

PER ALTRI APPUNTI CONSULTARE IL SITO:
https://luigi-v.github.io/Appunti_Universita/

1. INTRODUZIONE AI SISTEMI DISTRIBUITI E PROBLEMI DI CONSENTO

Un **sistema distribuito** sono componenti hardware o software, situati in computer connessi in rete, che comunicano e coordinano le loro azioni solo passando messaggi, ad esempio Java RMI. I computer collegati da una rete possono essere separati spazialmente da qualsiasi distanza.

La definizione di sistemi distribuiti ha le seguenti conseguenze significative:

- **Concorrenza:** in una rete di computer, è possibile l'esecuzione simultanea del programma, condividendo risorse come pagine web o file quando necessario. La capacità del sistema di gestire le risorse condivise può essere aumentata aggiungendo più risorse (ad esempio computer) alla rete.
- **Assenza di un clock globale:** quando i programmi devono cooperare coordinano le loro azioni scambiandosi messaggi. Lo stretto coordinamento spesso dipende da un'idea condivisa del momento in cui si verificano le azioni dei programmi. Ma si scopre che ci sono limiti all'accuratezza con cui i computer in una rete possono sincronizzare i loro orologi, non esiste un'unica nozione globale dell'ora corretta. Questa è una diretta conseguenza del fatto che l'unica comunicazione è l'invio di messaggi attraverso una rete, esempio problemi di temporizzazione (*drift dei clock*).
- **Guasti indipendenti:** i guasti nella rete comportano l'isolamento dei computer ad essa collegati, ma ciò non significa che smettano di funzionare. I programmi su di essi potrebbero non essere in grado di rilevare se la rete non è riuscita o è diventata lenta. Il guasto di un computer, o la chiusura imprevista di un programma da qualche parte nel sistema (crash), non viene immediatamente reso noto agli altri componenti con cui comunica. Ogni componente del sistema può guastarsi indipendentemente, lasciando gli altri ancora in funzione.

La motivazione principale per la costruzione e l'utilizzo di sistemi distribuiti deriva dal desiderio di condividere le risorse.

Un esempio di sistema distribuito è la rete internet, dove diversi processi connessi da reti di vario tipo si scambiano messaggi mediante protocolli di comunicazione. 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.

Nella progettazione dei sistemi distribuiti ci sono vari fattori di complessità e che impattano su di essi e creano problemi, e sono:

- **Eterogeneità**, varietà e differenza di reti (tecnologie e protocolli), sistemi operativi, hardware (server, client), linguaggi di programmazione, ecc.... Il termine **middleware** si applica a un livello software che fornisce un'astrazione di programmazione e maschera l'eterogeneità;
- **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 (es. SOA), standard condivisi e adeguati test di conformità;
- **Sicurezza**, un sistema distribuito essendo aperto è più facile da attaccare, dove è possibile estrarre informazioni utili. Quindi bisogna garantire confidenzialità (protezione dall'accesso da parte di utenti non autorizzati), integrità (protezione dall'alterazione) e disponibilità (protezione dall'interferenza coi mezzi di accesso alle risorse);
- **Concorrenza**, l'accesso a risorse e servizi condivisi deve essere consentita in maniera virtualmente simultanea a più utenti;
- **Trasparenza**, un sistema distribuito deve nascondere che i suoi processi e risorse sono fisicamente distribuite:
 - **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 duplicazione:** nascondere la duplicazione 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).
- **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 (smart city);
 - **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.

I sistemi distribuiti adottano un cosiddetto **modello** che 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 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.

Vanno anche specificate 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.

Si può andare a descrivere l'**organizzazione** del sistema distribuito nelle sue parti componenti, le relazioni tra esse, e la loro allocazione sui nodi del sistema di elaborazione, ad esempio Client-Server a 2 livelli, peer-to-peer, SOA, ecc....

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 quello di rendere esplicite le ipotesi rilevanti sul sistema e consentire di studiare comportamenti/proprietà possibili o non possibili del sistema, date le ipotesi. Esempi di modelli fondamentali sono modello di interazione (UML), modello dei guasti o modello di sicurezza.

Generalmente, un sistema distribuito è composto da **processi** (entità che eseguono azioni elaborate, in esecuzione su più nodi del sistema di elaborazione, ciascuno dotato di un proprio orologio) che incapsulano 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.

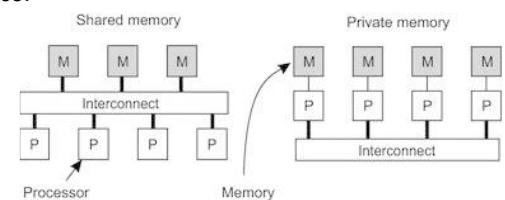
Non esiste un *clock* globale. La velocità dei processi e i tempi di trasmissione dei messaggi tipicamente non possono essere previsti con esattezza.

Esistono modelli di interazione sistema sincrono, sistema asincrono o sistema parzialmente sincrono.

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.



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:

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.

Vantaggi:

Per i sistemi sincroni 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.

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 e 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 sincroni e asincroni sono gli **estremi** di uno spettro di possibilità con cui modellare i sistemi reali.

MODELLI DI FALLIMENTO:

Il **modello di fallimento** definisce i modi in cui può verificarsi un guasto al fine di fornire una comprensione dei suoi effetti.

- **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*).

Si parla di "catena fault – error – failure".

Errori e fallimenti si propagano da un nodo all'altro in un ambiente distribuito.



Un modello dei fallimenti per un sistema distribuito definisce le modalità con cui si possono verificare guasti in **processi** e **canali di comunicazione**. In pratica, si possono avere fallimenti per *crash*, omissione (messaggio perso in rete), di valore, bizantini, e (per i sistemi sincroni) temporali.

Esiste una **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).

Si possono avere **fallimenti benigni**, generalmente non manipolati da qualcuno, e sono le *omission failures* e *timing failures*.

Esistono dei modelli che codificano il tutto e sono le **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*).
- **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:

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

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

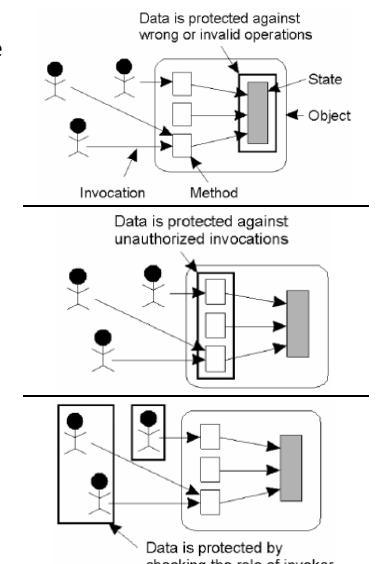
- **Intercettazione** dei messaggi (accessi non autorizzati);
- **Interruzione** (diniego di servizio - denial of service);
- **Alterazioni** (modifiche di dati non autorizzate);
- **Fabbricazione** (inserimenti di dati non autorizzati).

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.

Esistono tre approcci per la **protezione** da minacce alla sicurezza:

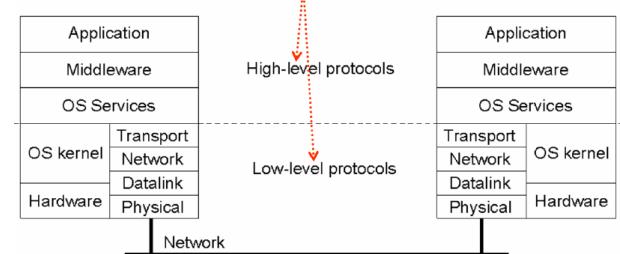
- **Protezione da operazioni non valide**, si va ad inserire un meccanismo di protezione per evitare che operazioni invalide o sbagliate non vengano realizzate, quindi si va a definire un perimetro di sicurezza e solo quello che è valido o autorizzato può entrare in quel perimetro. Si va a proteggere il dato da operazioni non valide.
Esempio, se un dato può assumere solo valori positivi, un utente che modifica in una valore negativo viene bloccato;



- **Protezione da invocazioni non autorizzate**, l'interfaccia del processo verso l'esterno viene protetto, ma non il dato, se viene protetto cosa viene da fuori, di conseguenza, viene protetto ciò che c'è dentro. Esempio, se un dato può essere solo letto, un utente che vuole scrivere viene bloccato;
- **Protezione da utenti non autorizzati**, si effettua una caratterizzazione degli utenti che possono o meno interagire col sistema. Si dà accesso solo ad utenti autorizzati.

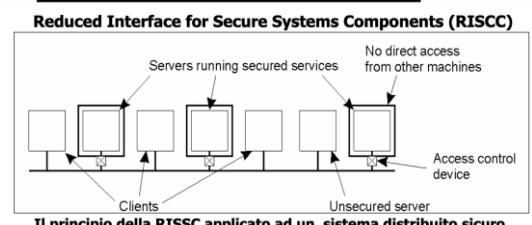
Bisogna decidere anche a quale livello mettere i meccanismi di sicurezza, basso livello o alto livello.

Nei sistemi distribuiti sono generalmente posti a livello del middleware, e bisogna scegliere i meccanismi di sicurezza nel loro complesso.



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, riducendo la TCB ad un piccolo sotto-insieme di nodi.



Il principio della RISCC applicato ad un sistema distribuito sicuro

1.1 CONSENSO NEI SISTEMI DISTRIBUITI

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 tale valore. Meno informalmente:

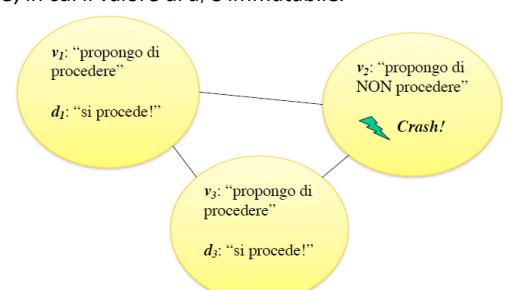
- N processi comunicanti tramite scambio messaggi;
- i canali di comunicazione sono affidabili;
- i processi sono soggetti a fallimenti di tipo *crash* oppure bizantini (valori aleatori);
- il processo p_i ($i=1 \dots N$) possiede una propria **variabile di decisione** d_i , e propone un proprio valore 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.

Esempio:

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.



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)**: 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 esempio, 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.

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 ed escluderlo dalla rete, perché devono prima raggiungere un consenso su quale componente ha fallito per primo.

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 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.

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 esempio, 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.

PROBLEMA DELLA CONSISTENZA INTERATTIVA:

È una variante del problema del consenso, dove 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.

RELAZIONE TRA I PROBLEMI C, BG, IC:

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

- $C_i(v_1, v_2, \dots, v_n) \rightarrow$ Decisione di p_i in un'esecuzione di un algoritmo di **Consenso**;
- $BG(j, v) \rightarrow$ Decisione di p_j in un'esecuzione di un algoritmo dei **Gen. Bizantini**, ove p_j è il comandante;
- $IC_i(v_1, v_2, \dots, v_n)[j] \rightarrow$ j-esimo valore del vettore di decisione di p_i in una esecuzione di un algoritmo di **Interactive Consist.**

- BG \rightarrow IC:

n esecuzioni di BG: $\forall i \in \{1, \dots, n\}$ p_i è il generale. Si ha: $IC_i(v_1, v_2, \dots, v_n)[j] = BG(j, v_j)$ ($i, j = 1, 2, \dots, n$);

- IC \rightarrow C:

Applicando una funzione di valutazione sul vettore di soluzioni prodotto da IC si ha: $C_i(v_1, v_2, \dots, v_n) = majority(|IC_i(v_1, v_2, \dots, v_n)[j]|)$ ($j = 1, 2, \dots, n$);

- C \rightarrow 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) $BG_i(j, v) = C_i(v_1, v_2, \dots, v_n)$ ($i = 1, 2, \dots, n$)

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

CONSENSO IN UN SISTEMA SINCRONO:

Nei sistemi sincroni, l'algoritmo di consenso è basato sul **multicast**, cioè la possibilità di trasmettere la stessa informazione a più destinatari. 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.

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 , ogni processo p_i ($i=1 \dots N$):

- esegue il *B-multicast*, invia i valori che non ha inviato nei cicli precedenti;
- esegue il *B-deliver*, riceve 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 *timeout* (sistema sincrono), visto che alcuni processi possono andare in crash, altrimenti ci sarà un tempo di attesa della risposta indefinito, ma anche per poter rilevare i crash.

Algoritmo di Dolev et al:

Algoritmo eseguito dal generico processo p_i

Inizializzazione:

$$V_i^0 = \emptyset; \\ V_i^1 = \{v\};$$

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:

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

1. Inizialmente il vettore è inizializzato vuoto;
2. Nel primo ciclo, si aggiunge al vettore il valore che il processo i -esimo propone, si vanno ad effettuare delle iterazioni, si effettua il B-multicast non nel vettore formato, ma di quei valori che non sono stati visti (si inviano solo gli aggiornamenti del vettore e non tutto il vettore, per ragioni di prestazioni);
3. Nel nuovo ciclo, si prende il vettore del ciclo precedente (dato che il vettore cresce progressivamente) e poi si fa una iterazione di attesa dei messaggi provenienti dai vari processi, di cui si suppone che restituiscano qualcosa. Quando arriva tale messaggio, si aggiunge al vettore e si itera per k volte (necessari $f+1$ cicli);
4. Alla fine, la decisione è data dal valore minimo presente tra le componenti del vettore.

Ovviamente, quando si vanno a specificare degli algoritmi distribuiti si vanno a studiarne delle proprietà, ovvero:

- **Proprietà di terminazione**, ovvia (il sistema è sincrono), i cicli sono in numero finito ($f+1$) e il ciclo in sé ha una durata finita perché ha un timeout;
- **Proprietà di 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).

Dimostrazione (per assurdo):

Assumiamo che due processi non abbiano lo stesso vettore finale, esempio 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.

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:

Per la risoluzione del problema dei generali bizantini è la seguente:

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

Ad esempio, se il comandante dice v su entrambi i canali 1 e 2, e p_2 dice che il comandante ha detto v e p_3 dice che ha detto u , i valori sono discordi.

Nel secondo caso, se il generale è bizantino e manda sul canale 1 il valore v e sul canale 2 in valore u , se si riescono a confrontare i due valori e sono differenti.

Il processo p_2 è l'unico processo corretto in entrambe le situazioni, non riesce a capire la decisione da prendere, in più, non riesce ad individuare il processo bizantino.



I processi in **rosso** sono bizantini, Com è il Comandante, p_2, p_3, p_4 i luogotenenti. Notazione: $i:j:v = i$ dice che j dice v .

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 $N \leq 3f$.

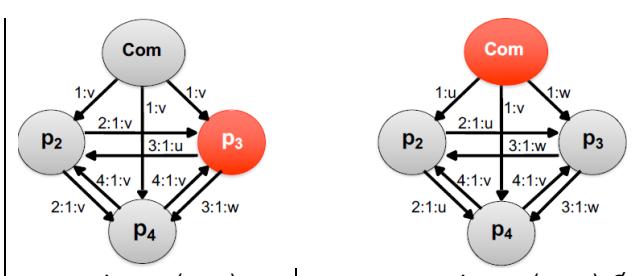
PROBLEMA CON QUATTRO GENERALI:

Se si andasse ad incrementare il numero di nodi, in questo caso p_2 potrebbe prendere una decisione, perché:

Se si vede il primo caso dove p_3 è malizioso, p_2 riceve il messaggio v da Com, il messaggio u da p_3 e v da p_4 . Quindi ha due messaggi v e una u , può decidere a maggioranza e capire che p_3 è bizantino.

Se si vede il secondo caso, p_2 riceve il messaggio u da Com, il messaggio w da p_3 e da p_4 il messaggio v , ovvero tre valori discordi. Qui non prende una decisione, però può determinare che il comandante è bizantino, dati tutti i valori differenti.

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



$p_2: \text{maggioranza}(v, u, v) = v$
 $p_4: \text{maggioranza}(v, v, w) = v$

$p_1, p_2, p_3: \text{maggioranza}(u, v, w) = \emptyset$
 $p_4: \text{maggioranza}(v, v, w) = v$

CONSENSO NEI SISTEMI ASINCRONI:

Nei sistemi asincroni esiste un principio chiamato FLP (Fischer, Lynch, Paterson):

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.

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

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

Quando siamo in un sistema distribuito (esempio blockchain) il consenso è molto importante, se non c'è una soluzione garantita, per il raggiungimento di una soluzione è quella di:

- **Indebolire la condizione di Termination**, introducendo elementi di non determinismo o garantendo la *termination* esclusivamente durante periodi di sincronia del sistema, in generale, si cerca di far avanzare l'algoritmo più in avanti possibile, ritardando la terminazione;
- **Indebolire la condizione di Agreement**, individuando un insieme finito per i possibili valori decisionali dei singoli processi, in modo che questi siano concordi non su un solo valore ma su un insieme di valori;
- **Irrobustire il modello del sistema**, introducendo dei *failure detectors* per distinguere i processi lenti (ma corretti) da quelli effettivamente falliti.

Modo di Fallimento	Sistema Sincrono	Sistema Asincrono
Nessuno	Consenso ottenibile Conoscenza comune anche ottenibile	Consenso ottenibile Conoscenza comune 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

PARLAMENTO PART-TIME (ISOLA DI PAXOS):

I decreti approvati erano numerati (in ordine crescente) e i parlamentari potevano parlare solo scambiandosi messaggi, veicolati 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. Il compito principale del Parlamento era quello di determinare la legge del paese, definita da 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.

I legislatori potevano dimenticare ciò che avevano fatto se lasciavano l'aula parlamentare, e quindi scrivevano note su dei registro per ricordare a sé stessi importanti compiti parlamentari, come una sequenza numerata dei decreti approvati. 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. Tuttavia, un altro legislatore potrebbe non avere una voce nel suo registro per un preciso decreto approvato in precedenza, se non lo aveva ancora saputo. 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 approvati in un tempo accettabile 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:

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”.

