
- di valore,
- bizantini,

e, per i sistemi sincroni:

- temporali

::: Modello dei fallimenti (4/6)

Classificazione dei fallimenti (Hadzilacos e Toueg, 1994):

- *omission failure*: si ha quando un processo o un canale non eseguono un'azione che ci si aspetta che eseguano;
- *arbitrary* o *Byzantine failure*: è la semantica peggiore per un guasto, in cui si può verificare qualunque tipo di errore;
- *timing failure*: mancato rispetto di una scadenza (applicabile solo ai sistemi sincroni).

Fallimenti **benigni**:

- *omission failures*;
- *timing failures*

::: Modello dei fallimenti (5/6)

Tipologie di fallimenti:

- *fail stop*: si ha quando un processo non esegue più azioni, e gli altri processi sono in grado di rilevarne il fallimento;
- *crash*: si ha quando un processo non esegue più azioni, e gli altri processi possono non essere in grado di rilevarne il fallimento.
- *omissione*:
 - da parte di un canale: il canale non trasporta messaggi;
 - da parte di un processo: il processo fa una send / una receive, ma il messaggio non viene spedito / ricevuto (*send omission / receive omission*).

::: Modello dei fallimenti (6/6)

Tipologie di fallimenti (segue):

- *prestazionale*:
 - di un canale: il tempo di consegna di un messaggio eccede il limite superiore;
 - di un processo: il tempo di esecuzione di una azione eccede il limite superiore;
 - del clock: la deriva rispetto a un clock ideale eccede il limite superiore.
- *bizantino*: un fallimento di un processo o di un canale, che si comportano in modo arbitrario (es: mandano più messaggi, omettono azioni, o si comportano in modo malizioso).

::: Modello di sicurezza (1/6)

Il modello di interazione prevede processi che encapsulano risorse (oggetti), e che forniscono ad altri processi l'accesso ad esse mediante interazioni con scambio di messaggi.

Il modello di interazione è dunque la base per il modello di sicurezza: la sicurezza di un sistema distribuito può essere ottenuta rendendo sicuri i processi e i canali di comunicazione tra essi, e proteggendo le risorse che i processi encapsulano.

Aspetti della sicurezza:

- confidenzialità (protezione dall'accesso di non autorizzati)
- integrità (protezione dall'alterazione o compromissione),
- disponibilità (protezione dei mezzi di accesso alle risorse).

::: Modello di sicurezza (2/6)

La sicurezza nei sistemi distribuiti deve riguardare tutti i componenti del sistema e coinvolge due aspetti principali:

- Le *comunicazioni* tra utenti e processi » soluzione : canali sicuri;
- L'*Autorizzazione* di utenti e processi » soluzione : controllo degli accessi.

Le possibili minacce alla sicurezza sono

- Intercettazione (accessi non autorizzati),
- Interruzione (diniego di servizio -denial of service),
- Alterazioni (modifiche di dati non autorizzate),
- Fabbricazione (inserimenti di dati non autorizzati).

::: Modello di sicurezza (3/6)

Un sistema distribuito sicuro ha bisogno di una *politica di sicurezza*, che definisce le azioni che le entità del sistema possono eseguire e quelle che sono proibite. Una politica può essere realizzata tramite meccanismi di sicurezza, come crittografia, autenticazione, autorizzazione, auditing.

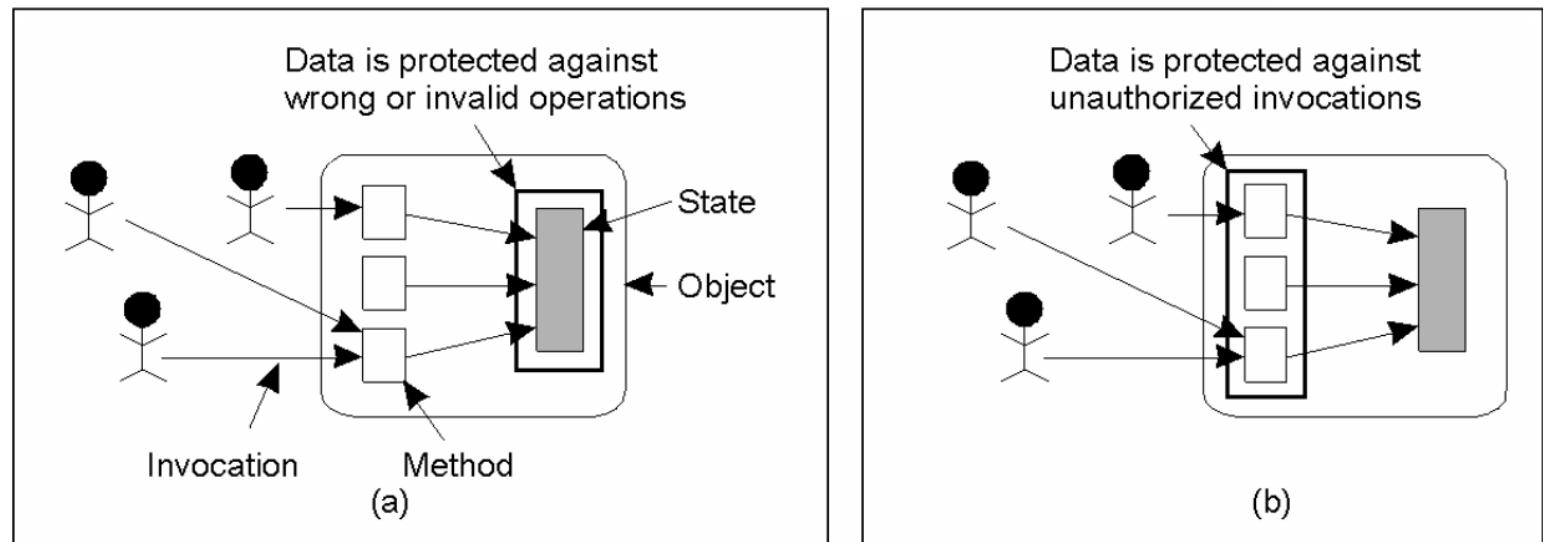
Diversi aspetti devono essere considerati nella progettazione di politiche di sicurezza: focus del controllo, meccanismi e livelli, semplicità.

Le minacce alla sicurezza rientrano in tre grandi classi:

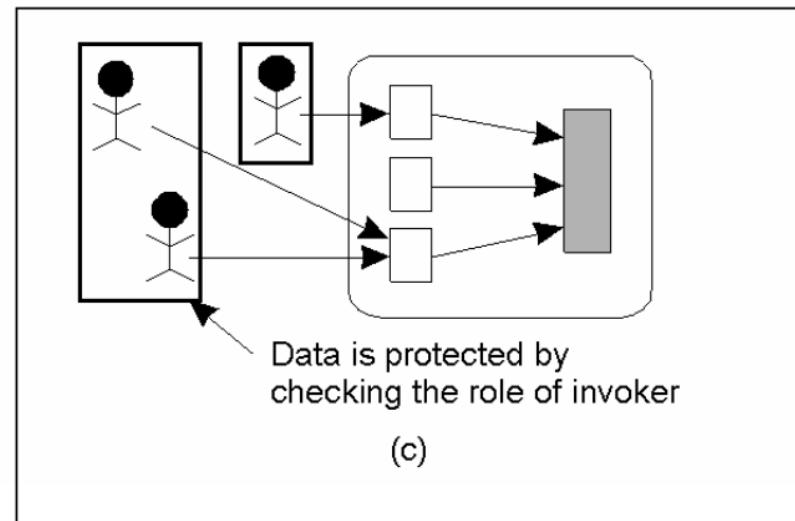
- Perdita: si riferisce all'acquisizione di informazioni da parte di destinatari non autorizzati,
- Manomissione: si riferisce all'alterazione non autorizzata delle informazioni,
- Vandalismo: si riferisce all'interferenza con il corretto funzionamento di un sistema senza guadagno per l'autore.

::: Modello di sicurezza (4/6)

Tre approcci per la protezione da minacce alla sicurezza.

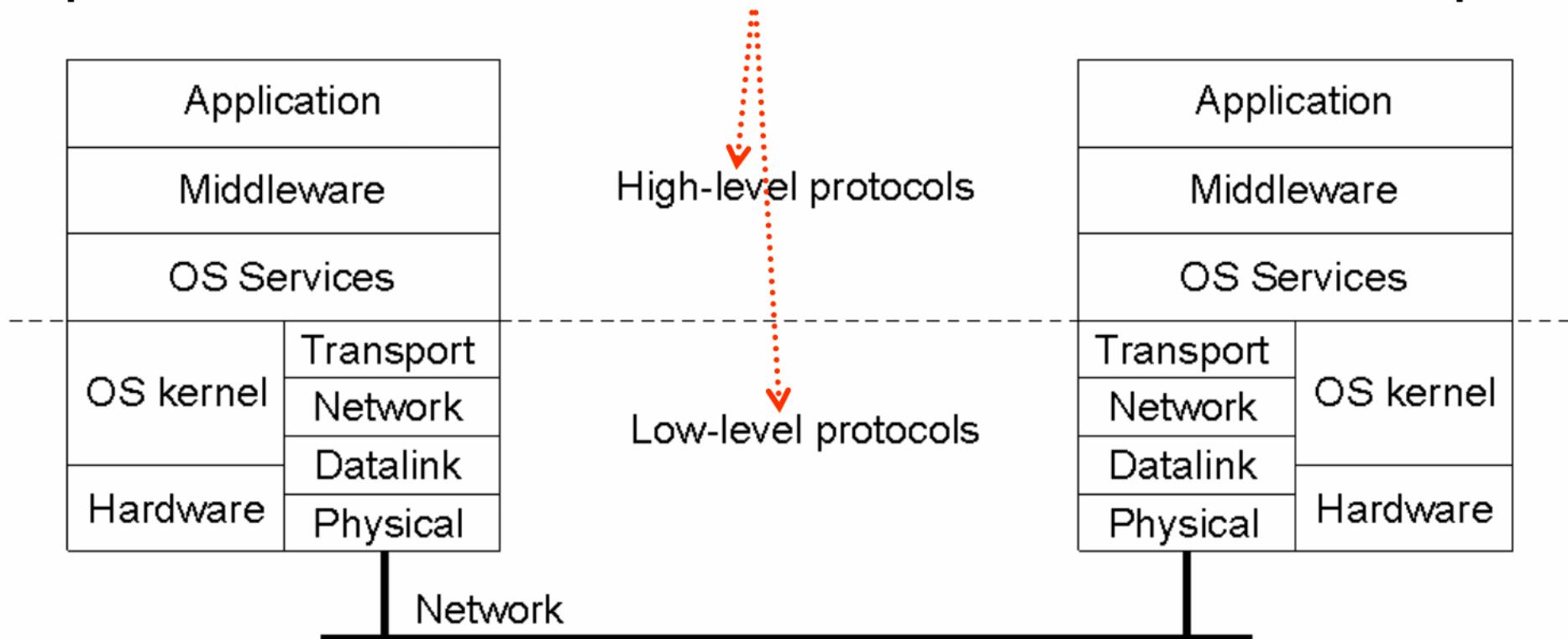


- (a) Protezione da operazioni non valide
- (b) Protezione da invocazioni non autorizzate
- (c) Protezione da utenti non autorizzati



::: Modello di sicurezza (5/6)

A quale livello i meccanismi di sicurezza devono essere posti?

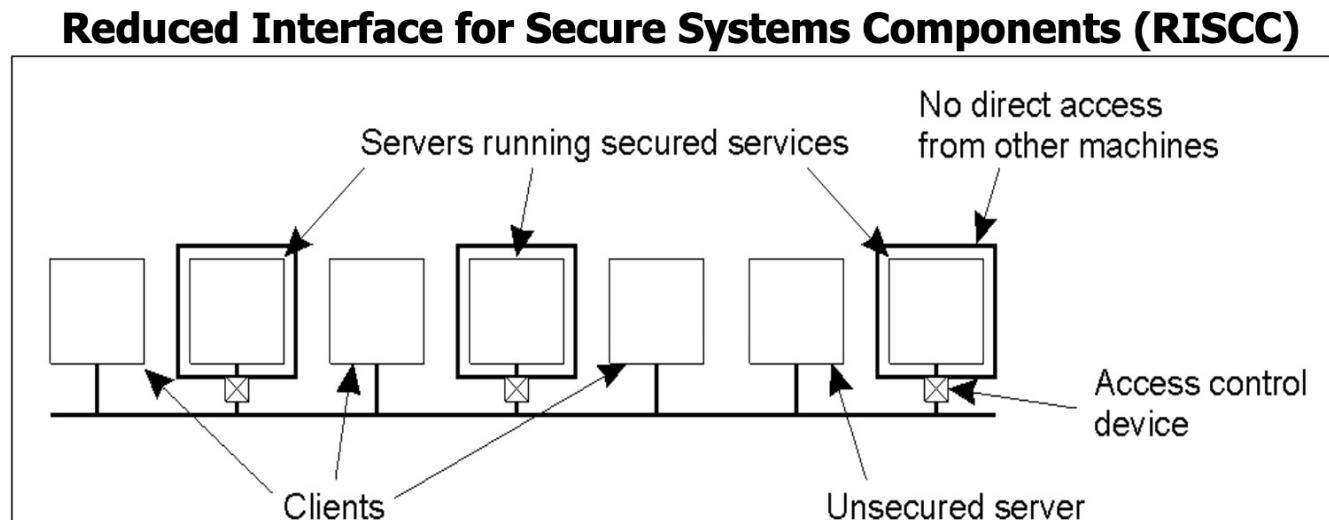


NB: I meccanismi di sicurezza nei sistemi distribuiti sono generalmente posti a livello del middleware.

::: Modello di sicurezza (6/6)

Le dipendenze tra servizi di sicurezza portano al concetto di *Trusted Computing Base* (TCB), ovvero l'insieme di tutti i meccanismi di sicurezza in un sistema distribuito che sono necessari per rispettare la sicurezza del sistema.

I servizi di sicurezza possono essere isolati da altri tipi di servizi, reducendo la TCB ad un piccolo sotto-insieme di nodi.



Il principio della RISCC applicato ad un sistema distribuito sicuro



Il Consenso nei Sistemi Distribuiti

::: Il problema del consenso (1/3)

Informalmente, il problema del consenso distribuito consiste nel far sì che alcuni processi convengano su un valore, dopo che almeno uno di essi ha effettuato una proposta riguardo a tale valore.

Meno informalmente:

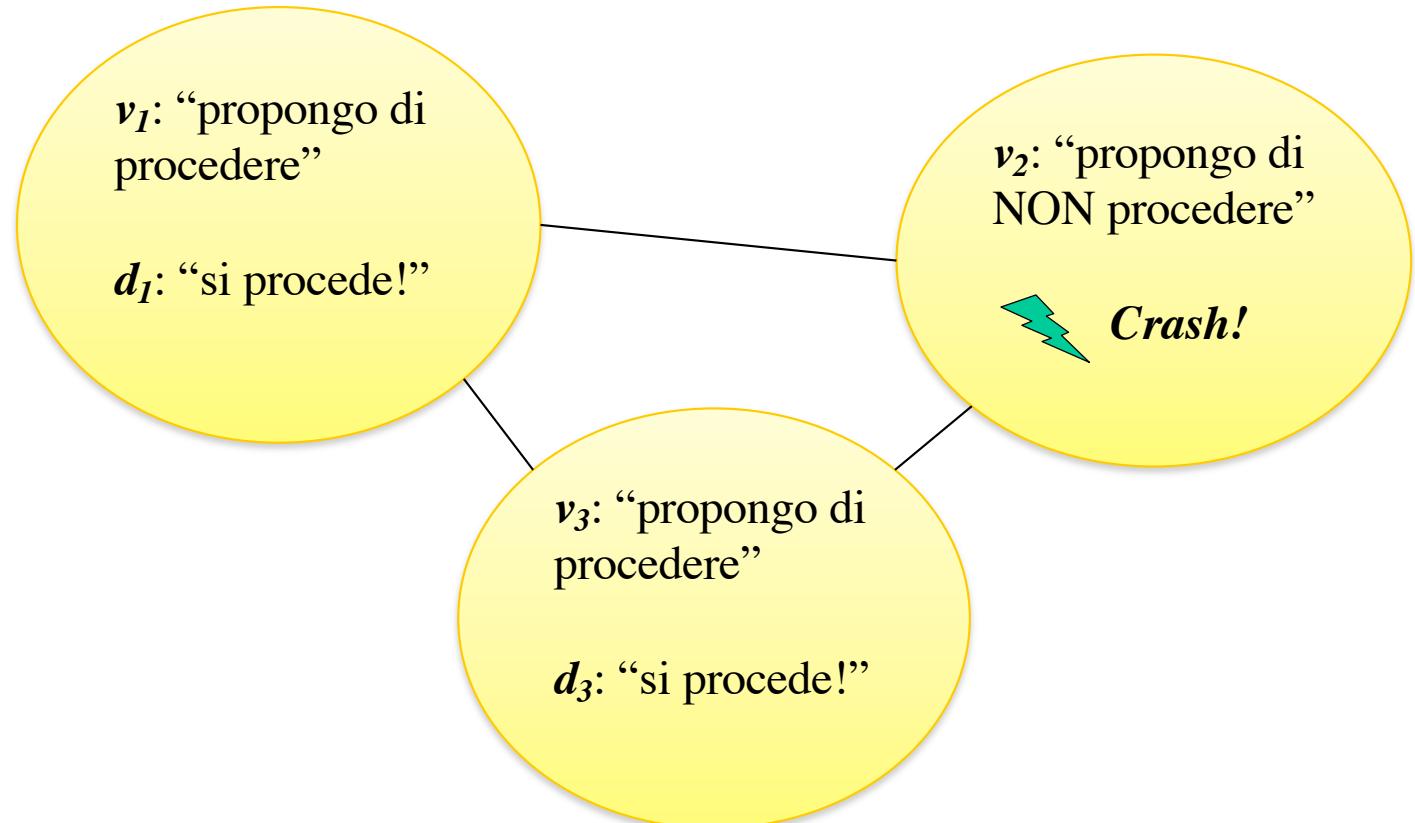
- N processi comunicanti tramite scambio messaggi;
- i canali di comunicazione sono affidabili;
- i processi sono soggetti a fallimenti di tipo *crash* oppure bizantini;
- il processo p_i ($i=1 \dots N$) possiede una propria **variabile di decisione** d_i , e propone un proprio valore v_i appartenente a un insieme D ;
- ciascun processo parte da uno stato di indecisione, e mira a raggiungere uno stato di decisione, in cui il valore di d_i è immutabile.

::: Il problema del consenso (2/3)

Un esempio con tre processi:

- i processi p_1 e p_3 propongono di *procedere* ad un'azione comune;
- il processo p_2 propone di *abortire* l'azione comune, ma poi subisce un *crash*.

Un algoritmo di consenso distribuito fa sì che i due processi corretti decidano di procedere.



::: Il problema del consenso (3/3)

I requisiti che devono essere soddisfatti da un algoritmo di consenso distribuito in ogni sua esecuzione sono:

- **Terminazione** (*termination*): prima o poi ogni processo corretto prende una decisione;
- **Accordo** (*agreement*): due qualsiasi processi corretti non decidono diversamente;
- **Integrità** (*integrity*) o Validità (*validity*): se tutti i processi corretti propongono lo stesso valore, la decisione finale di ogni processo corretto corrisponde a quel valore.

Si possono avere varianti al requisito di integrità: per es., un requisito di integrità più debole richiede che il valore di decisione sia proposto da qualche processo corretto, ma non necessariamente da tutti i processi corretti.

::: Il problema del consenso (3/3)

I requisiti che devono essere soddisfatti da un algoritmo di consenso distribuito in ogni sua esecuzione sono:

- **Terminazione** (*termination*): prima o poi ogni processo corretto prende una decisione;
- **Accordo** (*agreement*): due qualsiasi processi corretti non decidono diversamente;
- **Integrità** (*integrity*) o Validità (*validity*): se tutti i processi corretti propongono lo stesso valore, la decisione finale di ogni processo corretto corrisponde a quel valore.

In alcune formulazioni possiamo trovare anche la coppia di proprietà: **Safety**, indica che qualcosa di “brutto” non avverrà durante l’esecuzione dell’algoritmo, e **Liveness**, indica che qualcosa di “bello” avverrà durante l’esecuzione dell’algoritmo.

::: Il problema del consenso (3/3)

I requisiti che devono essere soddisfatti da un algoritmo di consenso distribuito in ogni sua esecuzione sono:

- **Terminazione** (*termination*): prima o poi ogni processo corretto prende una decisione;
- **Accordo** (*agreement*): due qualsiasi processi corretti non decidono diversamente;
- **Integrità** (*integrity*) o Validità (*validity*): se tutti i processi corretti propongono lo stesso valore, la decisione finale di ogni processo corretto corrisponde a quel valore.

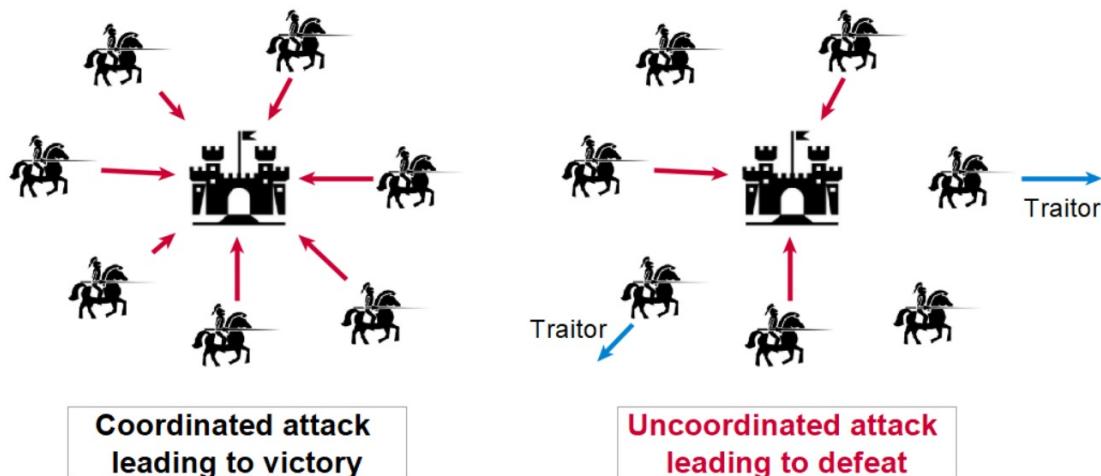
Tutti i validatori raggiungeranno un certo consenso su un valore proposto dai processi corretti, senza divergenze.

In alcune formulazioni possiamo trovare una coppia di proprietà: **Safety**, indica che qualcosa di “**brutto**” non avverrà durante l’esecuzione; e **Liveness**, indica che qualcosa di “**bello**” avverrà.

Tutti i validatori raggiungeranno un certo consenso entro un tempo utile, ovvero non si ha stallo nel consenso.

::: Comportamento Bizantino

Un difetto bizantino è una condizione di un sistema distribuito in cui i componenti possono guastarsi e c'è imperfetta conoscenza sull'eventuale guasto di un componente.



Il termine prende il nome dal "Problema dei generali bizantini", che descrive una situazione in cui, per evitare un fallimento catastrofico del sistema, gli attori devono concordare una strategia, ma alcuni sono inaffidabili.

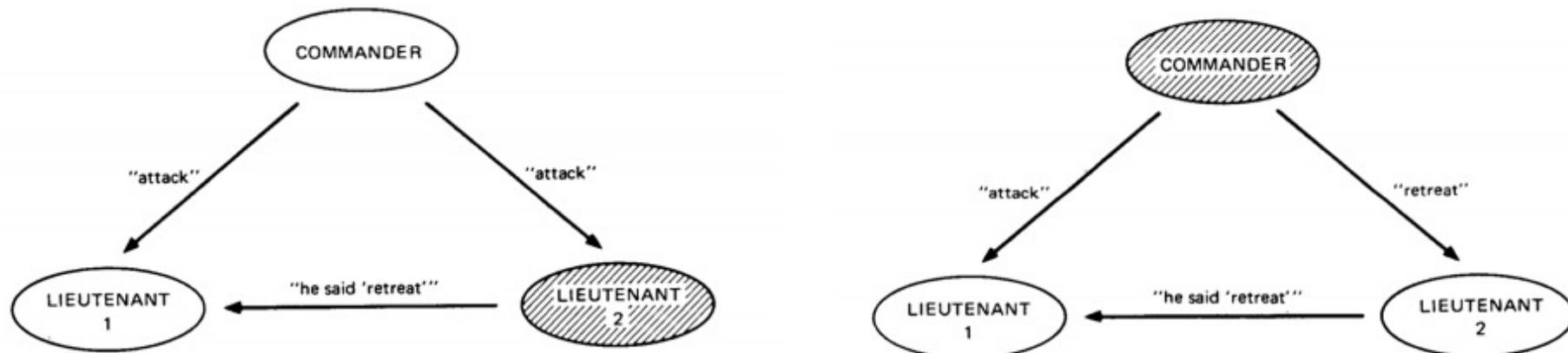
In un errore bizantino, un componente può apparire incoerentemente sia guasto che funzionante ai sistemi di rilevamento degli errori, presentando sintomi diversi a diversi osservatori. È difficile dichiararlo guasto e escluderlo dalla rete, perché devono prima raggiungere un consenso su quale componente ha fallito per primo.

::: Il problema dei generali bizantini (1/3)

Nella formulazione informale di Lamport et al. (1982), tre o più generali devono convenire se sferrare un attacco o ritirarsi. Uno di essi è il comandante; gli altri sono i suoi luogotenenti.

- Il comandante invia l'ordine di attaccare o ritirarsi; i luogotenenti devono convenire se attaccare o ritirarsi.
- Uno o più generali possono essere maliziosi.
- Se il comandante è malizioso, invia ordini diversi ai luogotenenti.
- Se un luogotenente è malizioso, dice ad alcuni suoi pari che il comandante ha ordinato di attaccare, ad altri che ha ordinato di ritirarsi.

::: Il problema dei generali bizantini (2/3)



Il problema differisce da quello del consenso in quanto:

- non tutti i processi che devono raggiungere l'accordo propongono un valore, ma uno specifico processo (il generale) fornisce loro un valore sul quale devono convenire;
- tipicamente si assume che i generali siano soggetti a fallimenti arbitrari (bizantini), sebbene il problema possa essere posto anche solo con riferimento a fallimenti per *crash*.

::: Il problema dei generali bizantini (3/3)

I requisiti del problema dei generali bizantini sono:

- **Terminazione** (*termination*): prima o poi ogni processo corretto prende una decisione;
- **Accordo** (*agreement*): due qualsiasi processi corretti non decidono diversamente;
- **Integrità** (*integrity*): se il comandante è corretto, tutti i processi corretti decidono per il valore proposto dal comandante.

Nel problema dei generali bizantini **l'integrità implica l'accordo se il comandante è corretto.**

Si possono avere varianti al requisito di integrità: per es., un requisito di integrità più debole richiede che il valore di decisione sia proposto da qualche processo corretto, ma non necessariamente da tutti i processi corretti.

::: Il problema della consistenza interattiva

È una variante del problema del consenso.

Ogni processo propone un solo valore, e l'obiettivo è che tutti i processi concordino su un vettore di valori, uno per ciascuno di essi.

I requisiti del problema della *interactive consistency* sono:

- **Terminazione** (*termination*): prima o poi ogni processo corretto prende una decisione;
- **Accordo** (*agreement*): il vettore di decisione è lo stesso per tutti i processi corretti;
- **Integrità** (*integrity*): se p_i è corretto, tutti i processi corretti decidono per il valore v_i proposto da p_i come elemento i -esimo del vettore.

::: Relazioni tra i problemi C, BG, IC (1/2)

I tre problemi sono equivalenti nel senso che una soluzione a ognuno di essi può essere utilizzato come soluzione agli altri due problemi.

$C_i(v_1, v_2, \dots, v_n)$ – Decisione di p_i in un'esecuzione di un algoritmo di **Consenso**.

$BG_i(j, v)$ – Decisione di p_i in un'esecuzione di un algoritmo dei **Gen. Bizantini**, ove p_j è il comandante

$IC_i(v_1, v_2, \dots, v_n)[j]$ – j-esimo valore del vettore di decisione di p_i in una esecuzione di un algoritmo di **Interactive Consist**.

BG → IC

n esecuzioni di BG. $\forall i \in \{1, \dots, n\}$ p_i è il generale. Si ha:

$$IC_i(v_1, v_2, \dots, v_n)[j] = BG(j, v_j) \quad (i, j = 1, 2, \dots, n)$$

::: Relazioni tra i problemi C, BG, IC (2/2)

IC → C

Applicando una funzione di valutazione sul vettore di soluzioni prodotto da IC si ha:

$$C_i(v_1, v_2, \dots, v_n) = \text{majority} (IC_i(v1, v2, \dots, vn)[j]) \quad (j = 1, 2, \dots, n)$$

C → BG

- 1) Il comandante p_j manda v a sé stesso e agli altri $n-1$ processi.
- 2) Tutti i processi eseguono C con i valori v_1, \dots, v_n ricevuti
- 3) $BG_i(j, v) = C_i(v_1, v_2, \dots, v_n) \quad (i = 1, 2, \dots, n)$

Le proprietà di *Agreement*, *Integrity* e *Termination* continuano a valere dopo di tali trasformazioni.