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. Quindi venivano fornite delle clessidre.

Analogie col problema del consenso:

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

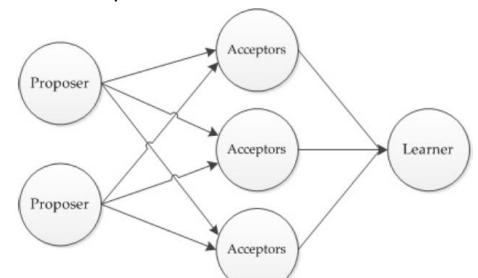
Le **proprietà del consenso** di Paxos sono:

- **Liveness**, uno tra i valori proposti prima o poi viene scelto. Se un valore viene scelto, ogni processo prima o poi apprenderà tale scelta. Esempio, uno dei decreti proposti, prima o poi, veniva approvato, rientrando poi all'interno dei registri;
- **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.

In realtà, queste proprietà sono quelle di terminazione, consenso e integrità. Queste due visioni sono del tutto equivalenti.

Nell'**algoritmo di consenso di Paxos**, 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 arbitraria, 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.

Esistono delle varianti dell'algoritmo di Paxos, in una di esse tutti sono *Proposer* ed esiste un solo *Accepter*, esso riceve le richieste e decide il valore. Questa soluzione è la più semplice ma meno efficace, perché il fallimento dell'*acceptor* (**Single point of failure**) renderebbe impossibile il progresso dell'algoritmo facendo fallire le tre proprietà.

Essendo unico e ricevendo tutto il traffico, la sua probabilità di fallimento è diverso dagli altri processi dato che può congestionarsi o esaurire le risorse.

La soluzione migliore è quella di avere un insieme di Acceptor, in tal caso, per raggiungere il consenso, un valore proposto è scelto se un insieme sufficientemente grande di essi lo accetta.

In assenza di fallimenti o perdite di messaggi, è desiderabile che un valore venga scelto anche se proposto da un solo *proposer*, e gli altri convergono. Quindi, il primo requisito che sia soddisfatto è:

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*.

Esempio, 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.

P1 è 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, così da identificare e quindi l'*acceptor* accetta la coppia identificativa - 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)

Possiamo ammettere che siano scelte proposte differenti, a condizione che contengano lo stesso valore. Quindi i *proposer* associano un identificativo che può essere diverso (dovuto ai clock diversi) tra loro però il valore uguale. 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.

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. Esempio, se vengono mandati messaggi in B-multicast (unicast verso a, ... unicast verso d), mentre c sta ricevendo e d non ha ancora avuto l'invio, a ha ricevuto e accettato, quindi è possibile una disparità temporale delle accettazioni (natura asincrona), ma è possibile anche che esista 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.

SODDISFIRE P2^b:

Si supponga che la proposta **(m,v)** sia stata scelta e accettata (da almeno un Acceptor), quindi si vuole che ogni proposta con numero seriale **n > m** abbia valore **v**.

È possibile procedere per induzione su n, e 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 assume 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*.

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:

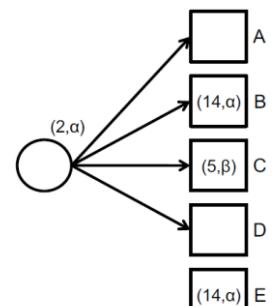
(a) nessun acceptor di S ha accettato una proposta con numero seriale minore di n (v può essere qualsiasi), oppure

(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.

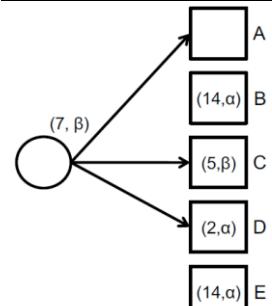
Si può soddisfare P2^b mantenendo l'invariante P2^c.

Esempio:

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



In questo caso il *proposer* deve proporre **β** (un'altra proposta), 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:

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 semplice (consultare lo stato degli *acceptor*), prevedere le scelte future degli *acceptor* è 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 . Risolvendo il problema dei due *proposer* che in maniera concorrente cercavano le accettazioni degli *acceptor*, così vengono troncate.

Questo è possibile, per la generazione delle proposte, grazie ad un messaggio 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.

Alla richiesta di *Prepare* segue sempre la 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 .

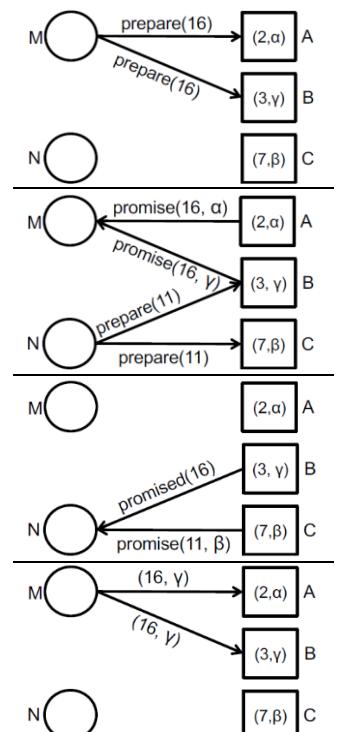
Il valore v associato alla proposta può essere:

- 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:

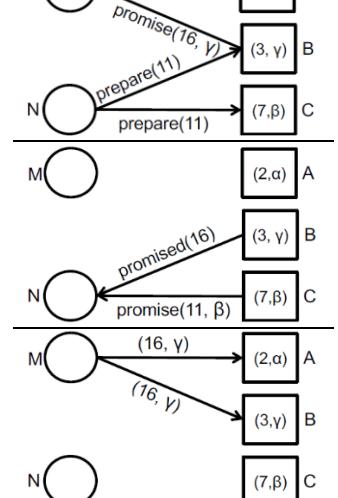
Assumiamo tre *acceptor* (A - B - C) con $n = 2, 3, 7$ e $v = \alpha, \gamma, \beta$ rispettivamente, e due *proposer*.

Il primo *proposer* M manda un messaggio di *prepare(16)* ad A e B, ma non è riuscito a mandarlo fisicamente a C.



Nel frattempo, il secondo *proposer* N ha mandato un *prepare(11)* a B e C. Dal momento che A e B avevano ricevuto il *prepare*, rispondono con un *promise*. A dice 16 e valore accettato α , mentre B dice 16 e valore accettato γ .

Quando la *prepare 11* arriva a B, esso non comunica γ , mentre C comunica β , però la *promise* che è andata in conflitto con l'altra, B ha risposto senza γ , questo significa che è già occupato



M adesso manda 16 (dato che è maggiore di 11) e tra α e γ , lui manda γ che sarà poi quello accettato e lo manda solo a chi ha risposto.

COMPORTAMENTO DEGLI ACCEPTOR:

Gli *acceptor* possono ricevere due tipi di messaggi: *prepare* e *accept*.

Un *acceptor* può sempre rispondere ad una richiesta di tipo *prepare*, ma può rispondere solo 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 .**

In realtà, 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 tenuto a mantenere in memoria stabile (così da soddisfare l'invariante P2^c anche in seguito al riavvio dopo un fallimento):

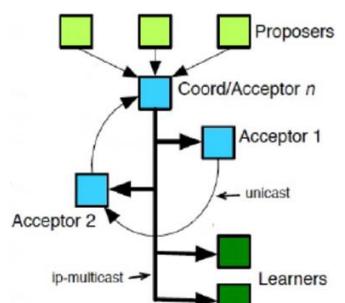
- La proposta di numero seriale più elevato accettata;
- Il numero seriale delle richiesta *prepare* con numero maggiore a cui ha risposto con un messaggio *promise*.

LEADER TRA GLI ACCEPTOR:

Nella formulazione classica gli *acceptor* sono replicati attivamente. Ma è anche possibile una formulazione passiva, perché avere *acceptor* attivi contemporaneamente può essere complesso e porta a problematiche, in realtà, i *proposer* non dialogano con tutti gli *acceptor* ma con un *coordinator* facendo sì che la replicazione sia passiva (solo uno che risponde ai *prepare*).

Quindi, un *coordinator* riceve le richieste dai *proposer*, e rappresenta il frontend del sistema di processi alla base di Paxos, e smista le richieste ai vari *acceptors* e *learners*.

Il *coordinator* propaga anche il valore per cui si è raggiunto il consenso.



ALGORITMO:

Fase 1:

- Un *Proposer* seleziona un numero di proposta n ed invia una richiesta $prepare(n)$ ad una maggioranza di *acceptors*;
- Quando un *acceptor* riceve una $prepare(n)$, se n è maggiore di ogni altra $prepare$ a cui ha già risposto, 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.

Fase 2:

- 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;
- 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 .

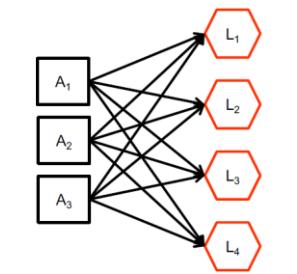
Fase 3:

- 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.

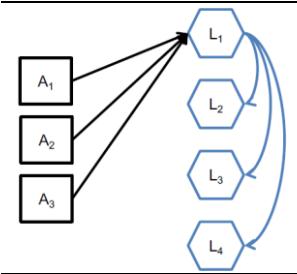
APPREDIMENTO (PROCESSI LEARNER):

È possibile dimostrare che la fase 2 (quella di commit) ha minor costo possibile per un algoritmo di consenso in presenza di fallimenti. In altri termini, Paxos può essere considerato l'ottimo. Fondamentalmente, per garantire la consistenza dello stato degli *acceptors* e disaccoppiare dal *proposer* la fase decisionale sul consenso (occupato magari ad effettuare altri processi), entrano in gioco i processi *learner*. Possono essere di 3 tipi:

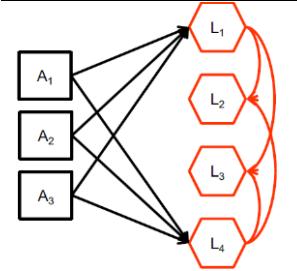
- Ogni *acceptor* informa tutti i *learner* circa l'accettazione di un valore. Questa soluzione riesce a tollerare i guasti, ma è onerosa.



- Gli *acceptor* informano un solo *learner distinto*, che poi informa tutti gli altri *learner*. Qui il problema è che se fallisce il *learner* attivo, prima di aver comunicato con quelli passivi, si perde la decisione. Soluzione efficiente ma più vulnerabile.



- Gli *acceptor* informano un sottoinsieme di *learner* che poi provvedono ad informare gli altri *learner*. Soluzione intermedia dei precedenti, riesce a tollerare i guasti.



Il sistema è asincrono, è un meccanismo di consenso, però 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 $n_4 > n_3$...

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 SICURO:

Nel consenso ci sono problematiche di sicurezza, potrebbero esserci attacchi volti a non far raggiungere il consenso, duplicando messaggi o iniettando messaggi spuri, oppure facendo accettare un valore non proposto. Senza alcun meccanismo di difesa, non è possibile distinguere quando un messaggio arriva da un nodo originale o da un attaccante.

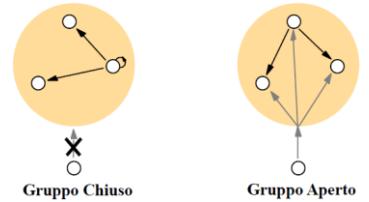
La soluzione per un consenso è usare la crittografia per dare un modo per non alterare i messaggi e anche per capire chi sta mandando un messaggio. Quindi, un modo per aiutare il consenso è cifrare i messaggi, evitando attacchi man in the middle, intercettando quindi la comunicazione.

2. COMUNICAZIONE DI GRUPPO E CONSISTENZA

Una **comunicazione di gruppo** consiste l'identificare un gruppo di nodi (ogni nodo ha un ID) e mandare un messaggi a tutti coloro che ne fanno parte.

Nel caso del consenso, si parla di primitive di gruppo, dove i nodi definiscono o meno la loro appartenenza ad un gruppo che può essere:

- **Gruppo chiuso**: solo i membri possono inviare messaggi in multicast al gruppo, cioè il messaggio può essere mandato dai membri del gruppo ad altri membri dello stesso gruppo;
- **Gruppo aperto**: anche i non membri possono inviare messaggi in multicast al gruppo, cioè il messaggio viene inviato da un membro che non fa parte di quel gruppo ai membri di un gruppo.

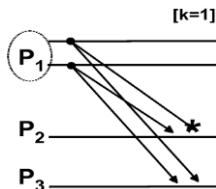


L'obiettivo della comunicazione di gruppo è quello di assicurare che i processi possano scambiare messaggi di gruppo anche in presenza di fallimenti dei canali di comunicazione (esempio problema configurazione o congestione dei router) e/o dei processi stessi, oppure assicurare che i messaggi inviati a un gruppo siano ricevuti in maniera ordinata (*consegna affidabile dei messaggi*).

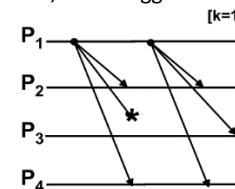
Le principali strategie per evitare questi fallimenti sono:

- **Error Masking**, l'affidabilità è ottenuta con tempi predibili, e si divide in:

- **Ridondanza spaziale**, i processi sono connessi con collegamenti ridondanti. Per mascherare k omissioni, occorrono $k+1$ collegamenti.



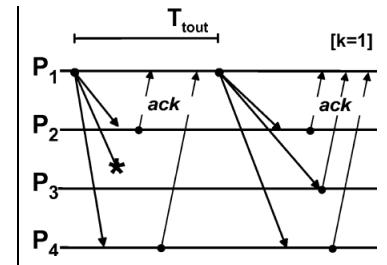
- **Ridondanza temporale**, il messaggio viene inviato più volte, anche se un processo lo ha ricevuto correttamente (link stress). Per mascherare k omissioni, il messaggio deve essere inviato $k+1$ volte.



In entrambi i casi si pone il problema dei messaggi duplicati, essi devono essere scartati dal ricevente. È difficile in generale trovare il giusto compromesso tra semplicità di *recovery* e consumo di banda.

- **Error Detection and Recovery**, basata su *acknowledgements* (acks) e *timeout*, ha un'affidabilità molto alta ma ha latenze alte non predibili. Gli ack possono essere:

- **Positive ack**, un messaggio viene ricevuto. Nello schema con positive ack, un messaggio viene ritrasmesso se non viene ricevuta una conferma al mittente entro un timeout predefinito. La failure viene individuata più rapidamente in caso di traffico sporadico.
- **Negative ack**, un timeout scade al ricevente, che decreta la perdita di un messaggio. Nello schema con negative acks, il ricevente richiede una ritrasmissione inviando un ack negativo, esso minimizza il traffico di rete ma richiede l'implementazione di uno dei seguenti meccanismi:
 - Numerazione dei messaggi, in maniera tale da poter richiedere la ritrasmissione di un particolare messaggio (se ad esempio viene ricevuto il messaggio i ma non il messaggio i-1);
 - Una strategia time-triggered in maniera tale che il ricevente sappia quando deve ricevere un messaggio.



RELIABLE MULTICAST:

In generale, nella comunicazione di gruppo esistono diverse primitive, come B-multicast e B-deliver. Esse garantiscono che un processo corretto prima o poi riceverà il messaggio, che viene inviato col B-multicast e ricevuto col B-deliver, a patto che il mittente non fallisca.

La primitiva di B-multicast può essere implementata utilizzando una primitiva per la comunicazione punto-punto su canale affidabile:

- B-multicast(g, m): for each process $p \in g$, send(p, m);
- B-receive(m) at p : B-deliver(m) at p .

La B-deliver soffre del problema dell'ack-implosion se si implementa il meccanismo di recovery con le ritrasmissioni, spreca larghezza di banda, e comunque il mittente può fallire in ogni istante durante l'esecuzione del B-multicast.

In realtà, si è abbandonato il meccanismo B-deliver basato sulle ritrasmissioni, facendo nascere il **Reliable Multicast** che soddisfa le proprietà:

- **Integrity**, un processo corretto p riceve m al più una volta. Inoltre, $p \in \text{group}(m)$ e m è stato inviato via multicast da sender(m);
- **Validity**, se un processo corretto p invia m in multicast, prima o poi riceverà m (riceve il messaggio anche sé stesso perché fa parte del gruppo);
- **Agreement**, se un processo corretto riceve m , allora tutti i processi corretti in $\text{group}(m)$ prima o poi riceveranno m .

Fino ad ora si è parlato di processi corretti, cioè se il processo è corretto allora riceve il messaggio, se invece si vogliono far garantire queste proprietà a processi corretti o meno, si parla di **Uniform Reliable Multicast** con la proprietà di agreement diversa:

- **Uniform agreement**, se un processo (corretto o meno) riceve m , allora tutti i processi corretti in $\text{group}(m)$ prima o poi riceveranno m .

Per progettare il R-Multicast ci si basa sul B-Multicast.

All'inizializzazione si imposta vuoto l'insieme dei messaggi ricevuti.

Il processo p , per effettuare il R-Multicast di m al gruppo g , invoca il B-Multicast. La differenza è sul B-Deliver, se m non è stato ricevuto (nuovo messaggio), lo si aggiunge all'insieme dei messaggi ricevuti, se poi il processo ricevente non è il mittente lo si ritrasmette eseguendo un'altra B-Multicast e lo si consegna all'applicazione

L'implementazione soddisfa le proprietà del Reliable Multicast:

- **Integrità**: discende dall'integrità dei canali di comunicazione;
- **Validità**: è evidente che un processo corretto prima o poi fa il B-deliver a sé stesso;
- **Accordo**: discende dal fatto che ogni processo corretto esegue B-multicast dopo B-deliver. Se dunque un processo corretto non effettua R-deliver, ciò può essere dovuto solo al fatto che non ha fatto B-deliver, ciò può sussistere solo se nessun altro processo corretto ha fatto il B-deliver e in tal caso nessuno farà R-deliver.

```

On initialization
Received := {};

For process p to R-multicast message m to group g
  B-multicast(g, m); // p ∈ g is included as a destination

On B-deliver(m) at process q with g = group(m)
  if(m ∉ Received)
    then
      Received := Received ∪ {m};
      if(q ≠ p) then B-multicast(g, m); end if
      R-deliver m;
    end if
  end if

```

NOTA: R-Multicast è una primitiva o tutto o niente, se si manda m o non lo riceve nessuno o lo ricevono tutti.