::: Consenso in un sistema sincrono (1/5)

Un algoritmo di consenso basato su *multicast*.

Ipotesi:

- sistema sincrono,
- N processi,
- al più f processi esibiscono fallimenti per crash,
- disponibilità di una primitiva **basic multicast** (*B-multicast*), che garantisce che i processi destinatari corretti ricevano il messaggio, finché il mittente (*multicaster*) non fallisce:

multicast(g, m) invia il messaggio m a un gruppo g di processi;

m porta con sé un identificativo del mittente, e un identificativo del gruppo g ;

deliver(m) consegna al chiamante il messaggio m ;

i processi non mentono su origine e destinazione dei messaggi.

::: Consenso in un sistema sincrono (2/5)

- V_i^k : vettore dei valori proposti noti al processo p_i all'inizio del ciclo k ;
 f numero di guasti per crash tollerati dall'algoritmo
 $f+1$ numero complessivo di iterazioni

Nel caso peggiore, nelle $f+1$ iterazioni si verifica il numero massimo di *crash* possibili f ; l'algoritmo garantisce che al termine i processi corretti sopravvissuti raggiungano il consenso.

Nell'iterazione k , il processo p_i ($i=1..N$):

- invia (*B-multicast*) i valori che non ha inviato nei cicli precedenti;
- riceve (*B-deliver*) i messaggi analoghi ed aggiorna V_i^k con i nuovi valori ricevuti.

Dopo $f+1$ cicli, ogni processo sceglie il valore minimo di V_i^{f+1} come valore di decisione.

La durata di un ciclo è limitata da un apposito *timeout* (sistema sincrono).

::: Consenso in un sistema sincrono (3/5)

Algoritmo di Dolev et al.

Algoritmo eseguito dal generico processo p_i

Inizializzazione:

$$V_i^0 = \{\}$$

$$V_i^1 = \{v_i\}$$

k -esima iterazione, $1 \leq k \leq f+1$: ($f+1$ cicli)

B-multicast($V_i^k - V_i^{k-1}$); *invia solo i valori non ancora inviati*

$$V_i^{k+1} = V_i^k;$$

while (in iterazione k) {

al B-deliver(V_j) da un processo p_j :

$$V_i^{k+1} = V_i^{k+1} \cup V_j;$$

}

Dopo $f+1$ iterazioni:

$$d_i = \minimo(V_i^{f+1});$$

::: Consenso in un sistema sincrono (4/5)

Terminazione. Ovvia (il sistema è sincrono).

Accordo e integrità: Sono raggiunti se si dimostra che tutti i processi sopravvissuti pervengono allo stesso vettore al termine dell'algoritmo, in quanto poi tutti applicano la stessa funzione (calcolo del minimo).

Dim.: assumiamo che due processi non abbiano lo stesso vettore finale, p. es. che p_i abbia un valore v che p_j non possiede ($i \neq j$).

L'unica spiegazione possibile è che v sia stato inviato da un altro processo p_k a p_i , e poi p_k sia andato in *crash* prima di inviarlo a p_j .

Ma se p_k possedeva il valore v senza che p_j lo possedesse anch'esso già a conclusione del ciclo precedente, vuol dire che ci deve essere stato nel ciclo precedente un altro processo ancora che ha inviato v a p_k ed è andato in *crash* prima di inviarlo a p_j .

Iterando il ragionamento, ci deve essere stato almeno un *crash* per ciclo.

Ma ci sono al più f guasti, e $f+1$ cicli, e l'ipotesi d'assurdo è contraddetta.

::: Consenso in un sistema sincrono (5/5)

Il risultato è generalizzabile:

Ogni algoritmo di consenso distribuito che voglia tollerare f fallimenti richiede almeno $f+1$ cicli (Dolev e Strong, 1983).

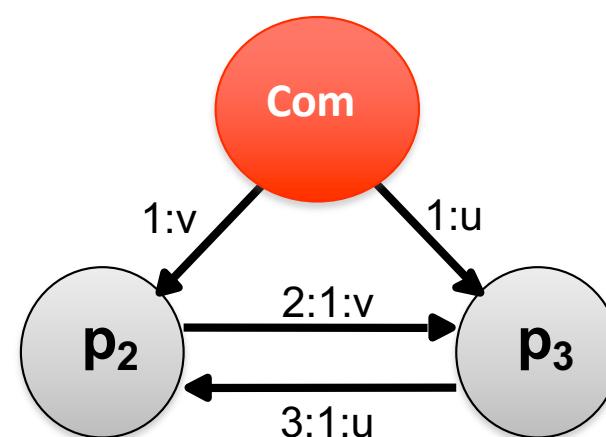
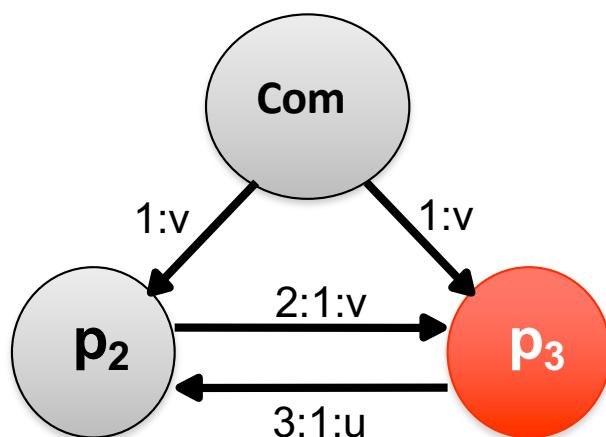
Ne segue che tale limite inferiore si applica anche agli altri problemi di *agreement* distribuito, come quello dei generali bizantini.

::: Problema con tre generali (1/2)

Si assume che il sistema sia **sincrono**, e che i canali siano privati (se i processi potessero ispezionare i messaggi degli altri processi, potrebbero riconoscere il processo bizantino).

Notazione: **i:j:v = i dice che j dice v.**

I processi **in rosso** sono bizantini.



::: Problema con tre generali (2/2)

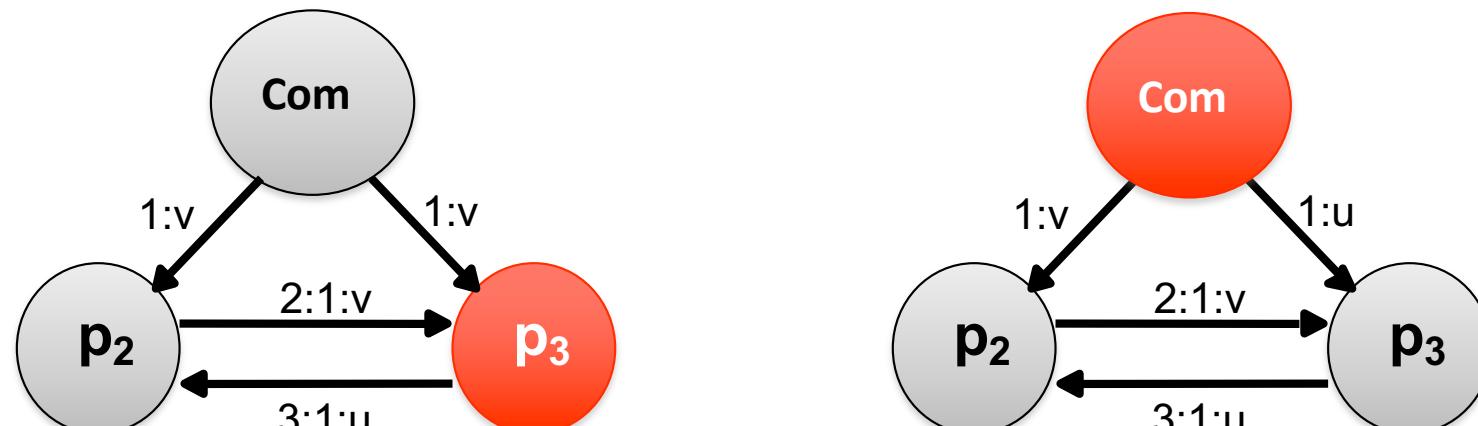
Se esistesse una soluzione, per il requisito di integrità p_2 dovrebbe scegliere il valore v fornito dal comandante se questo è corretto.

Ma nei due casi, p_2 è nella stessa situazione.

Analogamente, per simmetria si ha che se p_3 è corretto, deve scegliere il valore del comandante.

Ma ciò viola la condizione di *agreement*, perché se il comandante è *faulty* invia valori diversi ai due luogotenenti.

In generale, non esiste soluzione se $N \leq 3f$.

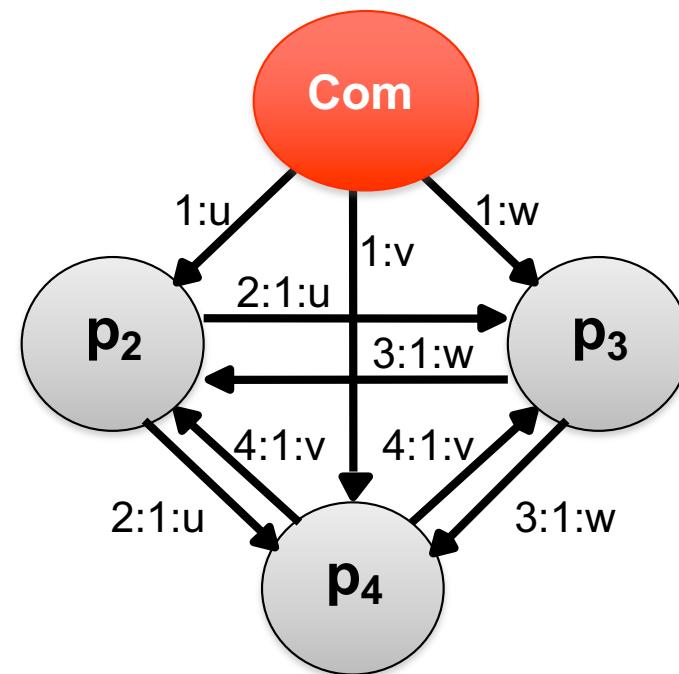
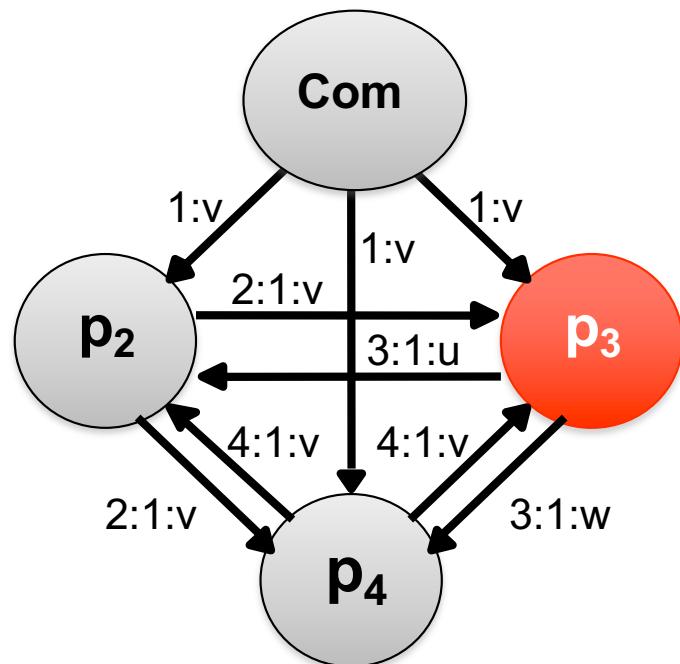


I processi in rosso sono bizantini; Com è il Comandante, p_2 , p_3 , p_4 i luogotenenti.

::: Problema con quattro generali

Esiste soluzione se $N \geq 3f + 1$.

Analisi con $N = 4, f = 1 \rightarrow 2$ passi:



I processi in rosso sono bizantini; Com è il Comandante, p₂, p₃, p₄ i luogotenenti.

p₂: maggioranza(v,u,v)=v

p₄: maggioranza(v,v,w)=v

p₁, p₂, p₃: maggioranza(u,v,w)=∅

::: Consenso nei sistemi asincroni (1/2)

Garanzia in sistemi asincroni (FLP)

Non esiste alcun algoritmo deterministico in grado di garantire il raggiungimento del consenso in un sistema asincrono a scambio di messaggi nel caso di anche un solo fallimento per *crash* di un processo.

(Fischer, Lynch, Paterson)

Di conseguenza, non esistono soluzioni garantite in un sistema asincrono per il problema dei generali bizantini e per quello della consistenza interattiva.

NB: ciò non significa che non si possa raggiungere il consenso in un sistema asincrono, ma che non c'è algoritmo che possa garantire il raggiungimento.

::: Consenso nei sistemi asincroni (2/2)

- **Indebolire la condizione di Termination**
 - Introducendo elementi di non determinismo oppure garantendo la *termination* esclusivamente durante periodi di sincronia del sistema.
- **Indebolire la condizione di Agreement**
 - Individuando un insieme finito per i possibili valori decisionali dei singoli processi.
- **Irrobustire il modello del sistema**
 - Introducendo dei *failure detectors* per distinguere i processi lenti (ma corretti) da quelli effettivamente falliti.

::: Sommario dei risultati sul consenso

Modo di Fallimento	Sistema Sincrono	Sistema Asincrono
Nessuno	Consenso ottenibile Conoscenza comune anche ottenibile	Consenso ottenibile Conoscenza commune concorrente ottenibile
Crash	Consenso ottenibile per $f < n$ processi non corretti con $\Omega(f+1)$ cicli	Consenso non ottenibile
Bizantino	Consenso ottenibile per $f \leq (n-1)/3$ processi bizantini con $\Omega(f+1)$ cicli	Consenso non ottenibile

::: Isola di Paxos

“Recenti scoperte archeologiche sull’isola di Paxos rivelano che il parlamento ha funzionato nonostante la propensione peripatetica dei suoi legislatori part-time.

[...]

All’inizio di questo millennio, l’isola egea di Paxos era un fiorente centro mercantile. La ricchezza portò alla sofisticazione politica e i Paxon sostituirono la loro antica teocrazia con una forma di governo parlamentare.

[...] ”



::: Isola di Paxos

Paxos è una famiglia di protocolli di consenso, che garantiscono la coerenza delle decisioni in un sistema distribuito di processori inaffidabili (ovvero i nodi possono fallire). Il protocollo Paxos è stato introdotto nel 1989 da Leslie Lamport, dal nome di un sistema di consenso legislativo immaginario utilizzato sull'isola di Paxos in Grecia. L'obiettivo dell'algoritmo è mantenere lo stesso ordinamento dei comandi tra più repliche in modo che tutte le repliche alla fine convergano allo stesso valore.