Questa implementazione è corretta in un sistema asincrono, ma inefficiente (il messaggio è inviato $|g|$ volte a ciascun processo), si fanno tanti B-Multicast quanti sono i membri del gruppo.

L'implementazione, però, soddisfa anche la proprietà di **uniform agreement**.

Se ad esempio un processo non è corretto e va in crash dopo aver effettuato R-deliver, poiché in precedenza ha sicuramente effettuato B-deliver, nondimeno dunque tutti i processi corretti riceveranno il messaggio.

NOTA: Se si invertono le istruzioni "R-deliver m " e "if($q \neq p$) then B-multicast(g, m) endif", l'implementazione non soddisfa più la proprietà di uniform agreement.

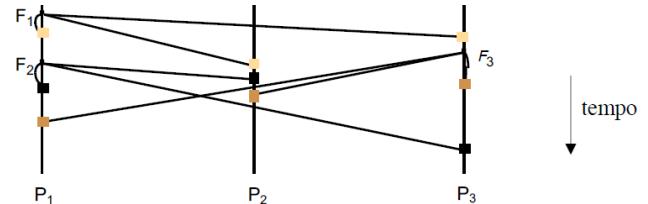
ORDINAMENTO FIFO:

Fino ad ora si è parlato della consegna dei messaggi, ma non dell'ordine in cui arrivano i messaggi.

Uno di essi è l'**ordinamento FIFO** (First In First Out):

Se un processo corretto invia in multicast m e poi m' , allora ogni processo corretto che riceverà m' avrà già ricevuto m .

Nelle versioni uniforme stessa funzionalità sia con messaggi corretti o meno.



ORDINAMENTO CAUSALE:

La relazione happened-before è come dire che qualcosa viene prima di un'altra, ad esempio se due eventi su uno stesso sistema hanno timestamp differenti, si può dire che uno dei due viene prima dell'altro, ma in un sistema distribuito gli orologi dei nodi non sono sincronizzati (drift clock).

Lamport, studioso di sistemi distribuiti, disse che non c'era bisogno di un orologio fisico per stimare l'ordinamento degli eventi, ma si può utilizzare delle relazioni logiche, ovvero la relazione happened-before.

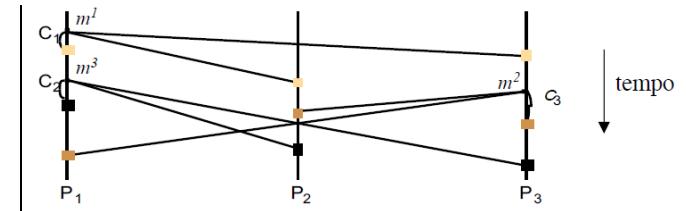
Un evento a è in **relazione happened-before** rispetto ad un elemento b se sussistono le tre seguenti motivazioni:

1. Se gli eventi a e b si verificano sullo stesso processo (quindi gli orologi sono sincronizzati), se il verificarsi dell'evento a ha preceduto il verificarsi dell'evento b, allora $a \rightarrow b$ o $HB\ b$;
2. Se a è l'invio di un messaggio e b è la ricezione del messaggio, allora $a \rightarrow b$ o $HB\ b$;
3. Se due eventi accadono in diversi processi isolati (quindi gli orologi non sono sincronizzati), allora si dice che i due processi sono concorrenti, allora è vero sia $a \rightarrow b$ che $b \rightarrow a$.

L'ordinamento causale estende l'ordinamento FIFO con l'ordinamento della ricezione dei messaggi che, pur inviati da processi diversi, sono in relazione di potenziale causalità. Quindi, se un processo p invia in multicast un messaggio m ed un processo $p' \neq p$ riceve m prima di inviare in multicast m' , allora nessun processo corretto riceve m' a meno che non abbia già ricevuto m .

Nell'ordinamento causale, se l'invio di un messaggio è in relazione happened-before, di un altro invio di un messaggio nell'ambito dello stesso gruppo, allora un processo corretto riceve m' dopo m . Più formale:

Se $\text{multicast}(g, m) \rightarrow \text{multicast}(g, m')$, dove \rightarrow è la relazione happened-before nell'ambito del gruppo g , allora ogni processo corretto che riceverà m' avrà già ricevuto m .



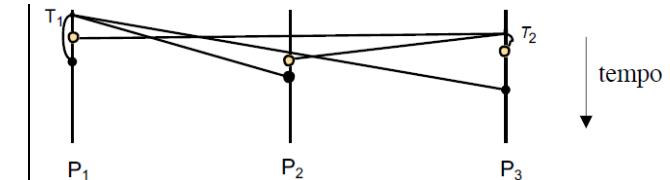
Se non sussiste sincronismo dei clock delle macchine, non è possibile sempre essere d'accordo sull'ordine in cui le cose accadono. L'unico modo per sapere con certezza è se qualcosa collega questi due eventi.

ORDINAMENTO TOTALE:

L'ordinamento qui è in ricezione, ovvero tutti devono ricevere i messaggi nello stesso ordine. Negli altri ordinamenti, invece, le ricezioni sono differenti. Nell'ordinamento causale includeva quello FIFO, quello Totale non implica né quella FIFO e né quella causale, dato che nella definizione non si sa se è stato mandato prima m o m' e se l'invio di m è in relazione happened-before con m' , si sa solo l'ordine di ricezione.

Se un processo corretto riceve i messaggi m e poi m' , allora ogni altro processo corretto che riceve m' avrà già ricevuto m .

Anche qui si può estendere la definizione con quella uniforme coi processi corretti o meno.



IP MULTICAST:

La soluzione più comune per realizzare un sistema di comunicazione di gruppo è di impiegare **IP Multicast**.

IP Multicast è definito al di sopra del protocollo IP, ed un'applicazione può effettuare comunicazioni multicast solo attraverso l'invio di datagrammi UDP.

Un gruppo multicast viene identificato per mezzo di indirizzi di classe D assegnati dalla Internet authority nell'intervallo 224.0.0.1 – 224.0.0.255.

I router sono responsabili dell'opportuna replicazione e istradamento dei pacchetti per la consegna ai membri attraverso il Protocol-Independent Multicast. IP Multicast rappresenta un'ottima soluzione per comunicazioni di gruppo in LAN, ma presenta limiti su WAN:

1. Gli ISP non consentono traffico IP Multicast per ridurre il carico sui router e proteggersi da traffico indesiderato;
2. IP Multicast richiede aggiornamenti all'attuale infrastruttura di rete, che non sono facilmente sostenibili dal business model degli attuali ISP;
3. IP Multicast viola il principio di stateless del protocollo IP, siccome i router devono mantenere informazioni sulla membership dei gruppi;
4. IP Multicast introduce forti complessità e limiti di scala su scenari di ampia scala come Internet.

Tali motivi hanno portato alla definizione di un nuovo approccio alla comunicazione di gruppo.

IP Multicast

IP

APPLICATION-LEVEL MULTICAST:

Una soluzione per ovviare agli inconvenienti di IP Multicast è quello di implementare le funzionalità di multicasting a livello applicativo piuttosto che a livello trasporto dello stack protocolare ISO/OSI. Le applicazioni sono interconnesse per mezzo di una overlay network, gestiscono i meccanismi di membership al gruppo e si fanno carico delle operazioni di replicazione e disseminazione dei pacchetti.

L'interconnessione al livello overlay può essere strutturata come un albero, oppure non strutturata come una mesh. L'uso di una struttura ad albero implica una maggiore efficienza nella disseminazione, ma anche più vulnerabilità ai fallimenti dei nodi overlay.

L'efficienza (η) è misurata come il numero medio di pacchetti scambianti sui link durante la disseminazione di un messaggio in multicast.

$$\eta_{\text{IP Multicast}} = 1 \text{ mentre } \eta_{\text{ALM}} < 1$$

ALM indica il caso in cui i nodi dell'Overlay sono realizzati dagli stessi computer situati presso gli utenti finali. Il risultato è che la distribuzione ottenibile è caratterizzata da una topologia spesso molto lontana dal caso ideale.

Il termine Overlay Multicast è usato per identificare il caso in cui i nodi dell'overlay sono disposti direttamente nella core network, permettendo di realizzare un albero di distribuzione molto più efficiente.

IMPLEMENTAZIONE ORDINAMENTO FIFO:

L'ordinamento FIFO si realizza mediante l'utilizzo di *numeri di sequenza*, utilizzati per individuare l'ordine dei messaggi.

Si assume che i gruppi non si sovrappongono:

- S^p , contatore dei messaggi inviati dal processo p al gruppo g (ogni gruppo ha il suo contatore);
- R^q , numero di sequenza dell'ultimo messaggio delivered da p, inviato a g da q (un contatore di invio e uno di ricezione);
- **Hold-Back-Queue(p)**, coda dei messaggi fuori sequenza in attesa di essere ricevuti da p.

To FO-multicast(m,g): piggy-back S^p onto m; $B\text{-multicast}(m,g)$; $S^p = S^p + 1$;	To FO-deliver(q,m) with m bearing $S = R^q + 1$: $FO\text{-deliver}(q,m)$; $R^q = S$; To FO-deliver(q,m) with m bearing $S > R^q + 1$: $put m in Hold\text{-Back\text{-}Queue}$; $wait until S(m) = R^q + 1$; $FO\text{-deliver}(q,m)$;
--	---

IMPLEMENTAZIONE R-MULTICAST:

Il protocollo multicast affidabile può essere eseguito su UDP e utilizza il servizio multicast IP per la consegna dei pacchetti. Poiché sia IP multicast che UDP sono protocolli inaffidabili, l'affidabilità si ottiene eseguendo un protocollo affidabile end-to-end a livello di applicazione. Per ottenere l'affidabilità, i pacchetti con errori o pacchetti persi verranno ritrasmessi, utilizzando un protocollo a finestra scorrevole basato sul feedback delle stazioni. Nel contesto dell'ALM oltre alla ridondanza temporale realizzata dalle ritrasmissioni, è possibile adottare anche ridondanza spaziale:

- Impiegando percorsi multipli con copie dei messaggi istradati nei vari percorsi;
- Usando tecniche di codifica per ottenere informazioni addizionali e ricostruire i pacchetti persi dalla rete.

IMPLEMENTAZIONE FIFO R-MULTICAST:

Si può usare la stessa implementazione del FIFO multicast, usando R-multicast al posto di B-multicast.

To FO-multicast(m,g): piggy-back S^p onto m; $R\text{-multicast}(m,g)$; $S^p = S^p + 1$;	To FO-deliver(q,m) with m bearing $S = R^q + 1$: $FO\text{-deliver}(q,m)$; $R^q = S$; To FO-deliver(q,m) with m bearing $S > R^q + 1$: $put m in Hold\text{-Back\text{-}Queue}$; $wait until S(m) = R^q + 1$; $FO\text{-deliver}(q,m)$;
--	---

IMPLEMENTAZIONE ORDINAMENTO CAUSALE:

L'implementazione tiene conto solo delle relazioni HB tra i messaggi multicast, non di quelle tra messaggi diretti (one-to-one) scambiati tra i processi. Si può far uso di orologi vettoriali (**Vector Clocks**), dove l'elemento i-esimo conta il numero di messaggi inviati dal processo i-esimo in relazione HB con il prossimo messaggio. Si assume che i gruppi chiusi non si sovrappongono. Per effettuare il CO-multicast, un processo incrementa il "proprio" elemento nel vettore ed esegue B-multicast al gruppo g del messaggio marcato col vector clock.

Quando p_i fa il B-deliver di un messaggio inviato da p_j , prima di effettuare il CO-deliver deve inserirlo nella hold-back queue finché non ha fatto il CO-deliver di tutti i messaggi in relazione HB con esso.

p_i deve perciò verificare di aver fatto:

- il CO-deliver di tutti i messaggi precedenti inviati da p_j ;
- il CO-deliver di tutti i messaggi di cui p_j aveva fatto il CO-deliver prima di inviare il messaggio in questione.

Queste condizioni possono essere verificate esaminando i vector timestamps dei messaggi ricevuti.

In questo caso, non si ha un unico contatore, ma un vettore di contatori tanti elementi quanti i processi.

Vero che ogni processo incrementa la propria componente del vettore, ma osservando anche gli altri componenti appartenenti a processi diversi si può stimare le relazione happen-before di tutti gli altri eventi

Algorithm for group member p_i ($i = 1, 2, \dots, N$)

On initialization

$$V_i^g[j] := 0 \quad (j = 1, 2, \dots, N);$$

To CO-multicast message m to group g

$$V_i^g[i] := V_i^g[i] + 1; \\ B\text{-multicast}(g, \langle V_i^g, m \rangle);$$

NB: il CO-deliver di un proprio messaggio può essere effettuato immediatamente

On B-deliver($\langle V_j^g, m \rangle$) from p_j , with $g = \text{group}(m)$

place $\langle V_j^g, m \rangle$ in hold-back queue;
 wait until $V_j^g[j] = V_i^g[j] + 1$ and $V_j^g[k] \leq V_i^g[k]$ ($k \neq j$);
 $CO\text{-deliver } m$; // after removing it from the hold-back queue
 $V_i^g[j] := V_i^g[j] + 1$;

IMPLEMENTAZIONE CAUSAL R-MULTICAST:

Il multicast con ordinamento causale affidabile può essere implementato a partire dalla implementazione del Causal Multicast, usando R-multicast al posto di B-multicast.

IMPLEMENTAZIONE ORDINAMENTO TOTALE:

Basata sulla marcatura dei messaggi con identificatori totalmente ordinati, in modo che tutti i processi possano prendere le stesse decisioni. Il delivery è come per l'ordinamento FIFO, ma i numeri di sequenza si riferiscono al gruppo, non al processo.

1 metodo, implementazione mediante processo sequenziatore:

- un messaggio m è inviato sia a g sia a $\text{sequencer}(g)$, etichettato con un id univoco $\text{id}(m)$;
- $\text{sequencer}(g)$ può coincidere con un membro di g ;
- $\text{sequencer}(g)$ gestisce un numero di sequenza per il gruppo s_g , con cui marca i messaggi di cui effettua B-deliver;
- $\text{sequencer}(g)$ annuncia i numeri di sequenza inviando messaggi di ordinamento a g tramite B-multicast;
- un messaggio resta nella hold-back queue finché non può esserne effettuato il TO-deliver, in base al suo numero di sequenza.

1. Algorithm for group member p

On initialization: $r_g := 0$;

To TO-multicast message m to group g
 $B\text{-multicast}(g \cup \{\text{sequencer}(g)\}, \langle m, i \rangle);$

On B-deliver($\langle m, i \rangle$) with $g = \text{group}(m)$
 Place $\langle m, i \rangle$ in hold-back queue;

On B-deliver($m_{\text{order}} = \langle \text{"order"}, i, S \rangle$) with $g = \text{group}(m_{\text{order}})$
 wait until $\langle m, i \rangle$ in hold-back queue and $S = r_g$;
 $TO\text{-deliver } m; // \text{ (after deleting it from the hold-back queue)}$
 $r_g = S + 1;$

2. Algorithm for sequencer of g

On initialization: $s_g := 0$;

On B-deliver($\langle m, i \rangle$) with $g = \text{group}(m)$
 $B\text{-multicast}(g, \langle \text{"order"}, i, s_g \rangle);$
 $s_g := s_g + 1;$

2 metodo, Algoritmo ISIS (Birman, 1987), valido per gruppi aperti o chiusi. I processi concordano collettivamente sui numeri di sequenza, con un algoritmo distribuito, qui non esiste il sequenziatore. Ogni ricevente propone il numero di sequenza al multicaster, che genera il numero "agreed". Ogni processo q di g possiede:

- A_g^q : il più alto numero di sequenza finora convenuto
- P_g^q : il più alto numero di sequenza finora proposto dallo stesso q

Ogni ricevente propone:

$$P_g^q := \text{Max}(A_g^q, P_g^q) + 1$$

e assegna temporaneamente al messaggio il numero proposto, inserendolo nella hold back queue, questa è ordinata con il valore più piccolo in testa. Il multicaster raccoglie le proposte e seleziona il valore maggiore a , quindi effettua $B\text{-multicast} \langle i, a \rangle$ (i è l'id univoco con cui il multicaster ha etichettato m).

Ogni ricevente assegna:

$$A_g^q := \text{Max}(A_g^q, a)$$

e associa il valore a al messaggio, quindi riordina la hold-back queue se il valore convenuto è diverso da quello proposto.

Quando al messaggio in testa alla coda viene assegnato il numero agreed, viene inserito in una delivery queue.

IMPLEMENTAZIONE CAUSAL TOTAL MULTICAST:

Se si combina l'algoritmo per l'ordinamento causale (CO) con quello per il multicast totalmente ordinato (TO) basato sul sequencer si ottiene un ordinamento che è ordinato sia causalmente sia totalmente (C&TO):

- il sequenziatore deve fare il delivery secondo l'ordinamento causale, e il multicast dei numeri di sequenza dei messaggi nell'ordine con cui li riceve;
- i processi nel gruppo di destinazione non effettuano il deliver prima di aver ricevuto un messaggio order dal sequenziatore e il messaggio è il prossimo nella sequenza;
- in tal modo, poiché il sequencer effettua il deliver in ordine causale, e tutti gli altri processi lo effettuano nell'ordine del sequencer, l'ordinamento è sia totale sia causale.