::: Parlamento part-time (1/6)

- I decreti approvati erano numerati (in ordine crescente).
- L'acustica dell'aula era scadente, rendendo impossibile l'oratoria, così i legislatori poteva comunicare tra loro solo tramite messaggi portati da messaggeri.
- I parlamentari, così come i messaggeri, potevano entrare ed uscire dal parlamento a piacere. I messaggeri potevano anche uscire prima di consegnare un messaggio affidatogli.
- Ma quando nell'aula parlamentare, parlamentari e messaggeri erano dediti al lavoro ed eseguivano prontamente i loro compiti.
- Il compito principale del Parlamento era quello di determinare la legge del paese, definita dalla successione dei decreti approvati.
- Un parlamento moderno impiegherà un segretario per registrare le sue azioni, ma nessuno a Paxos era disposto a rimanere per tutta la durata della seduta a svolgere le funzioni di segretario.

::: Parlamento part-time (2/6)

- I legislatori potevano dimenticare ciò che avevano fatto se lasciavano l'aula parlamentare, quindi scriveranno note su dei registro per ricordare a se stessi importanti compiti parlamentari, come una sequenza numerata dei decreti approvati.
 - Ad esempio, il libro mastro del legislatore *Λινχδ* riportava la voce «155: L'imposta sulle olive è di 3 dracme per tonnellata» se riteneva che il 155º decreto approvato dal Parlamento fissasse l'imposta sulle olive a 3 dracme per tonnellata.
- I libri mastri erano scritti con inchiostro indelebile e le loro voci non potevano essere modificate, ma solo cancellate.

Il primo requisito era la coerenza dei libri mastri: due registri non potevano contenere informazioni contraddittorie.

- Se il legislatore *Φισδερ* avesse avuto la voce «132: Le lampade devono usare solo olio d'oliva» nel suo registro, allora nessun altro registro di un legislatore avrebbe potuto avere una voce diversa per il decreto 132.

::: Parlamento part-time (3/6)

- Tuttavia, un altro legislatore potrebbe non avere una voce nel suo registro per il decreto 132, se non aveva ancora saputo che il decreto era stato approvato.
- La coerenza dei libri mastri non era sufficiente, perché poteva essere banalmente soddisfatta lasciando tutti i libri mastri vuoti.
- Era necessario qualche requisito per garantire che i decreti venissero infine approvati e registrati nei libri mastri.
- Nei parlamenti moderni, l'approvazione dei decreti è ostacolata dal disaccordo tra i legislatori. Non era così a Paxos, dove prevaleva un'atmosfera di fiducia reciproca.
- I legislatori erano disposti ad approvare qualsiasi decreto proposto. Tuttavia, la loro propensione peripatetica poneva un problema.

::: Parlamento part-time (4/6)

- Ciò poneva un problema:
 - Se dei parlamentari decretavano che “37. È proibito dipingere sulle pareti del Tempio” per poi abbandonare il parlamento per un banchetto. Un altro gruppo di legislatori, appena entrato in parlamento e ignari di quanto appena deciso, poteva far passare il decreto, in conflitto con il precedente, “37. È garantita libertà d'espressione artistica”.
- Il progresso non potrebbe essere garantito a meno che un numero sufficiente di legislatori non rimanesse nel parlamento per un tempo sufficientemente lungo. Poiché i legislatori non erano disposti a ridurre le loro attività esterne, era impossibile garantire che qualsiasi decreto sarebbe mai stato approvato.
- Il comportamento corretto dei legislatori e i loro assistenti ha permesso al protocollo parlamentare di soddisfare la condizione di progresso.

::: Parlamento part-time (5/6)

- Se la maggioranza dei legislatori si trovasse in parlamento per una certa seduta, e nessuno entrasse o uscisse dall'aula per un periodo di tempo sufficientemente lungo, allora qualsiasi decreto proposto da un legislatore sarebbe approvato, e ogni decreto che fosse stato emanato comparirebbe in ogni registro.
- Il raggiungimento della condizione di progresso richiedeva che i legislatori fossero in grado di misurare il passare del tempo, quindi venivano forniti semplici orologi a clessidra.
- Tuttavia questo meccanismo aveva un punto di debolezza relativamente all'elezione dei nuovi membri del parlamento, che avveniva utilizzando le regolari procedure.

::: Parlamento part-time (6/6)

- Un giorno, a causa di un errore, passò una legge che asseriva che gli unici membri del parlamento erano dei marinai periti in un incidente navale.
- 
- Da quel momento il parlamento fu abbandonato e la civiltà di Paxos tramontò rapidamente, finché un generale di nome *Λαμπσων* prese possesso dell'isola con un colpo di stato instaurando una dittatura militare.

::: Analogie con il problema del consenso

Parlamento	→	Sistema Distribuito
Parlamentari	→	Processi
Uscita/Entrata dalla/nella Camera	→	Fallimento / Recovery
Registro	→	Memoria stabile
Messaggeri	→	Comunicazione asincrona tra i processi

::: Proprietà del consenso Paxos

Liveness

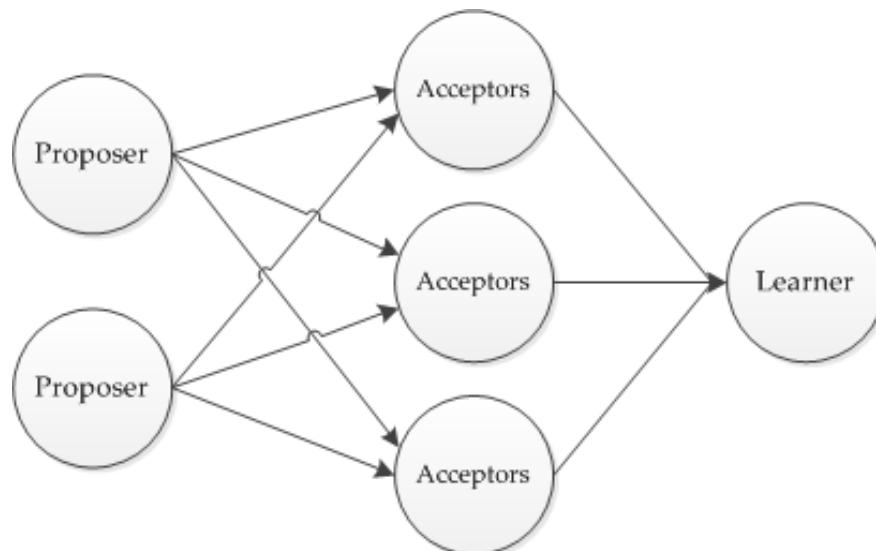
- Uno tra i valori proposti prima o poi viene scelto.
- Se un valore viene scelto, ogni processo prima o poi apprenderà tale scelta.

Safety

- Un valore può essere scelto solo tra quelli proposti.
- Il valore scelto deve essere unico.
- Un processo non deve mai apprendere che un valore è stato scelto a meno che esso non sia stato effettivamente scelto.

::: Ruoli dei processi

- Ogni processo può svolgere uno o più dei seguenti ruoli:
 - **Proposer**: Svolge tale ruolo un processo che ha la facoltà di proporre un valore.
 - **Acceptor**: Svolge tale ruolo un processo che ha la facoltà di accettare un valore precedentemente proposto da un *proposer*.
 - **Learner**: Svolge tale ruolo un processo che ha la facoltà di apprendere la scelta di un valore effettuata da un *acceptor*.



::: Ruoli dei processi

- Ogni processo può svolgere uno o più dei seguenti ruoli:
 - **Proposer**: Svolge tale ruolo un processo che ha la facoltà di proporre un valore.
 - **Acceptor**: Svolge tale ruolo un processo che ha la facoltà di accettare un valore precedentemente proposto da un *proposer*.
 - **Learner**: Svolge tale ruolo un processo che ha la facoltà di apprendere la scelta di un valore effettuata da un *acceptor*.
- Sistema asincrono, con processi non bizantini, eseguiti a velocità arbitrarie, e possono fallire (stop) e poi riprendere l'esecuzione.
 - Poiché possono fallire dopo che un valore è stato deciso e poi ripartire, è necessario che ricordino alcune informazioni anche dopo un *recovery* (altrimenti non c'è soluzione).
- I messaggi non hanno limiti di dimensioni, possono essere duplicati o persi, ma non corrotti.

::: Scelta di un valore (1/5)

- Il modo più semplice sarebbe quello di avere un solo *acceptor*.
 - Sarebbe un *single point of failure* (SPOF)
 - Il fallimento dell'acceptor renderebbe impossibile il progresso dell'algoritmo
- Quindi si considera un insieme di *acceptor*, e un valore proposto è scelto se un insieme sufficientemente grande di essi lo accetta.
- Quanto grande?
 - Si può considerare abbastanza grande una qualunque maggioranza, perché due maggioranze abbiano almeno un *acceptor* in comune
 - OK, a condizione che un *acceptor* accetti al più un valore

::: Scelta di un valore (2/5)

- In assenza di fallimenti o perdite di messaggi, è desiderabile che un valore venga scelto anche se proposto da un solo *proposer*.
- È quindi auspicabile che il seguente requisito sia soddisfatto:
P1: Un acceptor deve accettare la prima proposta che riceve.
- Tuttavia, se diversi *proposer* propongono valori diversi in tempi vicini, il requisito P1 potrebbe portare alla situazione in cui ogni *acceptor* ha accettato un valore ma nessun valore è scelto da una maggioranza di *acceptor*
 - Ad es., anche con due soli valori proposti, se ciascuno è accettato da circa la metà degli *acceptor*, il fallimento di uno potrebbe rendere impossibile apprendere quale è stato scelto

::: Scelta di un valore (3/5)

- P1 e il requisito che un valore sia scelto solo se accettato da una maggioranza di *acceptor* impongono quindi di ammettere che un *acceptor* possa accettare più di un valore.
- Per tener traccia delle diverse proposte accettate si associa ad ogni proposta un numero (naturale) di serie distinto
 - In effetti dunque una proposta è una coppia numero di serie - valore
- Un valore viene scelto quando una proposta con quel valore è accettata dalla maggioranza degli *acceptor*
 - In tal caso diciamo che la proposta è stata scelta (con il suo valore)

::: Scelta di un valore (4/5)

- Possiamo ammettere che siano scelte proposte differenti, a condizione che contengano lo stesso valore
- È sufficiente a tale scopo garantire che:

P2: Se viene scelta una proposta con valore v e numero seriale n, allora ogni proposta scelta con numero seriale $n' > n$ deve avere valore v.

- Affinché un valore sia scelto, deve essere accettato da almeno un acceptor.
- Si può allora soddisfare il requisito P2 richiedendo che:

P2^a: Se viene scelta una proposta con valore v e numero seriale n, allora ogni proposta con numero seriale $n' > n$ accettata da un qualsiasi acceptor deve avere valore v.

::: Scelta di un valore (5/5)

- La comunicazione asincrona può far sì che una proposta sia scelta da un *acceptor* quando un altro *acceptor* c non ha ancora ricevuto alcuna proposta.
- Un nuovo *proposer* (che si sveglia ignaro all'improvviso) potrebbe allora fare una proposta con numero di serie maggiore e con valore diverso, e $P1$ impone che c accetti tale valore (il primo ricevuto da c) violando $P2^a$.
- Mantenere $P1$ e $P2^a$ richiede dunque di restringere il requisito $P2^a$.
- Si considera allora il requisito più forte:

$P2^b$: Se viene scelta una proposta con valore v e numero seriale n , allora ogni proposta con numero seriale $n' > n$ effettuata da un qualsiasi proposer deve avere valore v .

::: Soddisfare P2^b (1/2)

- Si supponga che la proposta (m, v) sia stata scelta.
- Si vuole che ogni proposta con numero seriale $n > m$ abbia valore v .
- È possibile procedere per induzione su n
 - Si tratta di mostrare che ogni proposta con numero n ha valore v se ogni proposta effettuata con numero in $[m; n-1]$ ha valore v
- Si assuma che v sia il valore per ogni proposta con numero in $[m; n-1]$.
- Se (m, v) è stata scelta, esiste una maggioranza C di acceptor che ha accettato (m, v) .
- Dunque:
ogni acceptor in C ha accettato una proposta il cui numero è compreso in $[m; n-1]$, e ogni proposta accettata con numero in $[m; n-1]$ ha valore v .

... Soddisfare P2^b (2/2)

- Poiché **C** è una maggioranza degli acceptors, ogni insieme **S** che sia una maggioranza deve includere uno degli acceptors di **C**.
- Si può concludere che una proposta con numero n ha valore v se garantiamo il seguente invarianto:

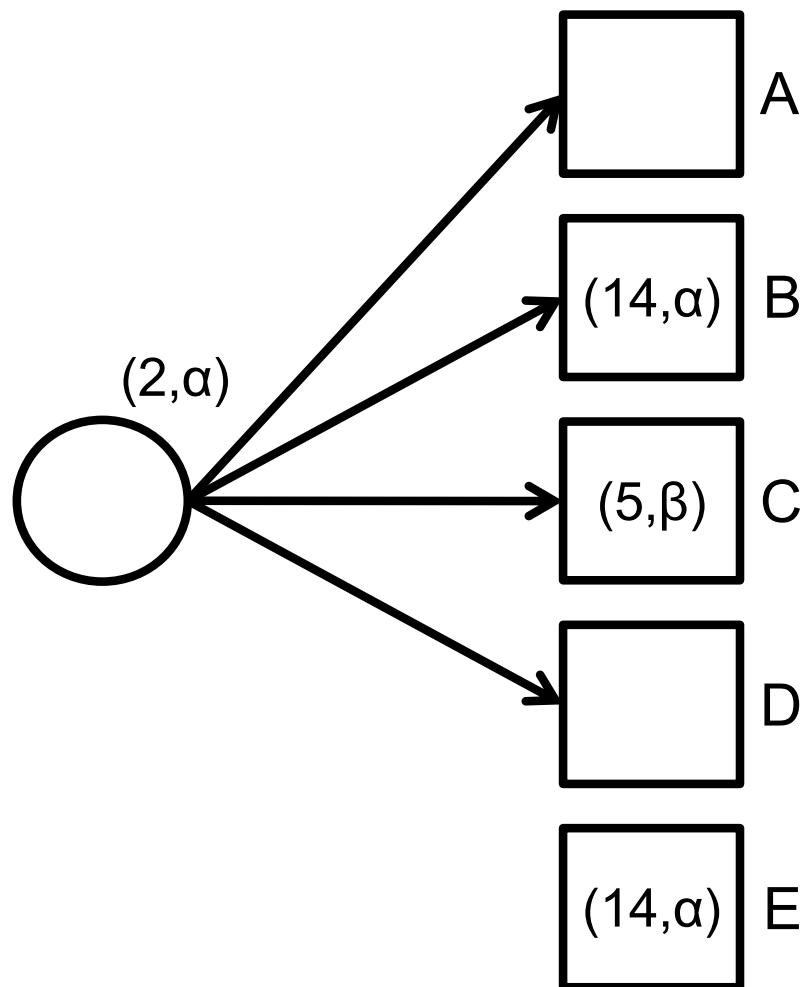
P2^c: Per ogni v ed n, se viene effettuata la proposta (n,v), esiste un insieme S di una maggioranza di acceptors tale che o:

(a) nessun acceptor di S ha accettato una proposta con numero seriale minore di n (v può essere qualsiasi), oppure

(b) v è il valore associato alla proposta con numero seriale più alto tra le proposte con numero minore di n accettate dagli elementi di S.

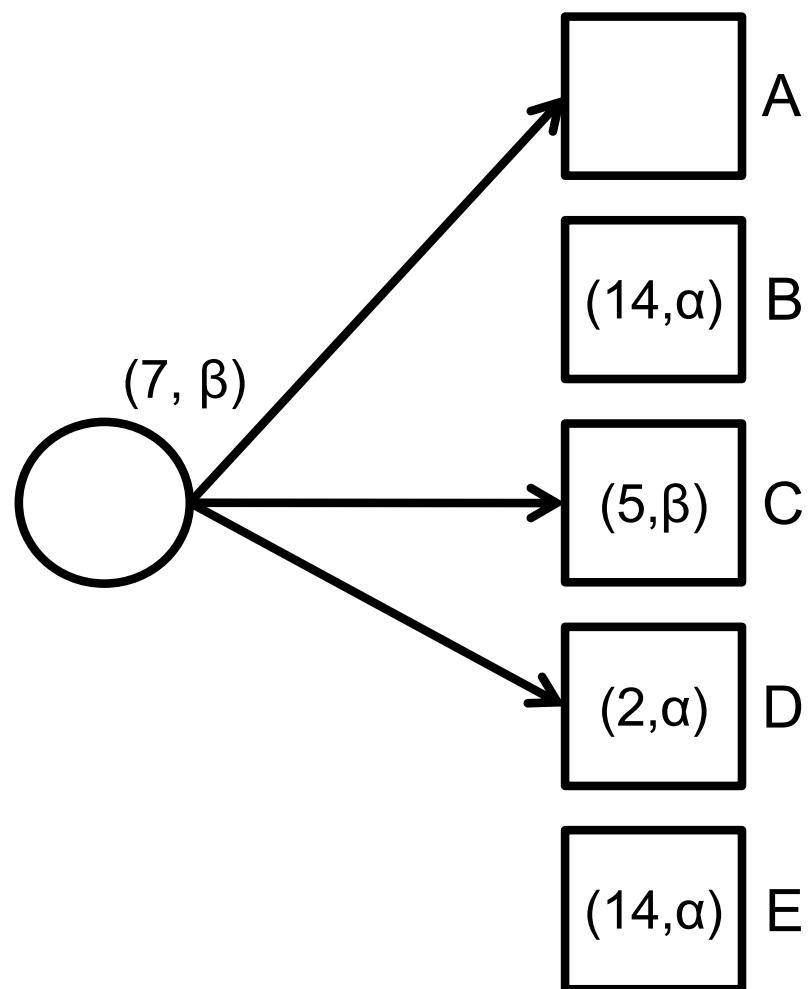
- Si può soddisfare **P2^b** mantenendo l'invariante **P2^c**

::: Esempio (1/2)



- In questo caso il *proposer* può proporre un qualsiasi valore perché nessun *acceptor* in S ha accettato alcuna proposta con numero seriale minore di 2.

::: Esempio (2/2)



- In questo caso il *proposer* deve proporre β , cioè il valore associato alla proposta con numero seriale più alto (5) tra quelle accettate con numero minore di 7 (5 e 2).

::: Generazione delle proposte (1/3)

- Affinché il requisito $P2^c$ sia soddisfatto, quando intende effettuare una proposta con numero n , un *proposer* deve conoscere (se esiste) il valore della proposta con numero seriale più alto minore di n :
 1. accettata da ogni *acceptor* in una maggioranza;
 2. che sarà accettato da ogni *acceptor* in una maggioranza.
- Mentre conoscere le proposte accettate è abbastanza semplice, prevedere le scelte future degli *acceptor* è piuttosto complicato...
- Invece di predire il futuro, il *proposer* di una proposta con numero n lo può controllare richiedendo agli *acceptor* la promessa di non accettare ulteriori proposte con numero minore di n .

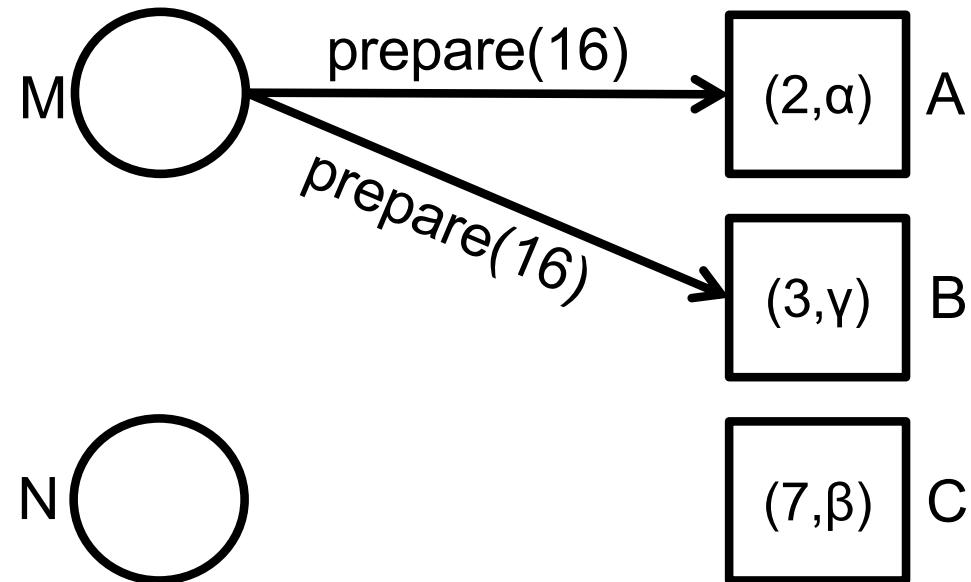
::: Generazione delle proposte (2/3)

- Richiesta di ***Prepare con numero n***:
Il *proposer* sceglie un numero **n** ed invia una richiesta ad ogni membro di un insieme di *acceptors* richiedendo:
 - la promessa di non accettare mai una richiesta con numero seriale inferiore a **n** (*promise*), e
 - il valore della proposta che esso ha già accettato con il più grande numero seriale minore di **n**, se presente.

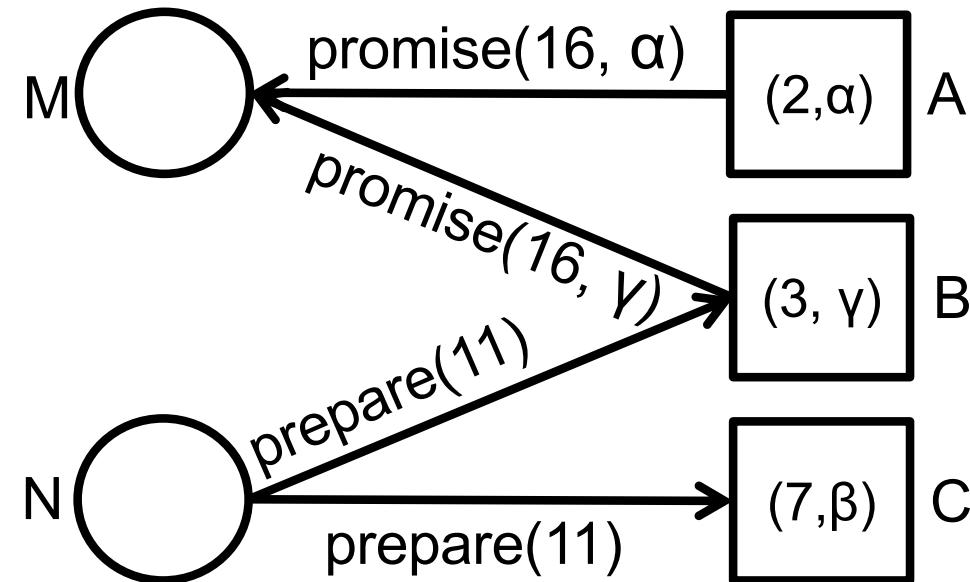
::: Generazione delle proposte (3/3)

- Richiesta di **Accept**:
Se il *proposer* riceve da una maggioranza degli *acceptors* risposta alla richiesta di *prepare*, invia ad essi una richiesta di *accept* per una proposta con numero **n**.
- Il valore **v** associato alla proposta può essere:
 - Il valore della proposta con numero seriale più alto tra quelle delle risposte degli *acceptors*,
 - oppure:
 - Un qualsiasi valore, nel caso in cui le risposte non riportavano proposte precedenti.

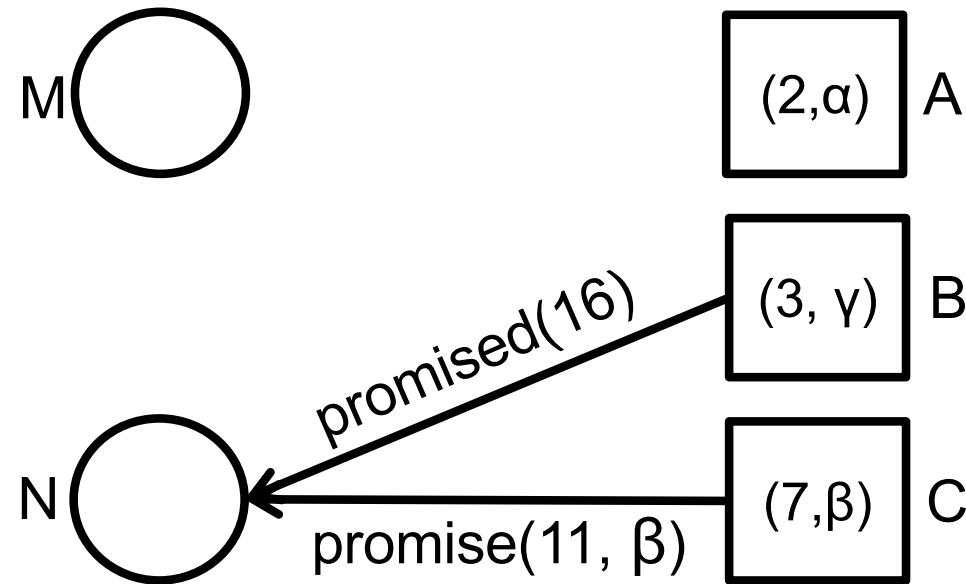
::: Esempio (1/4)



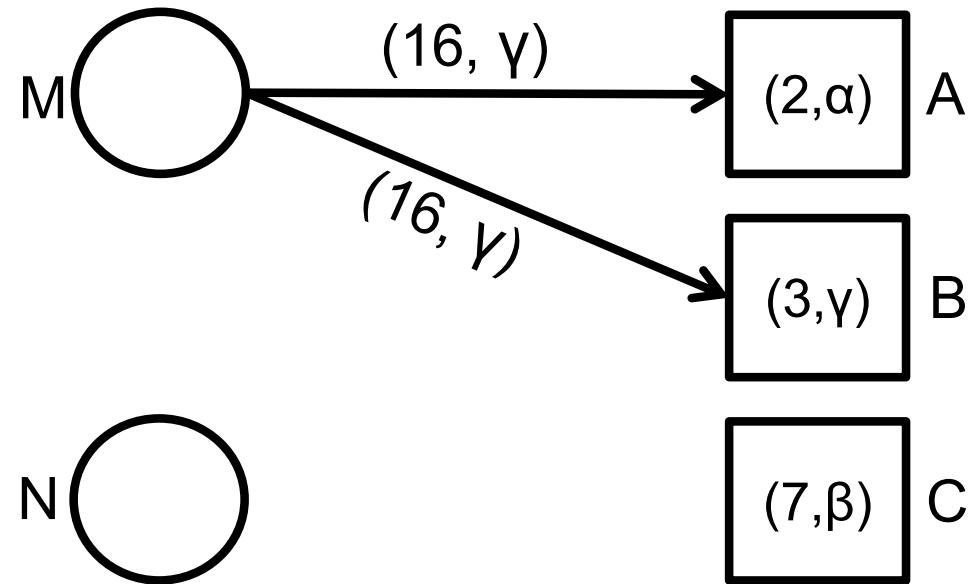
::: Esempio (2/4)



::: Esempio (3/4)



::: Esempio (4/4)



::: Comportamento degli acceptor (1/2)

- Gli *acceptor* possono ricevere due tipi di messaggi: *prepare* e *accept*.
- Un *acceptor* può sempre rispondere ad una richiesta di tipo *prepare*.
- Un *acceptor* può rispondere ad una richiesta di tipo *accept*, accettando la proposta, solo se non ha promesso a qualche processo di non farlo.
- Viene quindi formulato il seguente requisito:

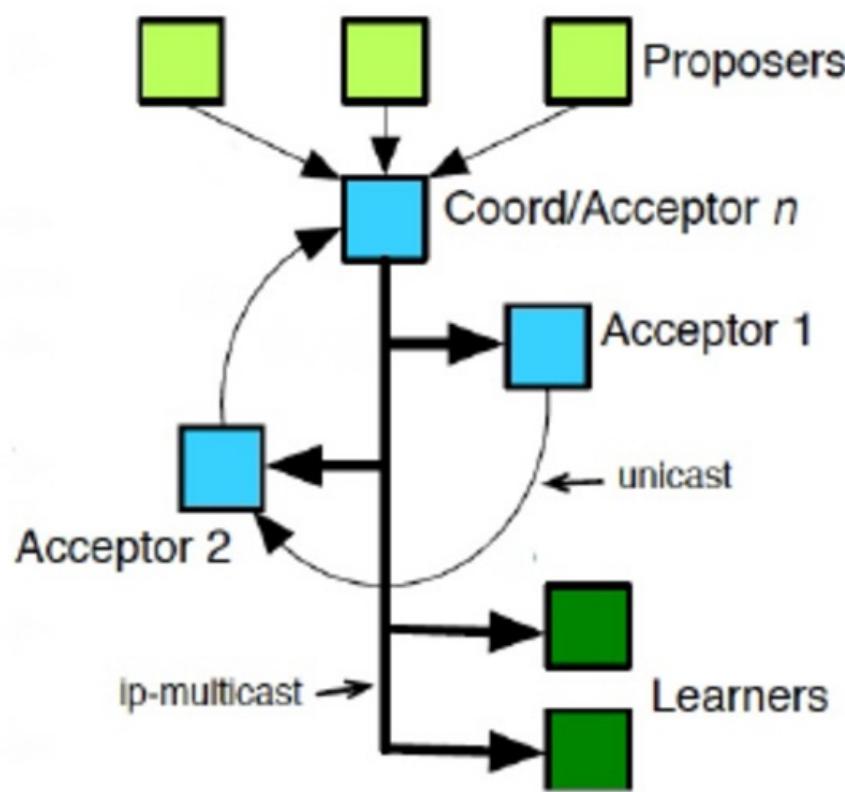
P1^a: Un acceptor può accettare una proposta con numero seriale n se e solo se non ha risposto ad una richiesta prepare il cui numero è maggiore di n.