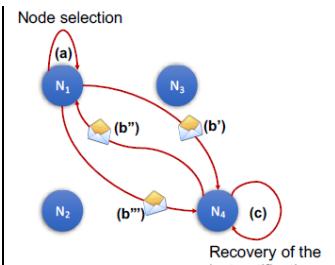
GOSSIPING:

Se si ha un sistema di grandi dimensioni e si vogliono implementare meccanismi di comunicazione di gruppo con affidabilità (non con l'ordinamento) si utilizza allora il Reliable Multicast. Esso permette di inviare il messaggio col B-Multicast, in ricezione se il messaggio non proviene dal mittente ed è nuovo (non è già nella coda), si effettua un B-Multicast e poi un B-Deliver. Quindi, ogni nodo, all'atto di ricezione di un nuovo messaggio, lo si ritrasmette agli altri. Questa primitiva delle problematiche, ovvero quello di generare uno stress sulla rete abbastanza alto.

Nelle reti reali, fare **flooding** utilizzando il R-Multicast è molto oneroso, quindi si adottano degli algoritmi più ottimizzati che hanno le stesse garanzie. Uno di questi algoritmi è il **gossiping**, che rappresenta un algoritmo distribuito per poter garantire la consegna di messaggi sebbene fallimenti di link, processo e di rete si possano verificare nel sistema, ma non i fallimenti bizantini (non essendo un algoritmo bizantino). Realizza una soluzione di **flooding selettivo**, dato che il flooding causa molto stress di rete ma andandolo a fare selettivo si cerca di rimandare i messaggi in maniera oculata.

L'algoritmo di gossip ha varie versioni:

- **Modalità push:** Periodicamente, ogni processo (o nodo) decide di inviare un messaggio a un insieme k di interlocutori scelti in maniera casuale (k è un parametro dell'algoritmo detto fan-out), mentre nel Reliable Multicast i messaggi si ritrasmettono a tutti. In questa modalità, un nodo manda l'ultimo messaggio e i riceventi lo possono usare per recuperare eventuali perdite.
- **Modalità pull:** Un nodo manda un messaggio a k altri nodi, questi nodi con quel messaggio riescono a capire se hanno perso degli invii e richiedono una ritrasmissione, questa è una modalità periodica con un pull, ovvero richiede esplicitamente una ritrasmissione con un ack.



Il pull style manda la "storia" dei messaggi ricevuti, richiede che si invia una esplicita richiesta di ritrasmissione, per tanto i tempi di latenza sono più lunghi, ma la rete non viene saturata di messaggi. Il push style, invece, si manda l'ultimo messaggio (o gli ultimi n messaggi), esso consente di avere meno interazioni e la probabilità che una interazione vada a buon fine è più alta, il problema è che la rete viene caricata di messaggi, ma la latenza è più bassa.

L'informazione che viene scambiata (per la ritrasmissione), o digest, può essere di due tipi:

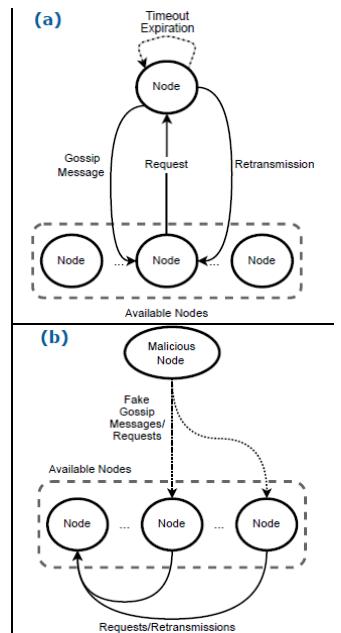
- **Positiva:** il messaggio di gossiping contiene gli identificativi dei messaggi correttamente ricevuti;
- **Negativa:** il messaggio di gossiping contiene gli identificativi dei messaggi noti per essere stati persi.

Sono consentite tutte le possibili combinazioni, ma nella pratica nel caso di una strategia di pull si usa un digest negativo, mentre per quella di push uno positivo. Lo schema pull/negative è intrinsecamente reattivo, mentre quello push/positive (*algoritmo probabilistico*) è implementato in accordo ad uno schema proattivo. Consente di ottenere la consegna affidabile con una data probabilità prossima a 1.

SICUREZZA DEL GOSSIP:

Uno schema gossiping convenzionale mostra **vulnerabilità** che possono essere sfruttate in determinati attacchi o per compromettere le applicazioni basate sul gossiping. Gossiping può essere compromesso al fine di realizzare attacchi di tipo **Denial of Service (DoS)** per congestionare una porzione di rete o un determinato nodo, ritrasmettendo una grande quantità di messaggi.

Normalmente la scelta dei nodi verso cui inviare messaggi di gossiping è casuale, oppure in base a determinate metriche di qualità.



Una compromissione può causare l'invio di molti messaggi di gossiping verso un nodo, o un sotto-insieme di nodi, con un effetto di «bombardamento».

La protezione contro compromissioni del gossiping è di fondamentale importanza, esistono delle **contromisure**:

- Gli attacchi Sybil vengono evitati usando un servizio di **Certificate Authority (CA)** che assegna identità ai partecipanti al gossiping in modo che gli avversari malintenzionati non possano falsificare le identità;
- Un'altra soluzione è usare primitive crittografiche per proteggere integrità (cifrando i messaggi) e autenticità dei messaggi scambiati;
- Un approccio diverso è considerare validi i messaggi di gossiping se ricevuti da un numero elevato di peer (stesso messaggio inviato da più nodi);
- Un'alternativa è inviare i messaggi di gossiping a porte conosciute pubblicamente mentre si ritrasmettono a porte selezionate casualmente, e scegliendo casualmente i messaggi da scambiare dai digest per evitare di essere sopraffatti da messaggi fasulli.

2.1 FAILURE DETECTORS

Uno dei problemi che si riscontra nel progetto di un algoritmo distribuito è rilevare quando un determinato processo fallisce (con riferimento a fallimenti per crash e non bizantini) e prendere contromisure.

Un **failure detector** è un servizio in grado di stabilire se un processo è fallito. Tale servizio è tipicamente implementato da un insieme di oggetti, ciascuno locale ad uno degli N processi del sistema distribuito. Ciascun oggetto (detto **local failure detector**) esegue un algoritmo di detection in congiunzione con le sue controparti eseguite localmente ad altri processi.

NOTA: si assume che i processi falliscano solo per crash, che abbiano canali di comunicazione affidabili, e che il fallimento di un processo non impedisca ad altri di comunicare.

In un **sistema asincrono**, il rilevamento di fallimenti è soggetto ad errori poiché un processo può essere considerato come fallito sebbene questo sia correttamente in esecuzione (tempi di attesa a causa della rete).

La maggior parte dei failure detector sono **inaffidabili**, restituendo uno dei due risultati sullo stato di fallimento di un processo q:

- **Unsuspected:** il detector ha di recente avuto prova che il processo q non è fallito. Quindi, un messaggio è stato ricevuto di recente dal processo, tuttavia, ciò non esclude la possibilità che il processo sia fallito dopo che ha risposto;
- **Suspected:** il detector ha indicazioni sul fatto che il processo q potrebbe essere fallito. Il processo non riceve messaggi per un tempo maggiore del massimo nominale, tuttavia, il processo potrebbe funzionare correttamente, ma non rispondere a causa di partizionamenti di rete o eccessivi rallentamenti nell'esecuzione (congestione di rete o di esecuzione).

Entrambi i risultati sono da considerarsi suggerimenti, potrebbero non accuratamente riflettere il reale stato del processo q.

Un failure detector **affidabile** è sempre accurato nel determinare il crash di un processo. Può restituire i risultati:

- **Unsuspected:** come nel caso di detector inaffidabili, rappresenta solo un *suggerimento*;
- **Failed:** il detector determina con certezza che il processo q è fallito. Il processo non eseguirà alcuna azione ulteriore (per definizione, un processo fallito per crash permane nel suo stato, senza eseguire altri passi di elaborazione).

Quando si studiano i failure detector esiste una **caratterizzazione**, di Chandra e Toueg, in termini di:

- **Completezza (completeness):** C'è un istante di tempo dopo il quale ogni processo andato in crash è permanentemente sospettato da un processo corretto. La completezza indica la capacità del detector di rilevare i fallimenti, evitando che processi falliti siano considerati corretti (*falsi negativi*). Esistono 2 livelli di completezza:
 1. **Completezza forte (strong completeness):** Prima o poi ogni processo fallito è permanentemente sospettato *da ogni* processo corretto;
 2. **Completezza debole (weak completeness):** Prima o poi ogni processo fallito è permanentemente sospettato *da qualche* processo corretto.
- **Accuratezza (accuracy):** C'è un istante dopo il quale nessun processo corretto è sospettato da un processo corretto. L'accuratezza indica, invece, la capacità di non rilevare un fallimento in modo errato, cioè di non considerare falliti processi che sono invece corretti (*falsi positivi*). Esistono 4 livelli di accuratezza:

- 1 e 2. I primi 2 sono detti di **accuratezza perpetua (perpetual accuracy)**;
3. **Accuratezza forte (strong accuracy):** I processi corretti non sono mai sospettati da alcun processo corretto;
4. **Accuratezza debole (weak accuracy):** C'è almeno un processo corretto che non è mai sospettato da alcun processo corretto.

Nei sistemi asincroni, tali livelli sono difficili da soddisfare, si possono rendere meno stringenti consentendo a un detector di sospettare di un processo corretto in un qualsiasi istante dell'esecuzione, ma prima o poi (eventually) deve soddisfare le proprietà di accuratezza forte o debole.

- **Accuratezza forte eventuale (eventual strong accuracy):** C'è un istante di tempo dopo il quale i processi corretti non sono sospettati da alcun processo corretto;
- **Accuratezza debole eventuale (eventual weak accuracy):** C'è un istante di tempo dopo il quale almeno un processo corretto non è sospettato da alcun processo corretto.

NOTA: la completezza da sola non serve a molto, basta considerare tutti i processi come falliti per piena completezza.

CLASSIFICAZIONE FAILURE DETECTOR (giusto per presentarli):

Dai livelli appena introdotti, è possibile definire otto tipi di failure detector.

Relazione:

$$\mathcal{P} \subseteq \mathcal{L}; \quad \diamond\mathcal{P} \subseteq \diamond\mathcal{L}; \quad \mathcal{S} \subseteq \mathcal{W}; \quad \diamond\mathcal{S} \subseteq \diamond\mathcal{W}$$

Completezza	Accuratezza			
	Strong	Eventually Strong	Weak	Eventually Weak
Strong	Perfect \mathcal{P}	Eventually Perfect $\diamond\mathcal{P}$	Strong \mathcal{S}	Eventually Strong $\diamond\mathcal{S}$
Weak	\mathcal{L}	$\diamond\mathcal{L}$	Weak \mathcal{W}	Eventually Weak $\diamond\mathcal{W}$

- **Perfect failure detector P (strongly complete, strongly accurate)**: tutti i processi falliti prima o poi sono sospettati permanentemente da ogni processo corretto, e tutti i processi corretti non sono mai sospettati da alcun processo corretto.
- **Eventually perfect failure detector [rombo] P (strongly complete, eventually strongly accurate)**: tutti i processi falliti prima o poi sono sospettati permanentemente da ogni processo corretto, e tutti i processi corretti sono non più sospettati da un istante t in poi da alcun processo corretto.
- **Strong failure detector S (strongly complete, weakly accurate)**: tutti i processi falliti prima o poi sono sospettati permanentemente da ogni processo corretto, e almeno un processo corretto non è mai sospettato da alcun processo corretto.
- **Eventually strong failure detector [rombo] S (strongly complete, eventually weakly accurate)**: tutti i processi falliti prima o poi sono sospettati permanentemente da ogni processo corretto, e almeno un processo corretto è non più sospettato da un istante t in poi da alcun processo corretto.
- **Weakly complete, strongly accurate failure detector L**: tutti i processi falliti prima o poi sono sospettati permanentemente da qualche processo corretto, e tutti i processi corretti non sono mai sospettati da alcun processo corretto.
- **Weakly complete, eventually strongly accurate failure detector [rombo] L**: tutti i processi falliti prima o poi sono sospettati permanentemente da qualche processo corretto, e tutti i processi corretti sono non più sospettati da un istante t in poi da alcun processo corretto.
- **Weak failure detector W (weakly complete, weakly accurate)**: tutti i processi falliti prima o poi sono sospettati permanentemente da qualche processo corretto, e almeno un processo corretto non è mai sospettato da alcun processo corretto.
- **Eventually weak failure detector [rombo] W (weakly complete, eventually weakly accurate)**: tutti i processi falliti prima o poi sono sospettati permanentemente da qualche processo corretto, e almeno un processo corretto è non più sospettato da un istante t in poi da alcun processo corretto.

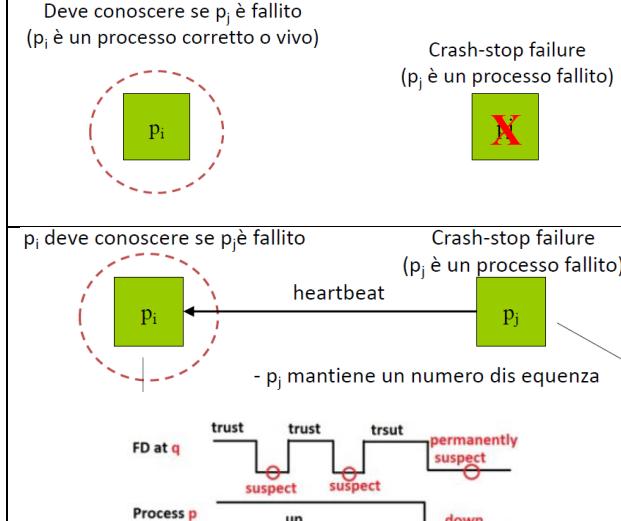
IMPLEMENTAZIONE FAILURE DETECTOR:

In generale, un failure detector è implementato con due processi p_i e p_j , il processo p_i monitora l'altro processo p_j . Il processo p_j può fallire con un crash-stop (e non crash-recovery), e il processo p_i deve capire se p_j è fallito. I meccanismi di realizzazione del monitoraggio del corretto funzionamento di un processo sono 2:

1. **Ping-Ack (proattivo)**, p_i manda il messaggio a p_j ed attende la risposta;
2. **Heartbeat (reattivo)**, p_i manda messaggi periodici, p_i monitora il suo funzionamento e capire se è fallito

Un failure detector **inaffidabile** può essere implementato in un sistema asincrono attraverso la tecnica di heart beat:

- Ogni processo p_j invia un messaggio " p_j -is-alive" a ciascun altro processo nel sistema distribuito, ogni T secondi;
- Il failure detector adotta una stima del massimo tempo di trasmissione di un messaggio, pari a Δ secondi;
- Se il failure detector locale al processo p_i non riceve un " p_j -is-alive" entro $T+\Delta$ secondi, allora riporta che il processo p_j è *Suspected*;
- Se successivamente riceve un " p_j -is-alive", corregge la sua stima, e riporta a p_i che il processo p_j è *Unsuspected*. Altrimenti, assume il processo come fallito.



La bontà della risposta fornita ad un processo dipende dall'informazione disponibile localmente. Un failure detector potrebbe dare suggerimenti errati a causa di condizioni di comunicazione estremamente variabili. Un processo che fallisce per *crash* inevitabilmente non invierà più messaggi *p-is-alive*, quindi, prima o poi, ogni altro processo corretto, non ricevendo i messaggi, lo sospetterà di fallimento. Ciò implica **strong completeness**.

Tuttavia, per l'accuratezza possiamo solo pensare che prima o poi i messaggi *p-is-alive* di un processo corretto arriveranno, quindi ci sarà un processo corretto che non sarà sospettato da un altro processo corretto. Ciò implica **eventual weak accuracy**.

La scelta dei tempi T e Δ incide sulle prestazioni del *detector*:

- Tempi brevi portano ad indicare spesso come *Suspected* processi che in realtà non sono falliti (falsi positivi), e richiedono molta banda di rete per i messaggi " p_j -is-alive".
- Tempi lunghi portano ad indicare spesso come *Unsuspected* processi che in realtà sono falliti (falsi negativi).

Una soluzione pratica al problema prevede l'uso di valori adattativi per T e Δ , che riflettano le reali condizioni correnti della rete.

Il failure detector resta inaffidabile, ma l'accuratezza aumenta.

IMPLEMENTAZIONE PER SISTEMA PARZIALMENTE SINCRONO:

Un sistema parzialmente sincrono è un sistema inizialmente asincrono ma che dopo un istante (sconosciuto) t diventa sincrono. Questa parziale sincronia si può impostare con dei timeout e altre tecniche.

Per un sistema parzialmente sincrono, rendendo il timeout adattivo, il *detector* può essere reso **eventually perfect** ([rombo] P).

La **strong completeness** sussiste per fallimenti di tipo *crash-stop*. Per i *crash-recovery*, il sistema si riprende e quindi la completezza del failure detector ne viene ad impattare.

Per la **eventual strong accuracy**, si osservi che il sistema diventa sincrono da un istante t in poi, dopo il quale si conosce l'intervallo di tempo necessario per inviare un messaggio da p a q , per cui, prima o poi, nessun processo corretto sarà sospettato da un altro processo corretto. Nel periodo di asincronia si possono avere falsi negativi e falsi positivi, quando il sistema diventa sincrono la qualità del detector aumenta (il failure detector non è perfetto dato che non è un sistema sincrono ma lo diventa durante la sincronia).

Implementazione del failure detector parzialmente sincrono:

Periodicamente si effettua l'heartbeat e si riesce a sospettare o meno i processi.

```

 $Output_p \leftarrow \emptyset$  // insieme dei sospettati da p
 $\forall q \in \Pi$  // per ogni altro processo q del sistema
 $\Delta_p(q) \leftarrow \text{timeout di default}$  / // per il processo q
begin
    Task 1: periodicamente
        invia  $p\text{-is-alive}$  a tutti gli altri processi del sistema
    Task 2: periodicamente
         $\forall q \in \Pi$  // per ogni processo nel sistema
        if  $q \notin Output_p$  & // se non riceve il messaggio e q non è già
        sospettato
             $p$  non ha ricevuto  $q\text{-is-alive}$  negli ultimi  $\Delta_p(q)$  istanti
             $Output_p \leftarrow Output_p \cup \{q\}$  // lo aggiunge ai sospettati
    Task 3: alla ricezione di  $q\text{-is-alive}$  da un processo  $q$ 
        If  $q \in Output_p$  // se riceve il messaggio da un processo
        sospettato
             $Output_p \leftarrow Output_p - q$  // lo rimuove dalla lista dei sospettati
             $\Delta_p(q) \leftarrow \Delta_p(q) + 1$  // e incrementa il timeout per quel processo
end

```

IMPLEMENTAZIONE DETECTOR AFFIDABILE:

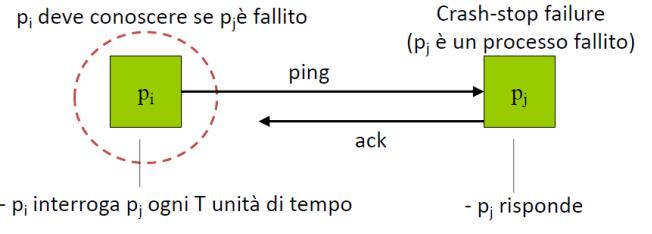
L'algoritmo di detection presentato diventa **affidabile** in un sistema sincrono. Nell'ipotesi di sistema sincrono, Δ diventa un limite superiore assoluto al ritardo di trasmissione di un messaggio, e non più una stima.

L'assenza di un messaggio " $p_j\text{-is-alive}$ " entro $T+\Delta$ permette al failure detector locale di dichiarare con certezza che p è fallito.

PROTOCOLLO SU MESSAGGI APPLICATIVI:

È evidente che l'algoritmo di *detection* presentato, sia esso utilizzato per un sistema asincrono, parzialmente sincrono o sincrono, è molto costoso, vista la richiesta di invio periodico di messaggi di *heart beat*, esempio avere 100 nodi si hanno 100 messaggi periodici sulla rete.

Può introdotto un protocollo di *detection* basato principalmente su messaggi applicativi e che fa uso (per la *detection*) di messaggi di controllo solo quando il processo che sta monitorando non invia alcun messaggio applicativo al processo monitorato.



Φ ACCRUAL FAILURE DETECTOR:

Una nuova astrazione, chiamata **accrual failure detectors**, promuove flessibilità ed espressività operando con un livello di sospetto Φ (phi) su una scala continua, invece di tradizionale informazioni di natura booleana (fiducia vs sospetto).

In parole povere, questo valore Φ rappresenta il grado di certezza che un corrispondente processo monitorato si è arrestato in modo anomalo. Se il processo si blocca effettivamente, è garantito che il valore si accumuli nel tempo e tenda all'infinito.

Viene lasciato alle applicazioni il compito di impostare una soglia di sospetto appropriata in base ai propri requisiti di qualità del servizio.

- Una soglia bassa è propensa a generare molti sospetti errati ma garantisce un rilevamento rapido in caso di incidente reale;
- Al contrario, una soglia alta genera meno errori ma richiede più tempo per rilevare gli arresti anomali effettivi.

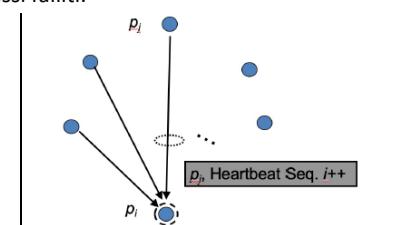
Heartbeat arrivano dalla rete e il loro tempo di arrivo viene memorizzato nella finestra di campionamento. I campioni vengono utilizzati per stimare una certa distribuzione degli arrivi. Il tempo dell'ultimo arrivo T_{last} , il tempo attuale e la distribuzione stimata vengono utilizzati per calcolare il valore corrente di Φ . Le applicazioni attivano sospetti in base a una certa soglia (Φ_1 per App. 1 e Φ_2 per App. 2), o eseguono alcune azioni in funzione di Φ .

FAILURE DETECTOR NEI SISTEMI DISTRIBUITI:

Esistono 3 approcci differenti per realizzazioni concrete su chi effettua il monitoraggio e rilevamento dei processi falliti:

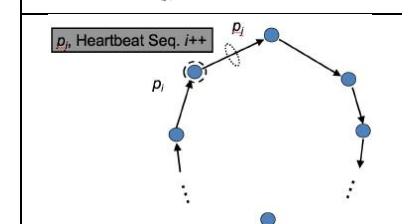
1. **Centralizzato**: Dove un unico elemento riceve gli heartbeat, stima gli altri processi se sono corretti o meno.

Semplice da realizzare ed in grado di avere una completa visione del sistema, ma p_i diventa una criticità, perché se lo si va ad attaccare viene compromessa tutta la failure detection.



2. **Ad anello**: si realizza una topologia ad anello e si manda l'heatbeat al nodo vicino.

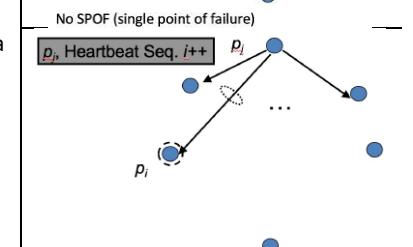
Più complesso da realizzare, non presenta un nodo critico, ma è vulnerabile a possibili problemi sulla rete.



3. **Distribuito**: soluzione più adottata, dove l'heartbeat viene mandato ad n processi sottoinsieme del sistema distribuito.

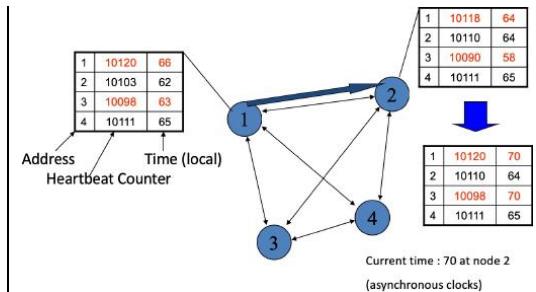
Una soluzione che tollera maggiormente problemi di processo e di rete, ma genera un forte carico di rete e consumo energetico.

Una realizzazione più efficiente dell'approccio distribuito si basa sul gossiping.



Il funzionamento dell'algoritmo è il seguente:

- Ogni processo mantiene un elenco di membri;
- Ogni processo incrementa periodicamente il proprio contatore di heartbeat;
- Ogni processo periodicamente invia per mezzo del gossiping la sua lista di membri;
- Al ricevimento, gli heartbeat vengono uniti e l'ora locale viene aggiornata.



Mentre l'heartbeat generale afferma che un processo è attivo, questo heartbeat afferma che un processo è attivo e contiene informazioni anche sugli heartbeat di altri processi con cui ha interagito, aiutando la failure detection, in quanto, contiene informazioni sia sulla failure detection locale che sullo stato del processo stesso, ed al ricevimento di questo messaggio si va ad unire tutta l'informazione locale, aiutando così la detection anche degli altri nodi. Se un processo perde un heartbeat per effetto di una comunicazione di rete, ma un processo lo ha già ricevuto, passando le informazioni ad un altro processo, il processo che aveva perso l'heartbeat recupera l'informazione persa e aggiorna il suo meccanismo di failure detection.

Il tempo di aggiornamento dipende dal numero di nodi, $O(\log(N))$ è il tempo per un aggiornamento heartbeat di propagarsi e raggiungere tutti i processi del sistema con alta probabilità. Lo schema è molto robusto contro gli errori, anche se un gran numero di processi si blocca, la maggior parte / tutti i processi rimanenti ricevono comunque tutti gli heartbeat.

- Se l'heartbeat non è aumentato per più di T_{fail} secondi, il processo corrispondente viene considerato non corretto, con T_{fail} solitamente impostato su $O(\log(N))$;
- L'entry nella lista del processo considerato fallito non cancellata immediatamente, ma si attende altri $T_{cleanup}$ secondi (di solito pari a T_{fail}).

CONSENSO CON FAILURE DETECTOR:

Il failure detector lo si può inserire nell'algoritmo di consenso. Alcuni sistemi reali utilizzano *detectors* "affidabili da progetto" per raggiungere il consenso. Quindi, i processi si accordano sull'*indicare* un processo p come *Failed* se p non risponde per più di un certo tempo nominale massimo, anche in un sistema asincrono.

In realtà, p potrebbe non essere fallito, ma i rimanenti processi agiscono come se lo fosse, essi fanno diventare p "fail-silent", scartando tutti i messaggi che ricevono da esso. In pratica, ciò equivale a trasformare un sistema asincrono in un sistema sincrono.

Un altro approccio consiste nell'usare failure detectors inaffidabili.

Il consenso viene raggiunto consentendo ai processi Suspected di agire come corretti e partecipare alla decisione, anziché escluderli. Chandra e Toueg hanno dimostrato che il consenso può essere raggiunto, anche in un sistema asincrono e usando detectors inaffidabili, se il numero di processi falliti non eccede $N/2$ e la comunicazione è affidabile.

Il più debole failure detector che può essere usato allo scopo è un eventually weak failure detector. Chandra e Toueg hanno dimostrato che in un sistema asincrono non è possibile implementare un eventually weak failure detector solo attraverso scambio di messaggi.

Tuttavia, un detector adattativo (con timeout variabili) va bene in molti casi pratici.

2.2 CONSISTENZA DEI DATI

In un sistema di grandi dimensioni si ha una tecnica importante che è quella della **replicazione dei dati**, che serve per ridondanza, garantire fault tolerance, e per velocità, incrementando i tempi di risposta. La replicazione, oltre a questi vantaggi, richiede che i dati siano **consistenti**, cioè se si hanno i dati replicati n volte, tutti gli n dati hanno lo stesso stato, ovvero che ci sia agreement. In più, le modifiche ad una replica devono essere eseguite su tutte le altre repliche, e quando si effettua una lettura su un dato si ha lo stesso risultato su una qualsiasi sua replica.

L'idea dei sistemi distribuiti, quando si ha la replicazione, è quella di avere una astrazione della memoria condivisa.

MODELLI DI CONSISTENZA:

Esiste un **modello di consistenza** (o semantica della consistenza), che è un **contratto** tra i processi e l'archivio di dati (distribuito). Se i processi rispettano un certo **insieme di regole** (o protocolli), l'archivio conterrà valori corretti, mentre processi concorrenti possono aggiornare simultaneamente un archivio di dati.

Gli accessi (lettura/scrittura) richiedono un tempo finito, ma sono operazioni diverse operate su processori diversi, che si possono sovrapporre nel tempo. Bisogna garantire che le operazioni conflittuali siano eseguite sulle varie repliche con un ordinamento con una chiara semantica.

In generale, nella consistenza bisogna gestire due tipologie di conflitto:

- **Conflitto read-write**: una lettura e una scrittura concorrentemente sullo stesso dato;
- **Conflitto write-write**: due scritture concorrentemente sullo stesso dato.

Il requisito generale è che una Read restituisca il valore scritto dalla Write più recente.

NOTA: "Più recente" è ambiguo in presenza di repliche e accessi concorrenti. Il modello di consistenza specifica come viene determinata/definita la Write "più recente" e rispetto a chi.

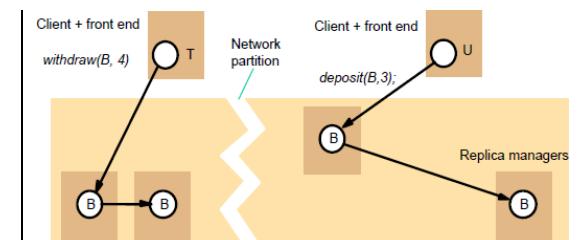
Esistono diversi modelli di consistenza che specificano diverse scelte, e sono:

- **Consistenza stretta**, tutti i processi vedono gli accessi condivisi nello stesso *ordine assoluto di tempo*;
- **Consistenza linearizzabile**, tutti i processi vedono gli accessi condivisi nello stesso *ordine*, gli accessi sono ordinati in base ad un timestamp globale (non unico);
- **Consistenza sequenziale**, tutti i processi vedono gli accessi condivisi nello stesso *ordine*, gli accessi non sono ordinati temporalmente;
- **Consistenza causale**, tutti i processi vedono gli accessi condivisi *correlati causalmente* nello stesso *ordine*.

Un sistema asincrono è **partizionabile**, segmentando la rete in più parti.

In presenza di un partizionamento, per mantenere la consistenza, bisogna bloccare il sistema, rendendolo indisponibile. Se venissero servite le richieste dalle due partizioni, ci sarebbe inconsistenza, quindi il sistema è **available**, ma **non consistente**.

Il teorema CAP formalizza questa situazione.



TEOREMA CAP:

Ogni sistema in rete che condivide dati può avere, in un dato istante, al più due delle tre proprietà desiderabili:

- **Consistency** (C), avere una copia aggiornata dei dati. Tutti i client vedono lo stesso stato, anche in presenza di aggiornamenti;
- **Availability** (A), disponibilità dei dati. Tutti i client possono accedere a qualche replica dei dati, anche in presenza di guasto;
- **Tolerance to network partitions** (P), la proprietà di sistema vale anche se il sistema è partizionato.

Se la priorità è la **consistenza**, si rinuncia all'availability (Sistema CP).

Se la priorità è l'**availability**, si rinuncia alla consistenza (Sistema AP), usando un modello di consistenza più "debole" (Eventual consistency).

Quando si adottano sistemi CP e AP, lo sviluppatore deve sapere cosa "offre" il sistema:

- CP: il sistema può non essere disponibile ad eseguire una write. Lo sviluppatore deve gestire il fallimento di una write causato dall'eventuale indisponibilità del sistema;
- AP: il sistema può accettare sempre una write, ma, sotto certe condizioni, una read non riporterà il risultato della write più recente. Lo sviluppatore deve decidere se il client può richiedere sempre l'accesso all'ultimo aggiornamento.

EVENTUAL CONSISTENCY:

Nel considerare le partizioni, si adotta spesso un modello di consistenza rilassato, detto **consistenza finale (eventual consistency)**. Essa garantisce che se non si verificano aggiornamenti, tutte le repliche (distribuite geograficamente) diventano gradualmente consistenti entro una finestra temporale (detta **inconsistency window**).

In assenza di failure, la dimensione dell'inconsistency window dipende da ritardi di comunicazione, carico del sistema e numero di repliche.

I **vantaggi** e che è semplice da realizzare e poco costosa, mentre gli **svantaggio** sono se l'utente accede a repliche diverse in un breve intervallo di tempo può accedere a dati non aggiornati, in tal caso, occorre risolvere il conflitto (**reconciliation**).

Il costo di garantire un maggior livello di consistenza ricade sullo sviluppatore dell'applicazione, esso deve sapere quale grado di consistenza viene offerto dal sistema. Con la consistenza finale, può accadere che una read non restituisca il valore della write più recente, lo sviluppatore deve decidere se tale inconsistenza (staleness dei dati) è accettabile per l'utente dell'applicazione.

RISOLUZIONE CONFLITTI:

Esistono strategie per decidere come risolvere conflitti su copie divergenti a causa di aggiornamenti concorrenti:

- Un approccio comune è "last writer wins" (in Cassandra), ovvero etichettare i dati con orologi vettoriali per catturare la causalità tra diverse versioni dei dati;
- Un'alternativa è demandare la risoluzione all'applicazione stessa (Amazon Dynamo) che invoca un gestore dei conflitti (conflict handler) specificato dall'utente;

La riconciliazione può essere effettuata a tempo di read (Amazon Dynamo) o, alternativamente, durante una write (write repair) e asynchronous repair.

3. INTRO ALLA BLOCKCHAIN

La blockchain nasce dall'esigenza di dover trasferire un bene sulla rete. In pratica, esistono due nodi di una rete che intendono scambiarsi delle risorse (asset), e l'obiettivo è riuscire a trasferire tale bene in maniera affidabile e sicura, anche se i nodi possano essere bizantini, la rete sia poco affidabile o ci sono degli attaccanti che vogliono manomettere il trasferimento.

I **modelli di fallimento** di questa tecnologia sono:

- **Fallimenti dei nodi**: crash o hang;
- **Fallimenti dei link**: crash persistente o intermittente;
- **Malfunzionamenti di rete**: perdita, alterazione o ritardo anomale di pacchetti o partizionamento della rete.

I **modelli di attacco**, invece, sono:

- **Replay Attack**: una transazione valida viene maliziosamente ripetuta o ritardata, con l'intento di manomettere il trasferimento;
- **Man-in-the-middle Attack**: la comunicazione è osservata e alterata da una terza parte non autorizzata;
- **Masquerade Attack**: un nodo malizioso utilizza un'identità forgiata ad hoc per la comunicazione. Un caso particolare (impersonation) è che un nodo impersona un altro legittimo;
- **Byzantine Behaviour**: un nodo si comporta in modo non previsto.

TRANSAZIONI CON MEDIATORE:

La gestione delle transazioni è realizzata impiegando una entità di terze parti (es. una banca) che convalida, supervisiona e preserva le transazioni. Tale soluzione è implementata ad esempio con il protocollo di aggiornamento a due fasi (2PC - Two-phase commit protocol).

In generale, quando si trasferiscono soldi lo si fa sempre con un **mediatore**. La soluzione con un mediatore consente di gestire attacchi del tipo Masquerade o Byzantine, e si richiede che la terza parte sia affidabile.