::: Comportamento degli acceptor (2/2)

- Un *acceptor* può ignorare una richiesta *prepare* per la quale non è in grado di effettuare una promessa.
 - Ovvero può non rispondere a richieste di tipo *prepare(m)* ove sia stato già inviato un messaggio *promise(n)* con $n > m$.
- Un *acceptor* può ignorare richieste *prepare* relative a proposte già accettate (messaggi duplicati).
- Ogni *acceptor* è quindi tenuto a mantenere in memoria stabile (così da soddisfare l'invariante $P2^c$ anche in seguito al riavvio dopo un fallimento):
 - La proposta di numero seriale più elevato accettata;
 - Il numero seriale delle richiesta *prepare* con numero maggiore a cui ha risposto con un messaggio *promise*.

::: Leader tra gli acceptor

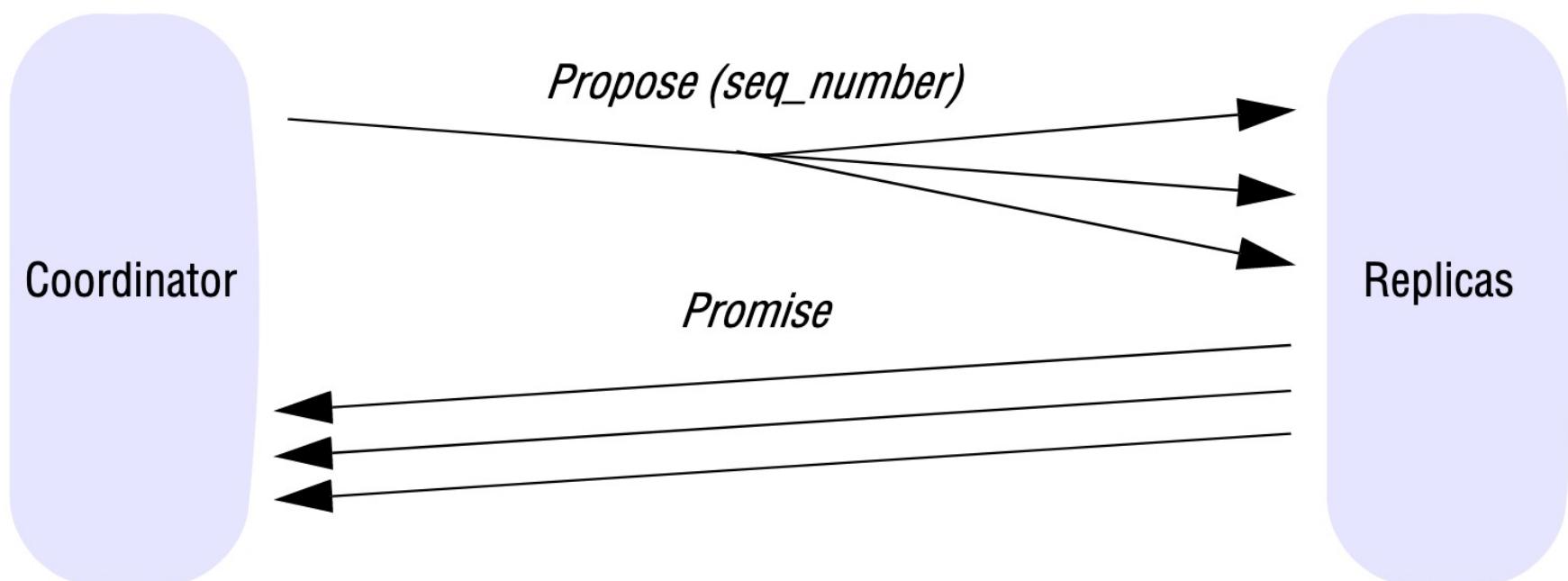
- Nella formulazione classica gli acceptor sono replicati attivamente, ma è anche possibile una formulazione passiva.



- Un coordinatore (eletto come $t \bmod n = s$) riceve le richieste dai proposers, e rappresenta il frontend del sistema di processi alla base di Paxos, e smista le richieste ai vari acceptors e learners.
- Il coordinatore proponega anche il valore per cui si è raggiunto il consenso.

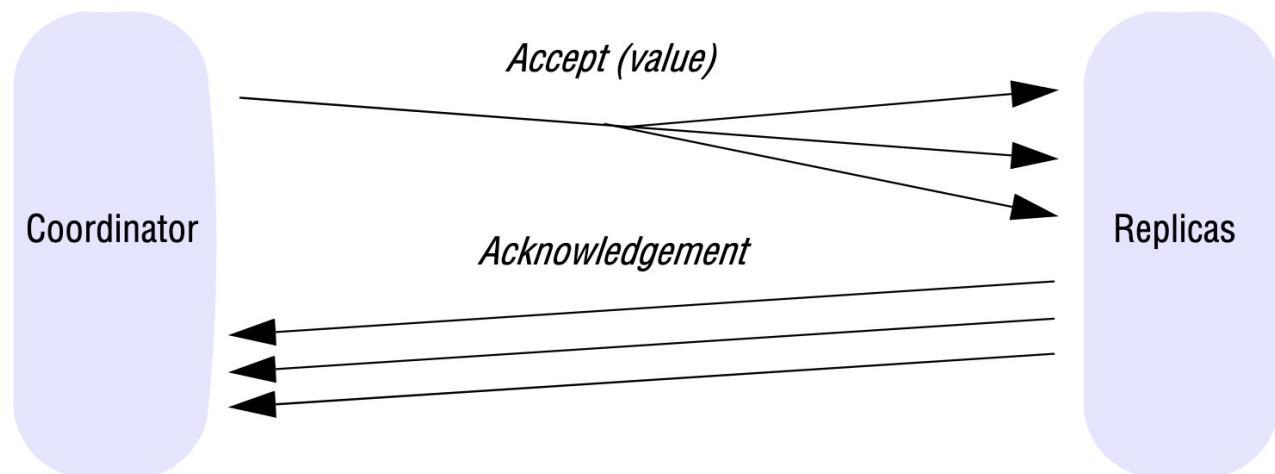
::: Algoritmo - Fase 1

- (a) Un *Proposer* seleziona un numero di proposta **n** ed invia una richiesta *prepare(n)* ad una maggioranza di *acceptors*.
- (b) Quando un *acceptor* riceve una *prepare(n)*, se **n** è maggiore di ogni altra *prepare* a cui ha già riposto, risponde con una *promise* di non accettare più proposte con numero seriale minore di **n** e con il valore della proposta già accettata con numero più grande.



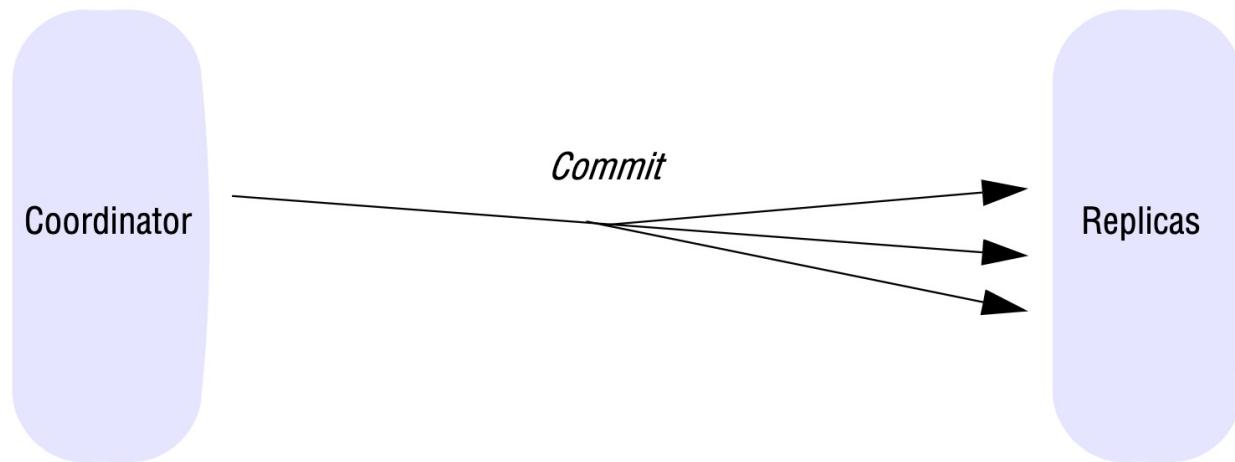
::: Algoritmo - Fase 2

- (a) Se il *proposer* riceve un messaggio $promise(n, v)$ da una maggioranza degli *acceptors*, invia un messaggio $accept(n, v)$ a tali *acceptors*, dove v è il valore della proposta con numero seriale più alto tra quelle riportate dagli *acceptors*, ovvero un qualsiasi valore, nel caso in cui le risposte non riportavano di una proposta precedente.
- (b) Se un *acceptor* riceve un messaggio $accept(n, v)$, accetta la proposta a meno che non abbia già risposto ad una richiesta *prepare* avente numero seriale maggiore di n .



::: Algoritmo - Fase 3

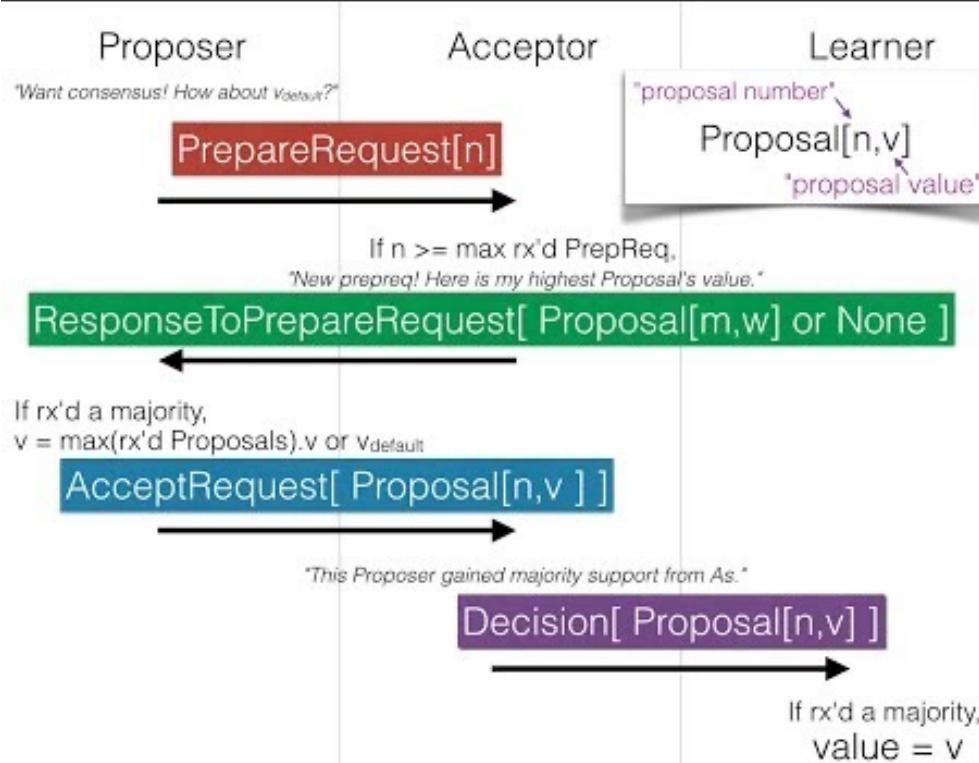
- (a) Se il *proposer* riceve tanti messaggi di $ack(n,v)$ quanta la maggioranza degli *acceptors*, allora risponde con un $commit(n,v)$ per comunicare il raggiungimento del consenso



::: Apprendimento

- Si può dimostrare che la fase 2 dell'algoritmo ha il minor costo possibile per un algoritmo di consenso in presenza di fallimenti.
- In altri termini, Paxos può essere considerato l'ottimo.

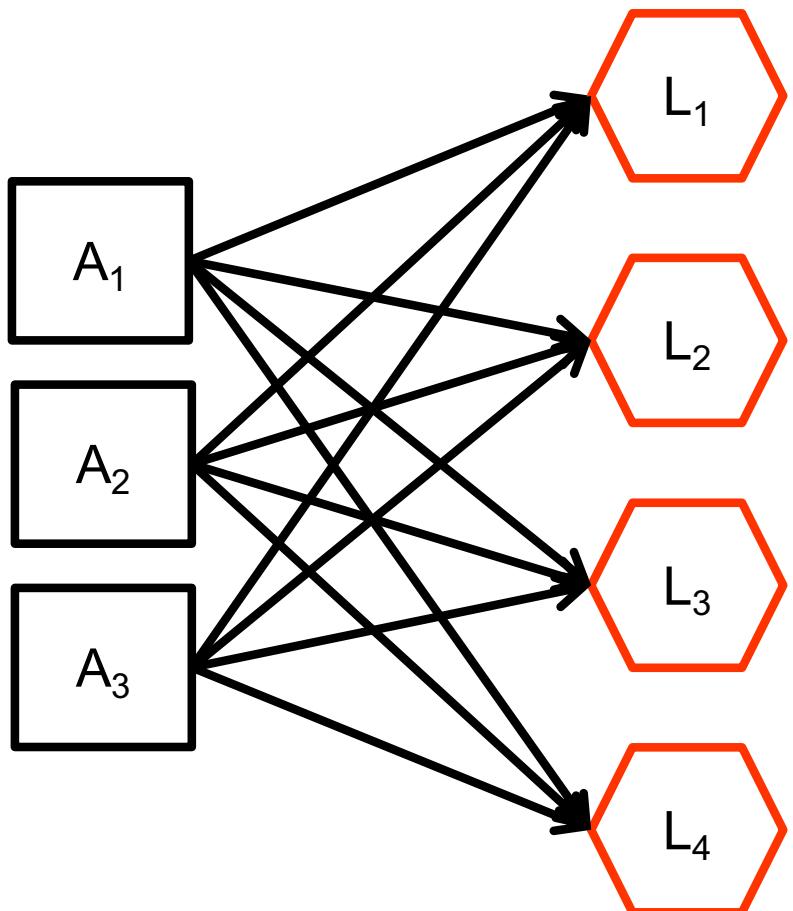
[Keidar, Rajsbaum – “On the cost of fault-tolerant consensus when there are no faults” – SIGACT News 32(2), 2001]



- Per garantire la consistenza dello stato degli acceptors e disaccoppiare dal proposer la fase decisionale sul consenso, entrano in gioco i processi *learner*.