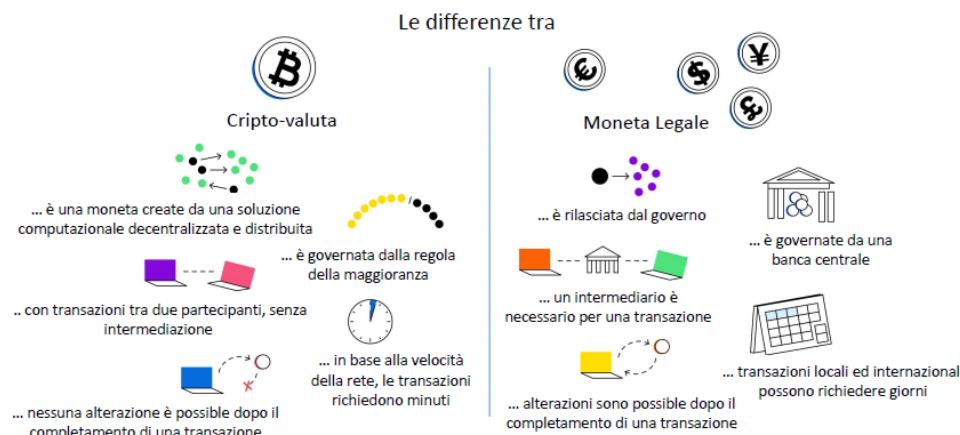
TRANSAZIONE SENZA MEDIATORE:

Per inviare un bene senza la collaborazione di un'entità di terze parti si può definire un protocollo per eseguire transazioni affidabili e sicure in un ambiente inaffidabile e insicuro. Tale soluzione presenta diverse problematiche legate a verifica dell'integrità del messaggio ricevuto, verifica dell'identità dei nodi partecipanti, verifica della validità delle transazioni, ecc...

Il primo ambito di applicazione (2008) è quello finanziario, in particolare per la gestione delle criptovalute. Lo scopo era quello di eseguire transazioni sicure utilizzando un'infrastruttura non sicura. In generale, l'obiettivo della tecnologia Blockchain è creare un ambiente decentralizzato in cui nessuna "terza parte" abbia il controllo delle transazioni e dei dati (**No trusted entities**, esempio banca).

CRYPTO-VALUTE:

Le blockchain sono state teorizzate a supporto delle cosiddette **cripto-valute**, ovvero una rappresentazione digitale di valore basata sulla crittografia e senza intermediazione.

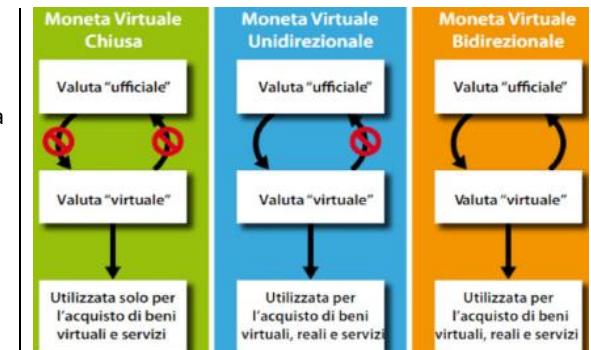


Le cripto-valute rientrano nel caso delle valute virtuali ma sono diverse dalle valute digitali:

Una **valuta virtuale** è una rappresentazione digitale di valore che non è né emessa da una banca centrale né da un'autorità pubblica, né necessariamente collegata a una valuta a corso legale, ma è accettata come mezzo di pagamento e possono essere trasferiti, archiviati o scambiati elettronicamente (ad esempio, valuta in game).

Le cripto-valute sono **valute virtuali bidirezionali**.

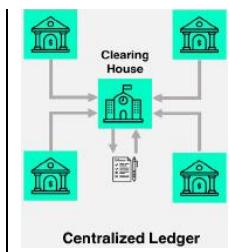
Una **valuta digitale** è una rappresentazione nel virtuale di una valuta a corso legale.



LEDGER (LIBRO MASTRO):

Il **ledger** (o **libro mastro**) è un registro in cui sono contenute tutte le transazioni eseguite tra i partecipanti di un sistema. Il libro mastro è caratterizzato da scritture sistematiche, ovvero le scritture sono registrate in base alla data e all'oggetto a cui si riferiscono. Con riferimento ad ogni oggetto, le operazioni sono indicate rispettando l'ordine cronologico.

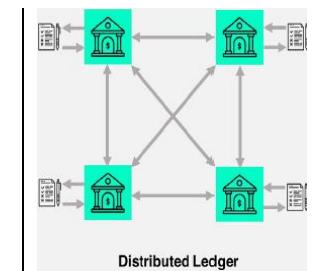
Si tratta di un "append-only multi-party system of record", ovvero un "**log**" di transazioni, "append-only" nel senso che si può scrivere in successione e non si può cancellare e "multi-party system of record" nel senso che le scritture non sono fatte da una sola organizzazione ma da più partecipanti (ognuno col proprio ledger).



In un **distributed ledger** (*libro mastro distribuito*), ciascun nodo della rete memorizza record (transazioni) in una copia locale del libro in maniera sequenziale, sotto forma di **blocchi** ordinati temporalmente.

Affinché i blocchi siano considerati validi, i partecipanti devono raggiungere un **quorum** di consenso (accettati dalla maggioranza).

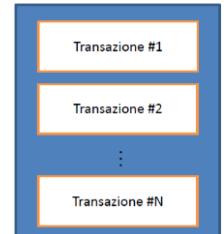
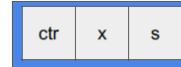
Il libro mastro è costruito come una catena di blocchi, ogni blocco (ad eccezione del **genesis block**, ovvero il primo blocco della catena) contiene un'informazione riguardo il blocco precedente nella catena.



BLOCCO:

Un **blocco** è una collezione di *transazioni*, e sono strutturate in tre parti:

1. **Header**, dove è presente una informazione di controllo (ctr);
2. I **dati** veri e propri (x);
3. Riferimento al **blocco precedente**.



In generale, questa struttura viene chiamata **intestazione del blocco** e il dato dipende dalla blockchain:

- In Bitcoin memorizza i dati finanziari (basati su "UTXO");
- In Ethereum memorizza i dati del contratto (basato sull'account);
- In Namecoin memorizza i dati del nome.

Per inviare un bene senza la collaborazione di un'entità di terze parti si può definire un **protocollo** per eseguire transazioni affidabili e sicure in un ambiente inaffidabile e insicuro. Tale soluzione presenta diverse problematiche legate a:

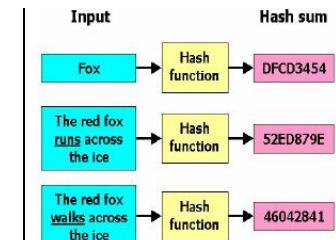
- Verifica dell'**integrità** del messaggio ricevuto, ovvero il contenuto non deve essere manomesso durante il tragitto;
- Verifica dell'**identità** dei nodi partecipanti;
- Verifica della **validità** delle transazioni.

HASH E PARADOSSO DEL COMPLEANNO:

Le **funzioni hash** restituiscono una stringa di poche centinaia di bit (definita hash) a partire da una sequenza di bit di lunghezza arbitraria.

Le funzioni hash crittografiche cercano di rendere **computazionalmente inammissibile** trovare due sequenze che producono la stessa impronta.

La possibilità che due sequenze di bit sottomesse alla stessa funzione di hash diano origine allo stesso valore (collisione) è definito «paradosso del compleanno»: la probabilità che almeno due persone in un gruppo compiano gli anni lo stesso giorno è largamente superiore a quanto potrebbe dire l'intuito.



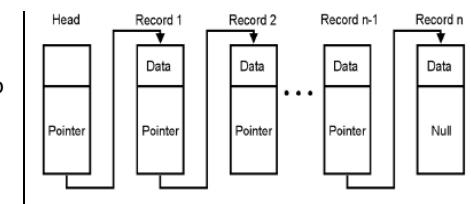
CATENA DI BLOCCHI:

L'hash viene utilizzata per l'integrità dei dati, ma viene anche utilizzata per proteggere i blocchi quando entrano nella catena.

Un blocco è collegato al precedente mediante un hash del blocco precedente. La modifica di un blocco della catena da parte di un nodo comporterebbe la generazione di un valore di hash differente.

Gli altri nodi possono verificare l'**integrità dei blocchi**, andando a verificare se è presente il valore di hash atteso altrimenti il blocco è stato modificato, e rifiutare modifiche di transazioni già validate.

Il contenuto del libro mastro è reso così **immutable**.

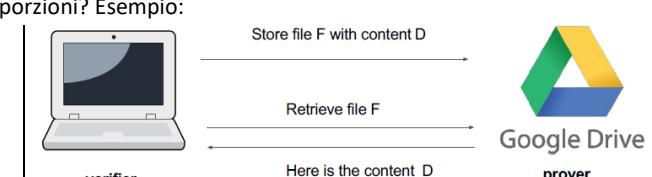


AUTENTICAZIONE CON ALBERI HASH:

Quando si gestiscono le funzioni hash, generalmente si ha un problema di proteggere informazioni e assicurare integrità.

Un blocco è formato da tanti pezzi e quando si fa l'hash di un blocco lo si fa sull'intero blocco. Questo lo si può trasporre con un problema dei file, dove un file ha più sotto porzioni e bisogna gestire l'integrità dell'intero file o delle sotto porzioni? Esempio:

1. L'utente desidera archiviare un file su un server;
2. Il file ha un nome F e un contenuto D;
3. Gli utenti desiderano recuperare il file F in un secondo momento e si ottiene il contenuto D.



Nel recuperare il file in un secondo momento, si vuole che il D che si recupera è uguale al D che si è memorizzato in precedenza (**integrità** da garantire). Ma se il provider fosse malizioso, potrebbe restituire un $D' \neq D$, per accorgersi che il contenuto è diverso, bisognerebbe conservare D.

Quindi, si conserva il file F, lo si archivia sul server e dopo di che quando si chiede la restituzione del file F e restituisce D' anziché D, si confronta la copia e si prendono provvedimenti.

Una **soluzione banale** è che l'utente non elimina D, ma una tecnologia di questo tipo serve per archiviare e non avrebbe senso se si deve comunque conservare i file.

Una **soluzione con Hash** è:

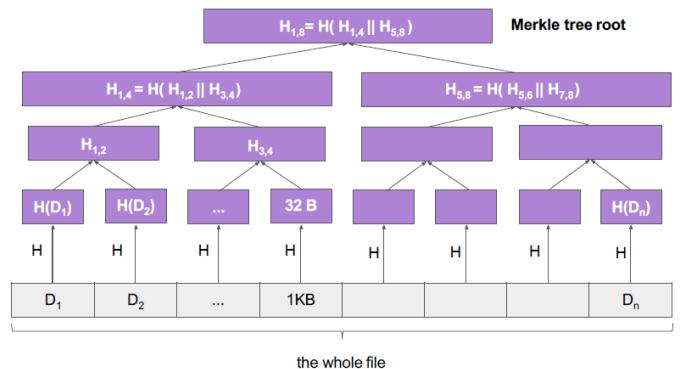
- L'utente invia il file F con i dati D al server;
- Il server archivia (F, D);
- L'utente memorizza H (D), elimina D;
- L'utente richiede F dal server;
- Il server restituisce D';
- L'utente confronta $H (D') = H (D)$?

Se l'utente è interessato al recupero solo di una parte del file, per controllare l'integrità di quella sotto-partita si ha comunque necessità di scaricare l'intero file (in quanto l'hash è applicato sull'intero file), e questo sulla rete può essere un problema di latenza, in quanto bisogna scaricare tutto il file, calcolare l'hash e poi effettuare il controllo. Per evitare questa latenza si usa il **Merkle Tree**, il suo funzionamento è il seguente:

Si calcola l'hash di tutte le sotto porzioni (chunck) del file, dove ogni blocco utilizza una funzione hash crittografica.

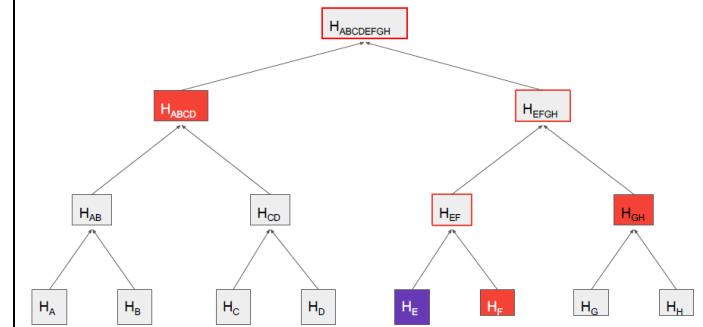
Dopo di che si combinano gli hash a coppie per creare un albero binario, e ogni nodo memorizza l'hash del concat dei suoi figli.

Si itera il processo fintanto che non si ha un unico hash, che sarebbe la radice del Merkle Tree.



Quindi si ha una nuova soluzione utilizzando i **Merkle Three**:

- L'utente crea l'MTR radice del Merkle Tree dai dati del file iniziale D;
- L'utente invia i dati del file D al server;
- L'utente elimina i dati D, ma memorizza MTR (32 byte);
- L'utente richiede il blocco x dal server;
- L'utente restituisce il blocco x e una breve prova di inclusione π ;
- L'utente controlla che il blocco x sia incluso in MTR usando la prova π .



La **prova di inclusione** funziona nel modo seguente:

- Il Prover invia un chunck (es. HE);
- Il Prover invia i vicini lungo il percorso che collega la foglia a MTR (es. HF, HG, HABCD);
- Il Verifier calcola gli hash lungo il percorso che collega la foglia a MTR;
- Il Verifier controlla che radice calcolata corrisponda a MTR (es. HABCDEFGH).

La prova di inclusione richiede $|\pi| \in \Theta(\log |D|)$. Se l'avversario può presentare la prova di inclusione per un nodo foglia errato, allora possiamo compromettere la funzione hash.

IDENTITÀ DEI NODI:

Sebbene applicate a problemi in cui si vuole fare a meno di **trusted entities** (terze parti), le tecnologie blockchain utilizzano anch'esse la firma digitale (e le **trusted authorities** di certificazione) per problemi di sicurezza tipici dei sistemi distribuiti:

- **Authentication**: autenticazione dei nodi della rete;
- **Integrity**: verifica di violazioni dell'integrità dei messaggi scambiati;
- **Non-repudiation**: Non ripudiabilità da parte dei nodi dei messaggi inviati.

VALIDITÀ DELLE TRANSAZIONI:

Si consideri un nodo che desidera "vendere" un bene attraverso una transazione. Il nodo predispone da sé la transazione, che deve sottoporre agli altri nodi affinché sia approvata (validata) da una maggioranza. È possibile che il nodo provi a "spendere" concorrentemente due volte lo stesso bene (double spending):

- In maniera **consapevole**: byzantine behaviour;
- In maniera **inconsapevole**: malfunzionamenti di rete (duplicazione di pacchetti), replay attack.

Un primo passo per risolvere il **problema del double spending** è determinare se un messaggio è "fresco" o è una copia ritrasmessa. È necessario poter identificare univocamente un messaggio contenente un blocco di transazioni eseguite.

Un **nonce** è definito come "un valore variabile nel tempo che ha al massimo una possibilità trascurabile di ripetersi, ad esempio un valore casuale che viene generato di nuovo per ogni utilizzo, un timestamp, un numero di sequenza o una combinazione di questi".

Un nonce può quindi essere ottenuto attraverso un "**timestamp**", e per garantire che ogni blocco sia caratterizzato da un valore differente è necessario che tale valore sia generato da una fonte "fidata".

CERTIFICAZIONE DEL TEMPO:

La certificazione del tempo serve a verificare quando un documento è stato creato, oppure è stata apportata l'ultima modifica in maniera affidabile e non falsificabile. Si impiega un **Trusted Time Server (TTS)** ed esistono due soluzioni:

1. L'utente invia al TTS e una copia del documento, e poi il TTS aggiunge data e ora. Conserva il documento nella cassetta di deposito;
2. Si impiega una soluzione crittografica per proteggere la privacy del documento, e ridurre i costi di trasmissione e memorizzazione, dove l'utente invia al TTS l'hash del documento e il TTS aggiunge data, ora e firma, per poi inviare questo certificato all'utente.

Della seconda soluzione esiste lo standard **ANSI ASC X9.95** che indica che un timestamp attendibile viene emesso da un'**autorità di timestamp** (TSA).

Un hash viene calcolato e inviato alla TSA. La TSA concatena un timestamp all'hash e calcola un hash, che viene firmato digitalmente con la chiave privata della TSA. Questo hash firmato e il timestamp vengono restituiti. Chiunque si fidi del timestamper può quindi verificare che i dati non siano stati creati dopo la data che il timestamper garantisce.

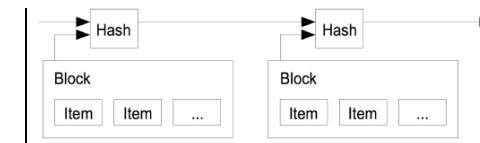
Il valore NONCE, essendo gestito da un unico punto della rete, si ha la garanzia di verificare se due documenti sono lo stesso.

Indipendentemente della correttezza del tempo certificato, si vuole che due documenti sottoposti sequenzialmente al TTS abbiano un nonce che esprima questa sequenzialità:

- Ogni certificato rilasciato contiene l'hash del documento, data, ora, firma del TSS e l'hash del certificato precedente (o di parte di esso);
- Complessivamente i certificati formano una catena di blocchi;
- È inammissibile inserire successivamente un certificato all'interno della catena, il TTS dovrebbe rilevare collisioni per la funzione hash.

Questa soluzione comporta una serie di vantaggi:

- Aggiungere il timestamp rende l'hash più resistente rispetto alle collisioni;
- Risolve il problema del double spending assegnando un ordine temporale alle transazioni e pubblicandole.