::: Esempio (1/3)

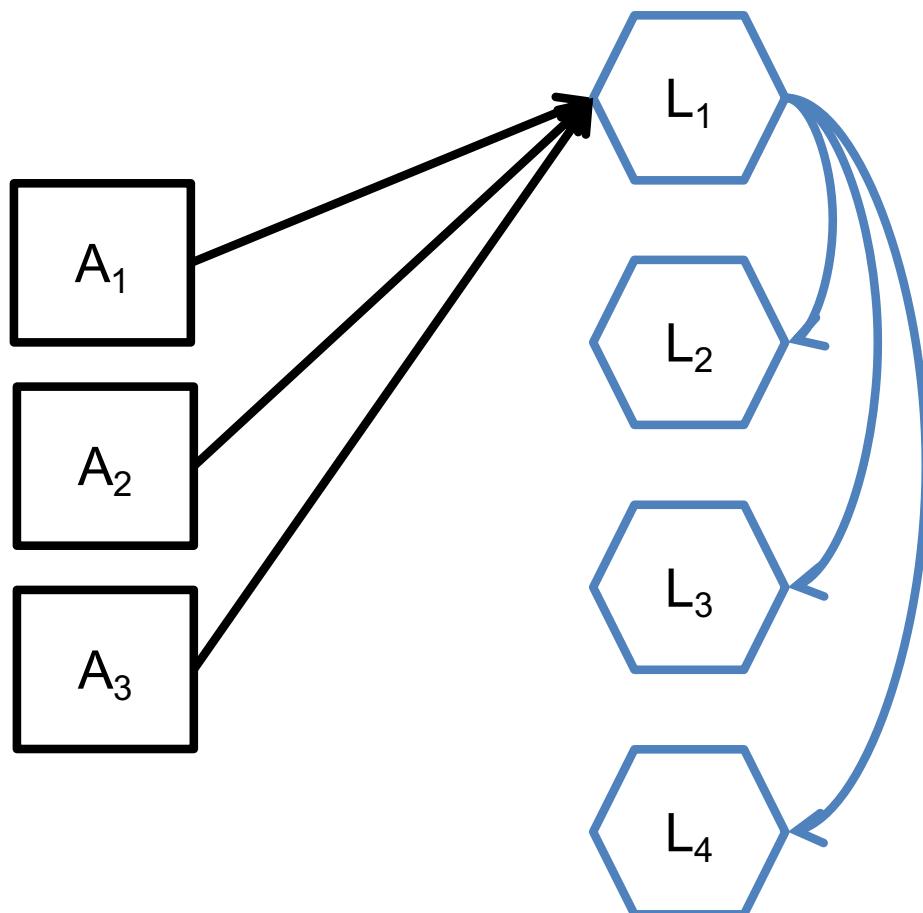
1. Ogni *acceptor* informa tutti i *learner* circa l'accettazione di un valore.



- $\# \text{ msg} = N_A * N_L$.
- Tollera il crash di $N_L - 1$ learner.

::: Esempio (2/3)

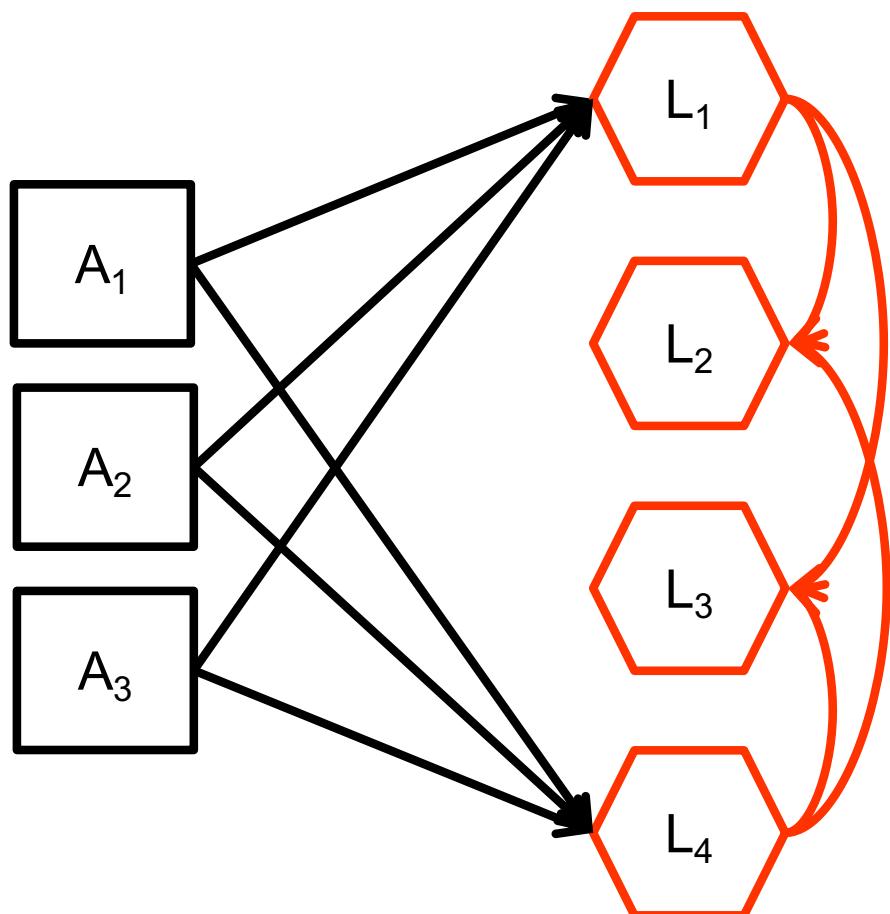
2. Gli *acceptor* informano un solo *learner distinto*, che poi informa tutti gli altri *learner*.



- $\# \text{ msg} = N_A + N_L - 1$.
- In caso di crash del *learner distinto* il valore scelto viene perso.

::: Esempio (3/3)

3. Gli *acceptor* informano un sottoinsieme di *learner* che poi provvedono ad informare gli altri *learner*.



- $\# \text{ msg} = N_A * N'_L + N'_L * (N_L - N'_L)$
- Tollera il crash di $N'_L - 1$ learner.

::: Progresso (1/2)

- Il progresso non è garantito:
 - Il *proposer* p completa la fase 1 per una proposta con numero n_1 .
 - Un altro *proposer* q completa la fase 1 con una proposta con numero $n_2 > n_1$.
 - Durante la fase 2, le *accept* di p sono ignorate poiché gli *acceptor* hanno promesso di non accettare proposte con numero minore di n_2 .
 - Quindi p ricomincia la fase 1 con una proposta con numero $n_3 > n_2$.
 - Le *accept* inviate da q con numero n_2 saranno ignorate (gli *acceptor* nel frattempo hanno fatto promessa a p di non accettare proposte con numero minore di n_3).
 - q fa una proposta con numero $n_4 > n_3 \dots$

::: Progresso (2/2)

- Per garantire la *liveness* si può eleggere un *proposer distinto*.
 - Esso deve essere l'unico processo autorizzato ad effettuare proposte (e quindi ad inviare i messaggi *prepare*).
- Se non si verifica un numero eccessivo di guasti nelle restanti componenti del sistema (*proposers*, *acceptors* e rete di comunicazione), è possibile raggiungere il consenso con un singolo *proposer*.
- Del resto sappiamo che

Non esiste alcun algoritmo deterministico in grado di garantire il raggiungimento del consenso in un sistema asincrono a scambio di messaggi anche nel caso di un unico fallimento per crash di un processo.
- Per cui occorre un algoritmo per l'elezione del *proposer distinto*, il quale richiede l'introduzione di *casualità* o *timeouts*.
 - **La Liveness è pertanto garantita esclusivamente durante i periodi di sincronia del sistema.**
 - **La Safety invece è garantita a prescindere dalla sincronia.**

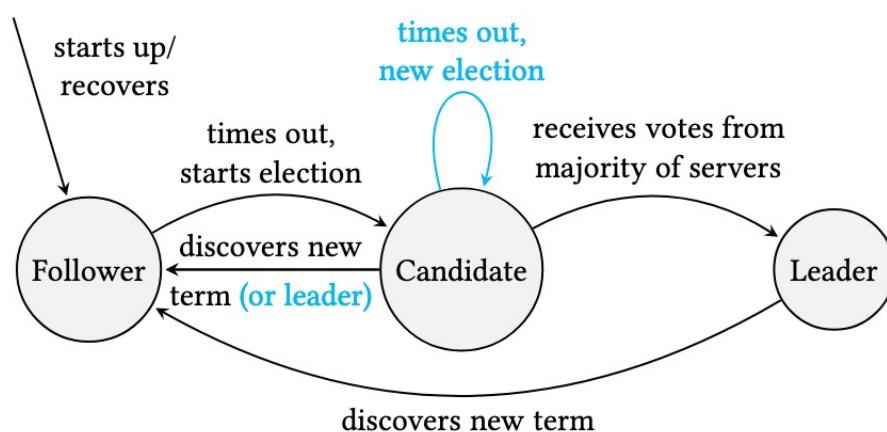
::: Consenso basato sul Leader

Molti algoritmi di consenso utilizzano un approccio basato sul leader per risolvere il consenso distribuito.

L'algoritmo Paxos di base richiede almeno due RTT per raggiungere una risoluzione. In caso di concorrenza, l'algoritmo potrebbe averne bisogno di più. In casi estremi, potrebbe persino produrre un livelock, il che è inefficiente.

::: Consenso basato sul Leader

Molti algoritmi di consenso utilizzano un approccio basato sul leader per risolvere il consenso distribuito.



Uno degli n nodi viene designato come leader. Tutte le operazioni vengono inviate al leader, che aggiunge l'operazione al proprio registro e avvia il consenso. Una volta che il leader ha ricevuto conferme dalla maggior parte dei nodi del raggiungimento del consenso, modifica il proprio stato.

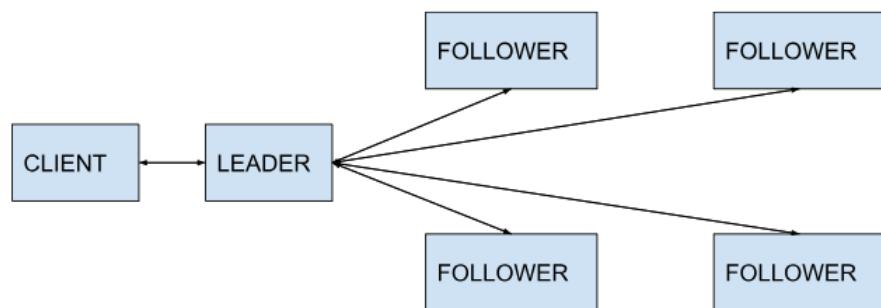
Questo processo si ripete finché il leader non fallisce, e un altro nodo assume il ruolo di leader.

State Machine Safety: Se un server ha applicato una voce di registro a un dato indice della sua macchina a stati, nessun altro server applicherà mai una voce di registro diversa per lo stesso indice.

Leader Completeness: Se un'operazione op viene eseguita all'indice i da un leader nel termine t , allora tutti i leader dei termini $> t$ avranno anche l'operazione op all'indice i .

::: RAFT (1/2)

Raft è progettato per essere più comprensibile di Paxos e funziona sulla base di un modello di leadership forte.



Elezione Leader: i nodi usano timeout casuali per eleggere un leader: ogni nodo ha un timeout e, quando scade, diventa un candidato e avvia un processo di elezione del leader.

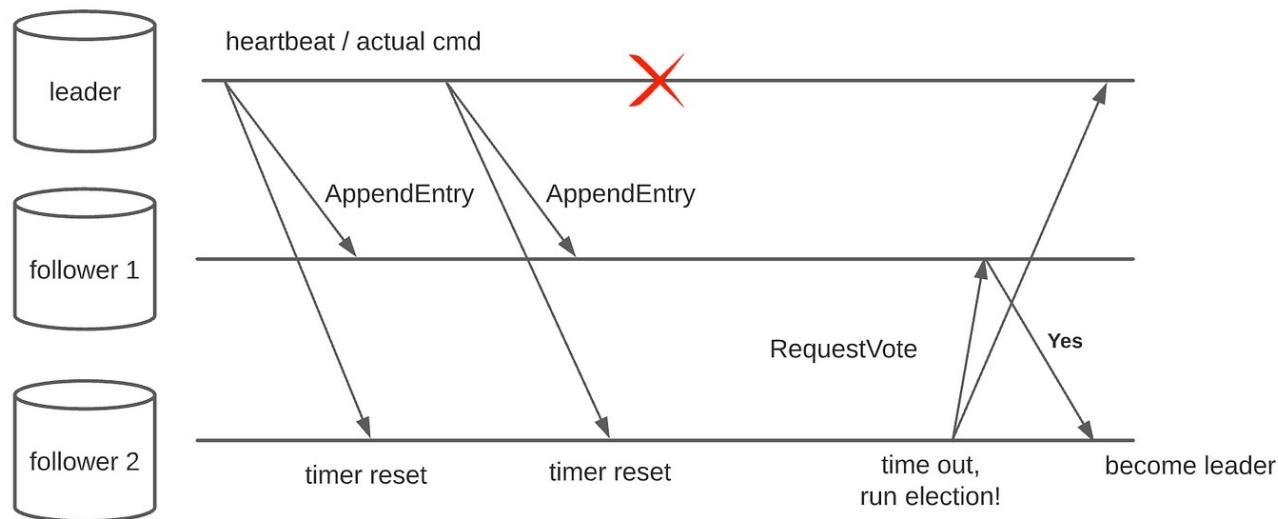
Il candidato invia una "richiesta di voti" ad altri nodi. Se un nodo non ha votato per nessun altro candidato nel mandato corrente e non è ancora diventato un leader, vota per il candidato. Se un candidato riceve voti da un quorum di nodi (maggioranza), diventa il leader.

Log Replication: Il leader riceve le richieste dei client e le aggiunge al suo log, e invia le entry di log ai follower (altri nodi) per replicare i dati. Una volta che una entry di log è replicata in un quorum di nodi, il leader esegue il commit della voce e notifica ai follower di applicare la voce di log.

::: RAFT (2/2)

Safety: Raft garantisce la sicurezza avendo sempre il log più aggiornato sul leader. Se un follower scopre un log più recente da un altro nodo, torna allo stato di follower e segue il nuovo leader.