PROOF-OF-WORK:

Per svincolarsi dall'utilizzo di un timestamp server (*approccio centralizzato*), alcune soluzioni blockchain (*approccio decentralizzato*) sfruttano un meccanismo differente per la generazione dei nonce. Tale meccanismo è definito **Proof-of-Work (PoW)** (da notare che il PoW non realizza il consenso ma è un meccanismo per proteggere il consenso), che funziona come segue:

Per proporre un nuovo blocco da aggiungere al ledger, un nodo deve dimostrare di aver utilizzato abbastanza risorse di calcolo per risolvere un enigma matematico, il cui risultato è inserito nel blocco stesso.

La Proof-of-Work implica *la computazione di un valore di hash con un determinato numero di bit iniziali pari a zero*, $H(ctr \mid\mid x \mid\mid s) \leq T$.

Ogni nodo (**miner**) cerca di trovare un **nonce** tale da soddisfare il vincolo sul numero di zero prodotti dalla funzione di hash.

Il lavoro medio richiesto è esponenziale rispetto al numero di bit zero richiesti e può essere verificato eseguendo un singolo hash. Tale operazione è molto onerosa dal punto di vista computazionale e richiede la collaborazione di diverse CPU per poter essere realizzata. L'attacco del 51% diventa molto oneroso.

TRANSAZIONI CON CONSENSO DISTRIBUITO:

Tutti i nodi della rete sono a conoscenza di tutte le transazioni, dato che ognuno ha la copia del ledger, e partecipano attivamente alla validazione dei blocchi attraverso il raggiungimento del consenso. Il modello di sicurezza deve contemplare anche il cosiddetto "51% Attack", un gruppo di nodi maliziosi possiede più del 50% della potenza computazionale della rete.

"Nessun algoritmo può garantire il raggiungimento del consenso in un sistema asincrono nel caso di anche un unico fallimento per crash di un processo" (**Teorema di Impossibilità**).

Per raggiungere il consenso in sistemi asincroni in presenza di fallimenti si può:

- **Rilassare i vincoli di consenso** (ciò che fa BitCoin): È consentito che in alcuni casi non si raggiunga il consenso tra tutti i nodi della rete, ovvero solo un sottoinsieme di nodi raggiungono il consenso, e quindi "51% Attack" possibile.
- **Rendere meno "asincrono" il sistema**, sfruttare i periodi di sincronia (ciò che fa Hyperledger Fabric).

BLOCKCHAIN FORKING:

Il rilassamento dei vincoli di consenso può dare origine a **biforazioni** della blockchain.

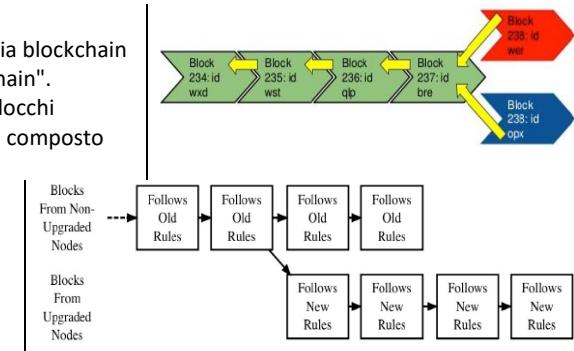
Nel caso di produzione "simultanea" di due blocchi alcuni blocchi aggiungeranno alla propria blockchain il blocco rosso ed altri il blocco blu, fino ad arrivare alla realizzazione di una "forked blockchain".

All'atto della produzione del blocco successivo, esso farà riferimento ad uno solo dei due blocchi precedentemente creati. Tale operazione comporta la rimozione del ramo della blockchain composto dal nodo blu.

In un certo istante di tempo la comunità può ritenere necessario cambiare le regole della blockchain in modo che determinate transazioni ritenute "non valide" fino a tale istante siano considerate " valide" a partire da tale istante e nel futuro.

Tale condizione genera un **hard fork**, che comporta la generazione "voluta" di un nuovo ramo della blockchain per distinguere i nodi che seguono il nuovo regolamento rispetto quelli fedeli al vecchio.

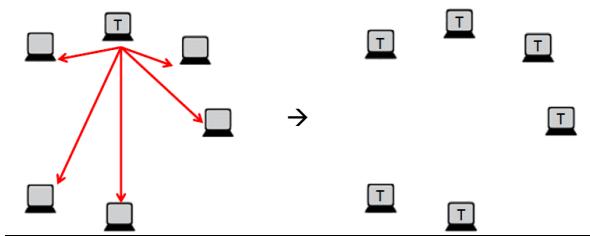
In qualsiasi momento i nodi possono decidere di abbandonare un ramo per seguire l'altro.



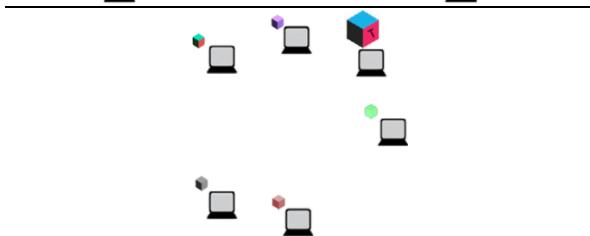
CONSENSO NAKAMOTO (BITCOIN):

L'algoritmo di consenso che ha adottato il BitCoin prende il nome di **consenso di Nakamoto** e che adotta la Proof-of-Work. Funziona come segue:

1. Ogni nuova transazione viene inviata in broadcast a tutti i nodi.
Un nodo genera una nuova transazione T e ne fa il broadcast a tutti i nodi. Gli altri nodi ricevono T.



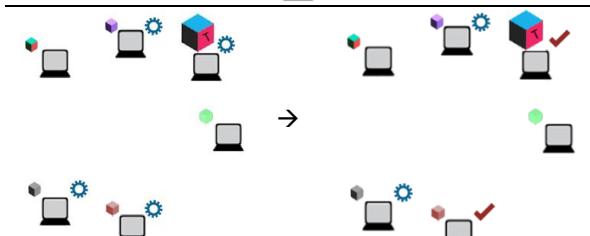
2. Ogni nodo colleziona le nuove transazioni in un blocco (da notare che il consenso è sul blocco e non sulla transazione).



3. Ogni nodo cerca una PoW "difficile" (cioè molte computazioni) per il proprio blocco.

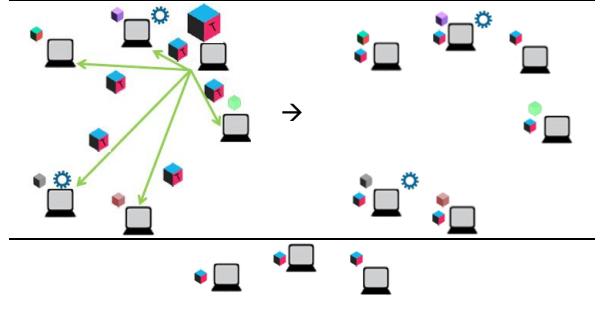
La PoW è il calcolo di un valore (anch'esso un nonce) tale che un hash di blocco + nonce inizi con uno stabilito numero di zeri.

I nodi cercano una Proof-of-Work per il proprio blocco (l'ingranaggio rappresenta il calcolo della PoW). Dopo di che alcuni nodi completano la PoW.



4. Quando un nodo trova una PoW, invia in broadcast a tutti i nodi il blocco, contenente le transazioni e la PoW.

Successivamente i nodi ricevono il blocco.

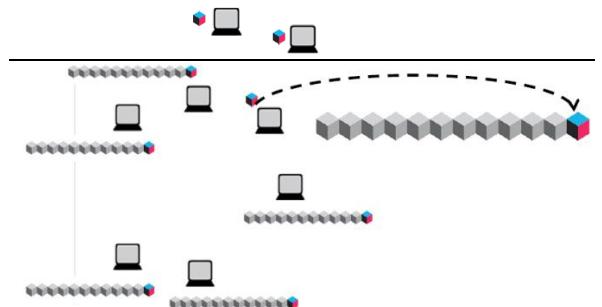


5. Un nodo accetta il blocco solo se:

- Tutte le transazioni (contenute nel blocco) sono valide;
- Le transazioni non sono relative ad un bene già speso (double spending);
- La PoW è valida. A tale scopo il nodo calcola a sua volta l'hash del blocco, e verifica che abbia lo stesso numero di zeri iniziali.

Tutti i nodi accettano il blocco (la blockchain è ancora inalterata).

6. I nodi manifestano l'accettazione del blocco aggiungendolo alla blockchain (contenuto nel ledger) all'atto della creazione del blocco successivo, utilizzando l'hash del blocco accettato per creare il prossimo blocco.



Il tutto avviene in maniera distribuita e decentralizzata, quindi due nodi possono fare concorrentemente il broadcast di blocchi differenti (passo 4). Nodi diversi possono ricevere i due broadcast in ordine diverso, e portare all'accettazione di alcuni blocchi prima di altri blocchi. Ciascun nodo lavora sul primo blocco che riceve, ma conserva l'altro ramo (soft fork). Per ritornare ad uno stato di consistenza, l'indecisione si risolve quando un ramo diventa più lungo, cioè i nodi che lavorano sul ramo meno lungo lo abbandonano e proseguono a lavorare sulla catena più lunga.

È opportuno incoraggiare la cooperazione dei nodi alla creazione del consenso (validazione delle transazioni "corrette"), affinché un blocco sia validato ci deve essere una maggioranza di nodi corretti e che spendono risorse (costruendo la PoW) poiché la PoW è onerosa anche i nodi corretti potrebbero non essere ben disposti a cooperare.

È prevista una politica di incentivi, incoraggiare processi corretti a collaborare nonostante l'onere computazionale, mettendo in minoranza i processi maliziosi. I nodi maliziosi dovranno avere ancora più potenza di calcolo per "contrastare" molti nodi corretti che validano i blocchi. La prima transazione di ogni blocco assegna un compenso al nodo creatore del blocco.

BLOCKCHAIN:

Una **blockchain** è un libro mastro pubblico distribuito (distributed public **ledger**) di transazioni o eventi digitali eseguiti e condivisi tra i partecipanti.

Ogni transazione riportata nella blockchain è validata tramite il raggiungimento del **consenso** tra i nodi del sistema. Una volta memorizzate, le informazioni non possono essere cancellate né modificate. Ogni blocco contiene uno o più record (un record per transazione). Come in un libro mastro, la storia delle registrazioni è immutabile e può essere verificata a partire dal primo blocco (*genesis block*). Le transazioni di Bitcoin sono in chiaro e non cifrate, usa un meccanismo di pseudo anonimizzazione per non collegare l'identità di un utente alle sue attività su Bitcoin. L'attività non viene crittografata perché risulta difficile condividere le chiavi, in quanto, nel consenso Nakamoto, oltre alle transazioni bisognerebbe condividere anche la chiave pubblica in broadcast, ma questa operazione comporta l'identificazione dei nodi, cosa non concessa in Bitcoin.

Le proprietà che una Blockchain fornisce sono:

- **Pubblica Verificabilità**: ogni transazione può essere verificata da ogni partecipante, non esiste una confidenzialità ma un anonimato;
- **Trasparenza**: ogni partecipante ha accesso ad un sottoinsieme di informazioni;
- **Privacy**: l'identità di chi esegue una determinata transazione deve essere tutelata;
- **Integrità**: le informazioni non vengono modificate da fonti non autorizzate;
- **Ridondanza**: dati ripetuti per ogni partecipante del sistema;
- **Assenza di una "Trust Anchor"**.

L'effettiva implementazione di questa tecnologia è fortemente dipendente dal tipo di Blockchain che si intende realizzare:

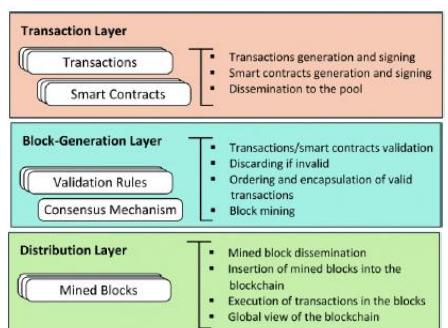
- **Permissionless**: i partecipanti non devono essere preventivamente "autorizzati" per il ruolo che intendono svolgere (es Bitcoin);
- **Permissioned**: le operazioni (tutte o anche solo alcune) possono essere svolte solo da nodi preventivamente autorizzati (es Hyperledger Fabric).

Sono classificabili in base all'ambito:

- **Public**: tutti i blocchi sono visibili a tutti i nodi e ogni nodo può partecipare al consenso (es Bitcoin e Ethereum);
- **Private**: una specifica organizzazione decide quali nodi possono leggere i blocchi e partecipare al consenso (throughput delle transazioni alto);
- **Consortium**: pochi nodi predeterminati possono leggere i blocchi e partecipare al consenso.

La stratificazione tecnologica di blockchain può essere la seguente:

- Il Transaction Layer definisce il linguaggio di codifica e i criteri utilizzati durante la generazione di transazioni e smart contract. Entrambi devono essere firmati prima della loro diffusione per garantire il non ripudio e consentire il controllo dell'accesso e l'autenticazione del loro contenuto.
- I processi di convalida delle transazioni e mining dei blocchi risiedono nel livello block-generation. Tutte le transazioni sono inserite in un blocco candidato, secondo regole di convalida. L'algoritmo di consenso viene adottato per garantire che tutti i nodi della rete abbiano una vista consistente dello stato della blockchain.
- Il processo di distribuzione fa parte del livello di distribuzione, così come l'inserimento del mined block nella blockchain. Tale inserimento riesce solo se l'hash del blocco estratto è corretto, altrimenti viene scartato. Al momento dell'inserimento di un blocco, lo stato globale della blockchain cambia e la vista globale della blockchain viene aggiornata.



3.1 CONSENSO NELLE BLOCKCHAIN

In una rete blockchain, ogni partecipante può validare transazioni e proporre nuovi blocchi. L'obiettivo del **protocollo di consenso nelle blockchain** è di garantire che tutti i nodi partecipanti concordino sulla storia comune delle transazioni nella rete. Le proprietà del consenso della blockchain sono:

- **Termination**: Da ogni nodo onesto, una nuova transazione è sia scartata o accettata nella blockchain, all'interno del contenuto di un blocco;
- **Agreement**: Ogni nuova transazione e il suo blocco che la contiene deve essere sia scartata o accettata da tutti i nodi onesti. Un blocco accettato deve avere lo stesso numero di sequenza per ogni nodo onesto;
- **Validity**: Se ogni nodo riceve uno stesso blocco/transazione valida, esso deve essere accettato nella blockchain;
- **Integrity**: Per ogni nodo onesto, tutte le transazioni accettate devono essere consistenti tra loro (senza double spending). Tutti i blocchi accettati devono essere correttamente generati e collegati con hash in ordine cronologico.

Termination e *validity* rappresentano la *liveness* del sistema, ovvero se si arriva a terminazione del consenso alcune cose sono accettate. *Agreement* e *integrity* rappresentano la *safety* del sistema, il consenso che si arriva è coerente e consistente con quanto accettato precedentemente.

L'*agreement classico* significa arrivare ad una decisione, nell'*agreement* delle blockchain si arriva a decisione con lo stesso numero di sequenza nei blocchi e quindi si impone il total ordering (*agreement* aumentato col **total ordering**), ovvero i nodi partecipanti hanno lo stesso ordine di accettazione e non di invio, così da garantire la consistenza di sequenzialità. Esempio se si accettano 10 bitcoin e successivamente spendo 5, il contrario potrebbe non consistere.

L'*integrity* impone la correttezza dell'origine di transazioni e blocchi, consentendo una protezione nel confronto del double-spending e favorendo la tamper-proofing nella blockchain.

Il teorema CAP viene applicato anche alle blockchain. Per i meccanismi di consenso, liveness e safety hanno una diretta correlazione col teorema CAP:

- La **liveness** garantisce che il processo di consenso completa sempre i suoi round. Anche se non si arriva a consenso, il meccanismo non attende indefinitamente, garantendo la sua availability;
- La **safety** garantisce che i suoi partecipanti sono nello stesso stato dopo un round, garantendo la consistenza nella rete.

Nelle reali implementazioni di soluzioni blockchain, non è mai possibile ottenere sia Consistenza che Availability, perché devono affrontare la **tolleranza alle partizioni**. Il sistema restituirà un errore o un timeout se non è possibile garantire che particolari informazioni siano aggiornate a causa del partizionamento di rete (nodi in errore). Ogni richiesta alla rete riceve una risposta, anche se la rete non può garantire che sia aggiornata a causa del partizionamento di rete (nodi in errore).

Tra le due opzioni rimanenti, i sistemi di tipo **permissionless** (come Bitcoin) in un gruppo aperto di nodi scelgono di privilegiare di garantire l'Availability anziché la Consistency, per poter essere in grado di utilizzare/inviare/ricevere criptovalute. I casi di fork che rappresentano i momenti di inconsistenza sono risolti nel tempo, così da garantire l'Eventual Consistency.

Tra le due opzioni rimanenti, i sistemi di tipo **permissioned** in un gruppo chiuso di nodi scelgono di privilegiare di garantire la Consistency anziché l'Availability, perché non sono focalizzate su criptovalute ma la gestione di dati in ambito distribuito.

Il protocollo di consenso per blockchain si compone di 5 elementi:

1. La **proposta di un blocco**: generazione di blocchi e associazione di prove;
2. La **propagazione di informazioni**: disseminazione di blocchi e transazioni nella rete;
3. La **validazione di blocchi**: controllo dei blocchi per la generazione di prove e verifica della validità delle transazioni;
4. La **finalizzazione dei blocchi**: raggiungimento del consenso sull'accettazione di blocchi validati;
5. Il **meccanismo di incentivazione**: promozione di partecipanti onesti e creazione di token di rete.