Raft dà priorità alla comprensibilità rispetto alla complessità, rendendolo più facile da implementare e ragionare. Mantiene i concetti fondamentali di Paxos, fornendo al contempo un approccio più strutturato e intuitivo al consenso distribuito.



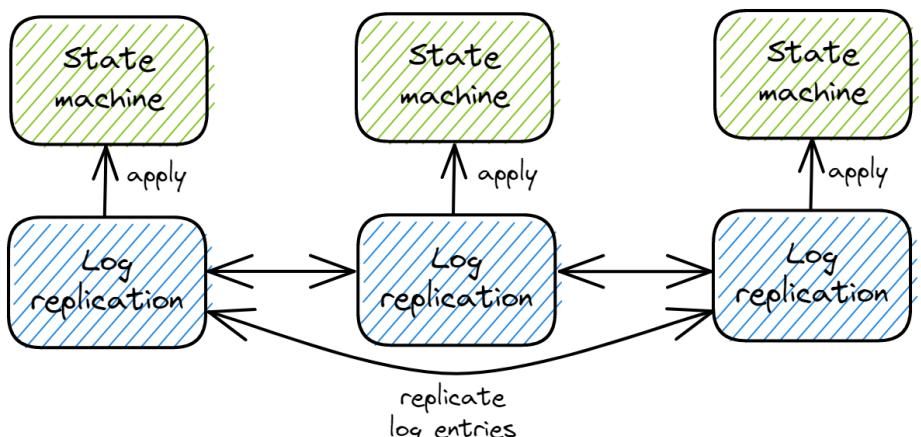
Un sistema di timeout e heartbeat viene impiegato per monitorare lo stato di salute di un leader.

::: RAFT (2/2)

Safety: Raft garantisce la sicurezza avendo sempre il log più aggiornato sul leader. Se un follower scopre un log più recente da un altro nodo, torna allo stato di follower e segue il nuovo leader.

Raft dà priorità alla comprensibilità rispetto alla complessità, rendendolo più facile da implementare e ragionare. Mantiene i concetti fondamentali di Paxos, fornendo al contempo un approccio più strutturato e intuitivo al consenso distribuito.

RAFT è una forma di State Machine Replication: il servizio è modellato come una macchina a stati, che esegue una sequenza di comandi per modificare il suo stato/eseguire azioni e la sequenza di comandi è il registro replicato, che viene replicata su diversi nodi in un sistema distribuito. Ogni replica mantiene lo stato del servizio e implementa le operazioni del servizio.

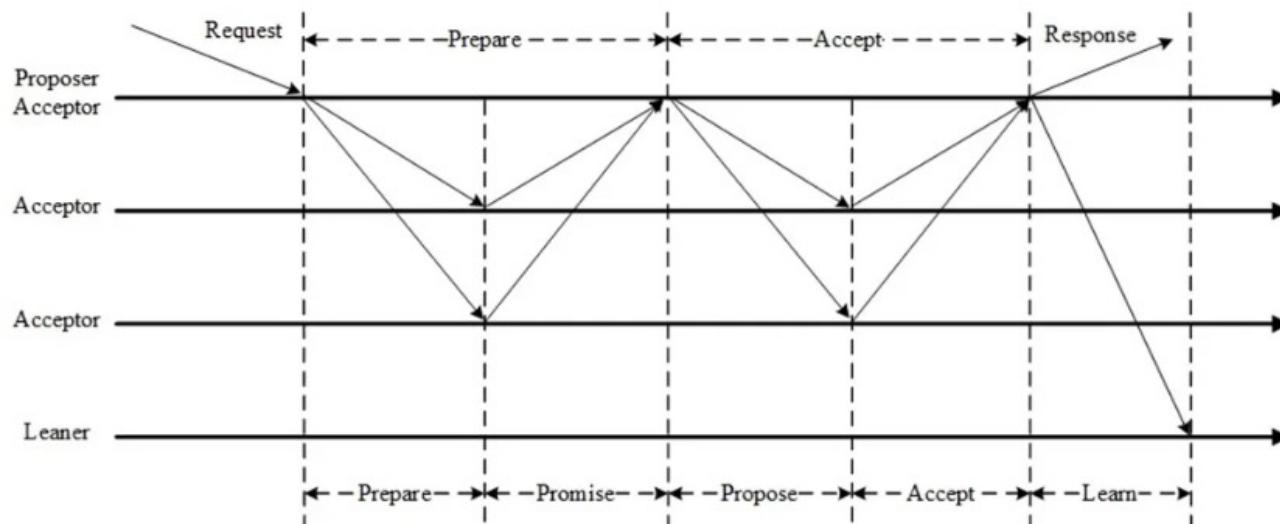


::: Robustezza nel Consenso

- Dal punto di vista della robustezza, la forza di un protocollo di consenso è solitamente misurata dal numero di componenti guasti che può tollerare:
 - Se un protocollo può tollerare almeno un guasto di tipo crash, lo si definisce crash-fault tolerant (CFT);
 - Se un protocollo può tollerare almeno un guasto di tipo bizantino, lo si definisce byzantine-fault tolerant (BFT);
- Un protocollo BFT è naturalmente CFT, ma non il contrario.
- L'algoritmo di Paxos è CFT, ma BFT solo per i guasti bizantini che non si verificano nel processo proposer.
- Neanche RAFT è BFT:
 - Se un nodo vota due volte in un dato termine, o per un altro nodo che ha un log non aggiornato come il suo e quel nodo diventa leader. Tale comportamento potrebbe causare il caso in cui due nodi credono di essere leader o incongruenze nel log.
 - L'invio di messaggi heartbeat falsi ma validi possono causare inconsistenze.

::: Multi-Paxos (1/2)

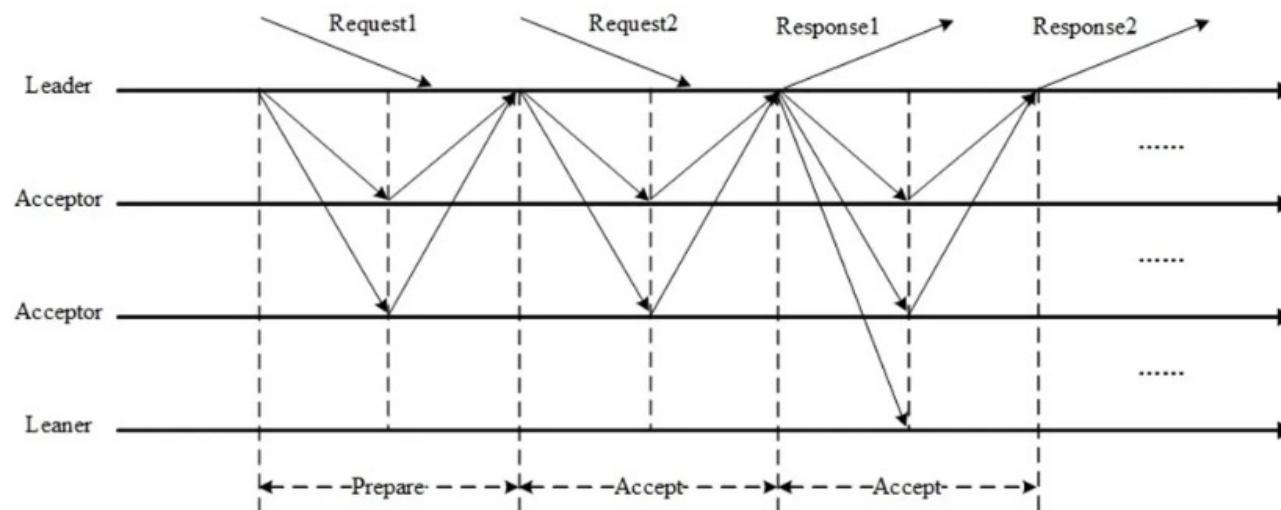
- Molte proposte di protocolli BFT sono troppo inefficienti per essere utilizzati nella pratica, o presuppongono la sincronia, cioè si basano su limiti noti sui ritardi dei messaggi e sulle velocità dei processi.
- Paxos richiede vari round di scambio messaggi per giungere a convergenza.



::: Multi-Paxos (2/2)

Multi-Paxos elegge un leader e la proposta viene avviata dal leader. Grazie all'assenza di competizioni, il problema del livelock viene eliminato.

Se tutte le proposte vengono avviate dal leader, la fase di preparazione può essere saltata per modificare il processo in due fasi in un processo in una fase, il che migliora l'efficienza.



Multi-Paxos non presuppone un leader unico, ma consente a più leader di proporre richieste contemporaneamente. Multi-Paxos può essere ottimizzato per saltare la fase di preparazione ed entrare direttamente nella fase di accettazione quando lo stesso proponente fa proposte continue.

::: Consenso Sicuro (1/3)

La maggior parte dei lavori esistenti riguardanti il consenso sono stabiliti sotto il presupposto che la rete sia sicura (e tutti i nodi sono sicuri).

Potrebbe verificarsi un attacco di manipolazione dei messaggi nella rete, dove il nodo attaccante potrebbe essere un nuovo partecipante alla rete o un nodo compromesso che era precedentemente un nodo sicuro.

Lo stato del nodo di attacco è selezionato arbitrariamente, e possono esserci principalmente due tipi di attacchi di manipolazione dei messaggi, ovvero attacco a iniezione costante e attacco a iniezione casuale.

Senza alcun meccanismo di difesa, i nodi non possono distinguere lo stato del nodo sicuro e quello dell'attacco in modo che può utilizzare le informazioni del nodo di attacco per aggiornare i propri stati.

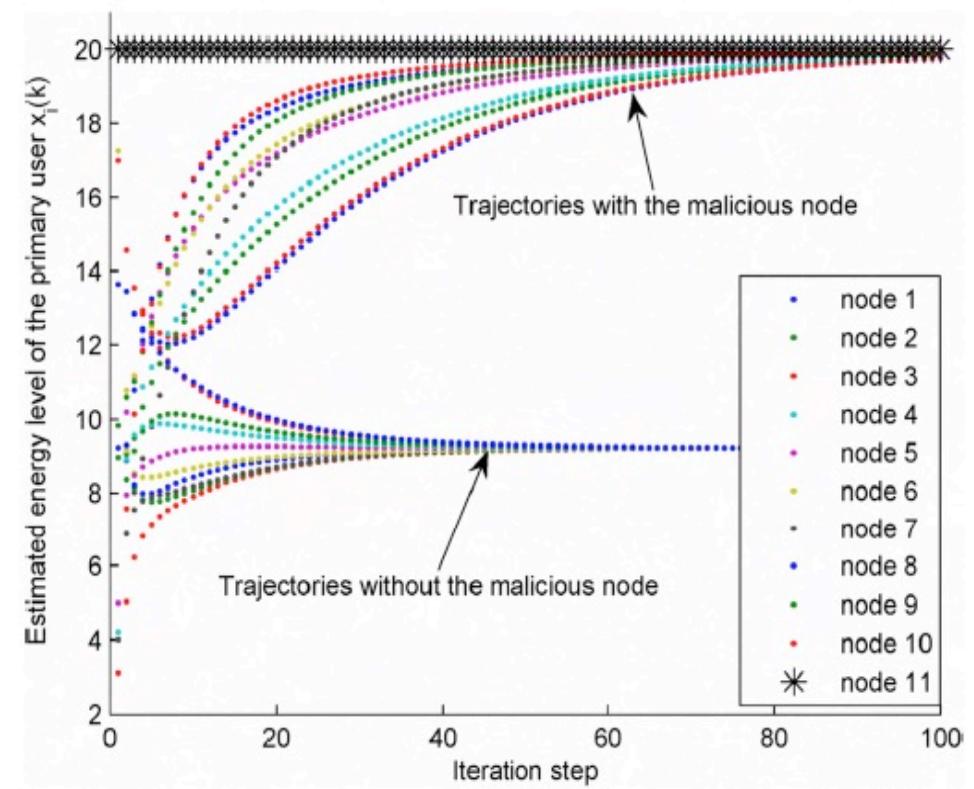
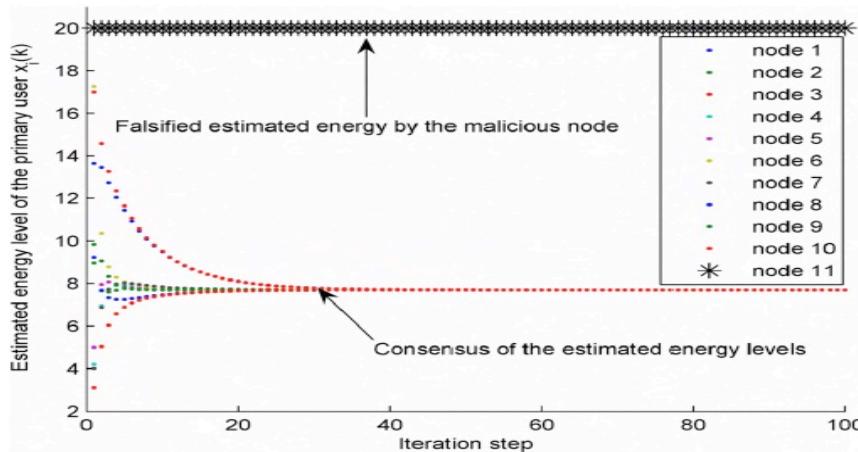
Attaccanti interni che dispongono di tutti i dati di supporto al consenso, attaccanti esterni che possono riprodurre messaggi autentici, camuffare altri nodi con le loro identità catturate. Gli esterni possono essere facilmente rimossi dall'autenticazione.

::: Consenso Sicuro (2/3)

L'effetto degli attacchi sugli algoritmi di consenso:

- Può far divergere la rete.
- Può invertire lo stato corretto del sistema / fenomeni osservati.
- Può far convergere la rete al suo valore iniettato.
- Può impedire alla rete di raggiungere un consenso.

Una soluzione è calcolare la devianza di ogni nodo dal valore di consenso della maggioranza, ed definire affidabili i nodi con una minima deviazione.



::: Consenso Sicuro (3/3)

PROBLEMA: Rimuoverà un utente anche se non ci sono aggressori. Quando gli aggressori colludono, i metodi di rilevamento dei valori anomali diventano meno efficaci.

Un miglioramento consiste nell'aggiungere la tecnica di autenticazione utilizzando la crittografia basata sull'identità:

- Supponiamo che un nodo voglia inviare un valore agli altri durante un algoritmo di consenso, esso firma il messaggio con la sua chiave privata e lo crittografa utilizzando l'identificativo di uno dei possibili destinatari, e quindi lo invia.
- Il ricevente decifra il messaggio ricevuto utilizzando la sua chiave privata e quindi utilizzando la chiave pubblica del mittente ne verifica la firma, di conseguenza l'autenticità.
- Se la verifica ha esito positivo, il messaggio può entrare nella procedura di aggiornamento del consenso.