STATE MACHINE REPPLICATION:

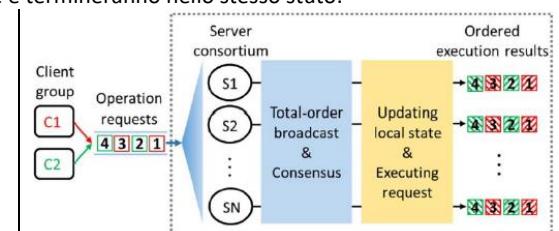
Il consenso su blockchain si ispira al meccanismo di **State Machine Replication (SMR)**. SMR stabilisce i seguenti requisiti:

1. Tutti i server iniziano con lo stesso stato iniziale;
2. Total-order broadcast: tutti i server ricevono la stessa sequenza di richieste secondo l'ordine di generazione dai client;
3. Tutti i server che ricevono la stessa richiesta emetteranno gli stessi risultati di esecuzione e termineranno nello stesso stato.

Un consorzio di N server accetta le richieste di operazioni dai client.

I server confermano il proprio stato prima di raggiungere il consenso ed eseguire le richieste.

SMR è spesso realizzato in maniera leader-based, con un server primario (esempio S1) che riceve le richieste dai client e inizia la procedura di broadcast, mentre gli altri ricevono le stesse richieste e aggiornano il proprio stato locale in modo che corrisponda a quello del leader.



Sussiste una corrispondenza tra gli elementi del consenso nelle blockchain e quelli in SMR:

1. La proposta di blocchi corrisponde alle richieste di operazioni da parte dei client in SMR e il leader che inizia il consenso;
2. La propagazione delle informazioni corrisponde al reliable broadcast delle richieste di operazioni;
3. La validazione dei blocchi corrisponde alla verifica delle firme e l'esecuzione delle operazioni richieste;
4. La finalizzazione dei blocchi corrisponde al raggiungimento del consenso da parte dei server sullo stato corrente;
5. Il meccanismo di incentivo non trova una corrispondenza.

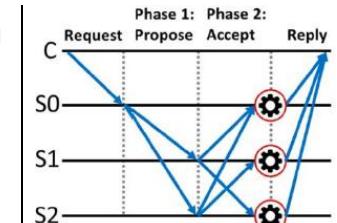
Questo perché SMR presuppone un gruppo ben definito di partecipanti che si presuppongono onesti.

L'**algoritmo di consenso di Paxos** è uno schema SMR progettato per garantire il consenso tollerante a guasti di tipo crash. Un proposer suggerisce un valore all'inizio e lo scopo del sistema è di far sì che gli acceptor concordano su un singolo valore e i learners apprendano tale valore dagli acceptor.

Il client è un learner e il leader tra i server è un proposer, mentre le repliche sono degli acceptors. Il client richiede il consenso su un singolo valore.

Il proposer propaga la richiesta agli acceptors, che si scambiano informazioni sui propri stati. Dopo essersi aggiornati allo stesso stato, tutti i server eseguono la richiesta, che la inviano al client.

Il client riceve i risultati dagli acceptor e formula l'esito a maggioranza. Quando il leader è indisponibile per crash, le repliche ne eleggono uno nuovo.



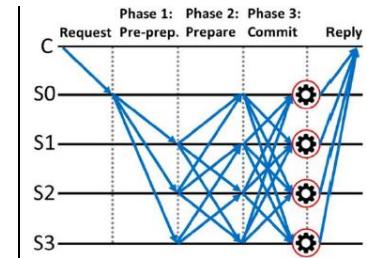
L'algoritmo tollera f crash dei server il cui numero N è maggiore o uguale di $2f + 1$, ma non tollera guasti bizantini perché i nodi ricevono un solo valore, ma se il mittente è bizantino non c'è modo di verificarlo, a meno che non si replica il valore nel tempo per vedere se i valori sono diversi.

PBFT (Practical Byzantine Fault Tolerance):

L'algoritmo di **Practical Byzantine Fault Tolerance (PBFT)** è uno schema SMR che tollera i guasti bizantini ed è diventato sinonimo di tolleranza ai guasti bizantini (BFT) nel contesto delle blockchain. Rispetto a Paxos, nel PBFT si ha una fase in più, nella prima fase (pre-prepare) viene inviato il valore proposto a tutti i nodi, poi gli acceptor hanno una fase di prepare dove scambiano le informazioni ricevute, successivamente una fase di commit in cui scambiano le loro decisioni. La seconda e terza fase consentono di tollerare i guasti bizantini.

Si compone di tre fasi:

1. Tutti i risultati inviati al client devono essere uguali, altrimenti il client decide a maggioranza. Il client invia una richiesta al nodo primario (leader), che la trasmette a tutti i nodi secondari (backup) assegnando un numero di sequenza.
2. I nodi secondari e il leader si accordano sull'ordinamento delle richieste, quando l'ordinamento è stato approvato, la richiesta viene eseguita e un risultato restituito al client.
3. Se il client riceve $f + 1$ risposte identiche, si raggiunge consenso.



Il leader è scelto per ogni round dell'algoritmo o vista. I nodi sono ordinati in base al proprio identificativo, ed indicando il numero della vista con v , il leader è il nodo con identificativo $i = v \bmod N$.

Quando una replica nota un comportamento errato da parte del leader, se ne può richiedere la sostituzione con il meccanismo della view change e l'elezione di un nuovo leader. L'algoritmo consente di tollerare f guasti anche di natura bizantina, con $N \geq 3f + 1$, ovvero i nodi bizantini non riescono a far deviare il consenso raggiunto. La primitiva di comunicazione alla base delle implementazioni PBFT nel contesto delle blockchain è il **gossiping** (Reliable multicast), impiegato per la propagazione di messaggi di blocchi o transazioni.

A differenza della formulazione classica, nelle reti blockchain si ha la terminazione probabilistica: Per ogni nodo onesto, ogni nuovo blocco è sia scartato o accettato nella sua blockchain locale. Un blocco accettato può essere ancora scartato (fork) ma con probabilità decrescente in maniera esponenziale con la crescita della dimensione della catena.

Il principale problema con la PBFT è che richiede che i nodi verifichino la validità dei messaggi degli altri e che il numero di nodi attivi in un dato momento sia sempre noto, e ciò lo rende applicabile in contesti permissioned.

Un altro svantaggio è che i leader vengono sostituiti solo quando le view change vengono attivate dalla rete. L'opportunità di diventare un leader è quindi unfair e mancano pochi incentivi per entrare a far parte della rete. Blockchain elegge i leader in base alla difficoltà del lavoro svolto, che genera incentivi, anche se spreca potenza di calcolo.

La sicurezza di PBFT si basa sul voto in tre fasi con MAC (Message Authentication Code) per la verifica dei messaggi. Sebbene non consumi molte risorse di elaborazione, crea inevitabilmente problemi di scalabilità: in PBFT è impossibile espandersi oltre i 1000 nodi.

PBFT garantisce fortemente il requisito di Safety, infatti un fork è quasi impossibile e viene garantita la terminazione immediata.

Al contrario, la blockchain Nakamoto è più focalizzata sulla liveliness che sulla safety, e quindi i fork si verificano abbastanza frequentemente (consenso multiplo) e affinché un blocco sia sicuro la sua catena deve essere più lunga di un certo numero di blocchi. A tale scopo si è formulato un diverso algoritmo di consenso, quello di Nakamoto. Lo scopo è di evitare il consenso in gruppi chiusi, e di non punire i singoli nodi in un gruppo aperto per essersi comportati in modo malevolo, ma bisogna disincentivare i nodi a replicare comportamenti malevoli.

Il processo di mining dei blocchi (ovvero calcolo della PoW) è stocastico, per cui è impossibile sapere con certezza chi troverà la soluzione, anche se al crescere della difficoltà i nodi capaci di portare avanti il processo diminuiscono. Questo scoraggia tutti gli agenti non disposti ad investire risorse economiche a partecipare al gioco. Con il crescere dell'uso, e quindi del valore, la difficoltà a minare bitcoin aumenta, disincentivando ulteriormente chiunque voglia attaccare la rete. Inoltre, il crescente valore costringe gli agenti onesti ad investire di più nella sicurezza dei propri nodi e, di conseguenza, della rete in generale. Le regole di validazione dei blocchi assicurano che nessun agente onesto accetti blocchi con informazioni scorrette.

CONSENSO NAKAMOTO:

Rispetto al **consenso di Nakamoto** implementato in BitCoin, è possibile trovare un parallelismo con le 5 componenti del consenso nelle blockchain:

1. La **generazione di blocchi** richiede una Proof-of-Work mediante la risoluzione di un puzzle crittografico con un determinato grado di difficoltà tale da mantenere un intervallo di generazione e un grado di protezione adatto;
2. Il **gossiping** viene impiegato per la distribuzione dei blocchi (transazioni appena ricevuto o localmente generati);
3. Un **blocco o transazione** deve essere **validata** prima di essere inviata in broadcast agli altri o collegata alla coda di una catena locale. La validità si realizza evitando la double-spending o controllando la PoW allegata al blocco;
4. La catena più lunga rappresenta il **raggiungimento del consenso** in caso di disaccordo (che ha causato la fork);
5. Chi ha generato un blocco accettato con successo può ottenere un **reward** o premio. Sottomettere una nuova transazione ha un costo monetario.

L'impiego di intensive PoW è necessario per evitare e tollerare attacchi Sybil, a causa della natura permissionless e pseudonima di alcune reti blockchain. Un attaccante può ottenere facilmente delle nuove identità, ma la risoluzione di una PoW implica il consumo di risorse di hash, che può essere difficilmente falsata. Le reward per i blocchi e il costo delle transazioni serve ad incentivare i nodi a partecipare onestamente.

Nelle classiche formulazioni del consenso distribuito, la caratteristica di tolleranza ai guasti è espressa in termini di numero di nodi non corretti che si possono tollerare. Nel caso del consenso di Nakamoto è caratterizzata in termini di percentuale di potenza di hashing avversaria tollerabile:

- Se la rete si sincronizza più velocemente del tasso di proposta di blocco basato su PoW, una maggioranza onesta può garantire il consenso su una parte stabile in continua crescita della blockchain.
- Fintanto che meno del 50% della potenza di hashing totale è controllata in modo malizioso, i blocchi prodotti da miners onesti vengono propagati tempestivamente, la catena principale è della maggioranza onesta che eventualmente supera qualsiasi ramo malizioso.

Nel consenso distribuito classico, il consenso di Nakamoto elude abilmente il vincolo fondamentale di **BFT** pari a 1/3 adottando finalità probabilistiche.

Nel consenso del BFT classico se più di 1/3 della popolazione è malizioso, i nodi onesti finiranno per decidere valori contrastanti, portando al fallimento del consenso. Nel consenso di Nakamoto, tuttavia, le decisioni contrastanti sono consentite temporaneamente sotto forma di fork della blockchain, a condizione che alla fine verranno eliminate dal continuo sforzo della maggioranza onesta.

Il consenso di Nakamoto soffre di alcune limitazioni, primo tra tutti un basso throughput delle transazioni. Si può dimostrare che l'intervallo di 10 minuti tra la generazione di blocchi garantisce che ogni nuovo blocco sia sufficientemente propagato prima che venga inserito nel sistema un nuovo blocco.

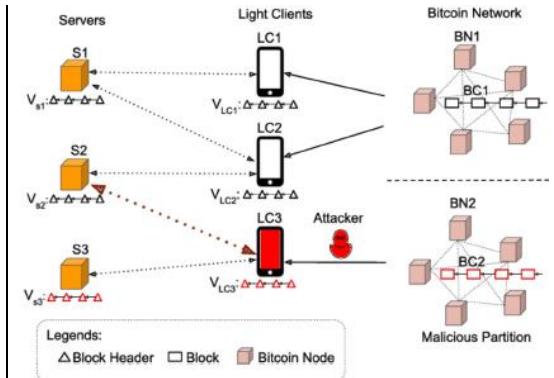
Ridurre l'intervallo tra i blocchi aumenta il throughput delle transazioni, ma lascia i nuovi blocchi non sufficientemente propagati e provoca più incidenti di fork, minando la sicurezza della catena principale. Aumentare la dimensione del blocco (attualmente 1 MB) ha lo stesso effetto, poiché dimensioni maggiori portano a ritardi di trasmissione più elevati e una propagazione insufficiente.

Inoltre, il meccanismo PoW di Nakamoto causa enorme consumo di energia, una transazione Bitcoin in media consuma 431 KWh di elettricità.

Esistono possibili **attacchi al consenso di Nakamoto**:

- Attacco Eclisse:** Se un potente attacco riesce a dominare la comunicazione in entrata/uscita tra un miner vittima e la rete principale, allora la vittima non sarà più in grado di contribuire all'estensione della catena principale. Se il potere di hashing dell'attaccante è α , allora un attacco double-spending è possibile se $\alpha + \varepsilon > 50\%$, con ε percentuali di miners eclissati (limitando quindi il potere degli altri).

L'attacco Eclisse è un exploit della debole connettività di una rete peer-to-peer senza autorizzazione basata su Internet. Per risolvere bisogna aumentare la connettività e la diversità geografica delle connessioni peer-to-peer.



- Selfish Mining:** Se un gruppo di miners maligni trattiene i blocchi appena estratti e li pubblicizza strategicamente per interrompere la propagazione dei blocchi estratti da miner onesti, può parzialmente annullare il lavoro di miner onesti e amplificare il loro potere di mining.

- (1) I blocchi non sono pubblicizzati ma tenuti segreti tra i miners selfish.
- (2) Gli altri miners estendono la catena con blocchi validi. Il miner egoista continua a estendere il suo ramo segreto fino a quando la catena pubblica è un passo indietro. Quindi la pubblica. Poiché la catena segreta è più lunga, le altre parti la considerano la catena principale, quindi ora tutti stanno seguendo i blocchi del minatore egoista. I blocchi generati dagli altri minatori vengono così eliminate.
- (3) Quando forma per la prima volta la sua catena segreta, il minatore egoista corre un rischio. Se ha generato il primo blocco segreto e poi un altro miner ha generato un blocco, non può pubblicare il suo blocco segreto e avere la catena più lunga, si ha una corsa tra due rami di lunghezza uno. Il miner egoista cercherà di estendere il proprio ramo come tutti gli altri miner. Se vince, pubblica la catena più lunga, e l'attacco riparte. Se vincono gli altri, il miner egoista è in svantaggio.

A prima vista potrebbe sembrare che l'attacco non funzioni: il miner di minoranza perderà più gare che vince. Tuttavia, un'attenta analisi mostra che non è così in generale. Un insieme di miner selfish più grande di $1/3$ della potenza di mining aumenterebbe le sue entrate deviando dal protocollo prescritto ed eseguendo Selfish Mining.

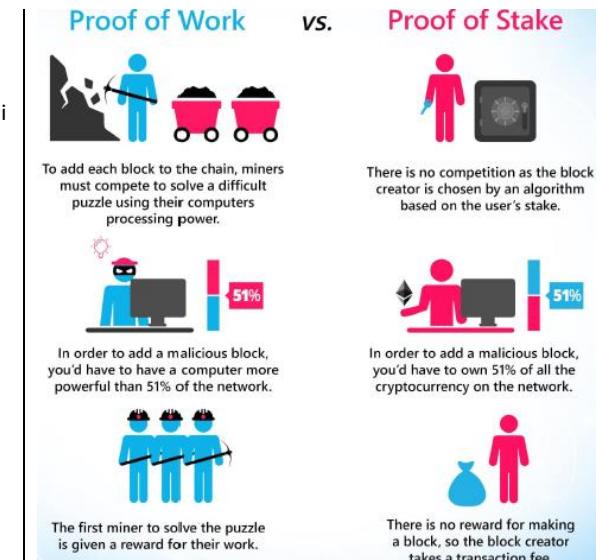
PROOF-OF-STATE:

La **Proof-of-Stake (PoS)** è un'alternativa efficiente dal punto di vista energetico al PoW. Uno Stake o puntata si riferisce alle monete o token posseduti da un partecipante che possono essere investiti nel processo di consenso.

Rispetto a PoW la cui possibilità di proporre un blocco è proporzionale alla sua potenza di calcolo, la possibilità di proporre un blocco per PoS è proporzionale al valore del suo stake.

PoS non si basa sull'hashing dispendioso per generare blocchi, poiché la difficoltà dell'hashing puzzle diminuisce con il valore dello stake del minter, il numero atteso di tentativi di hashing per un minter per risolvere il puzzle può essere significativamente ridotto se il suo valore di stake è alto.

Pertanto, PoS evita la competizione di hashing bruteforce che si verificherebbe se fosse stato usato PoW ottenendo così una significativa riduzione del consumo energetico. Questo implica l'assenza del premio per chi inserisce il blocco giusto nella blockchain, e del mining, in quanto non vengono create nuove unità di criptovaluta con la creazione di ogni blocco. I validatori sono ricompensati con una commissione per le transazioni validate.



Esiste la **Proof of Stake delegato (DPos)**, che consente ai nodi che detengono lo stake maggiore di votare per eleggere i verifieri di blocchi. Questo fa sì che i detentori di stake concedano il diritto di creare blocchi ai delegati che sostengono invece di creare blocchi stessi, riducendo così il loro consumo di potenza computazionale a 0.

PoW avvantaggia i miners che hanno investito maggiormente in hardware, PoS dà una influenza sproporzionata a coloro che posseggono un numero importante di criptovalute, con il rischio di un accentramento di ricchezza nelle mani di pochi, seguendo il dogma secondo cui «Rich people get richer» ("Le persone ricche diventano più ricche").

Un altro problema è "nothing at stake" per il quale nel caso di una fork del network i validatori saranno incentivati ad operare su entrambe le catene, risultando eventualmente in problemi di double-spending, questo problema è meno evidente in un sistema di DPos.

PoS e DPos sono stati applicati nel contesto di classici algoritmi BFT, come PBFT, e di consentirne l'applicazione in contesti open e permissionless.

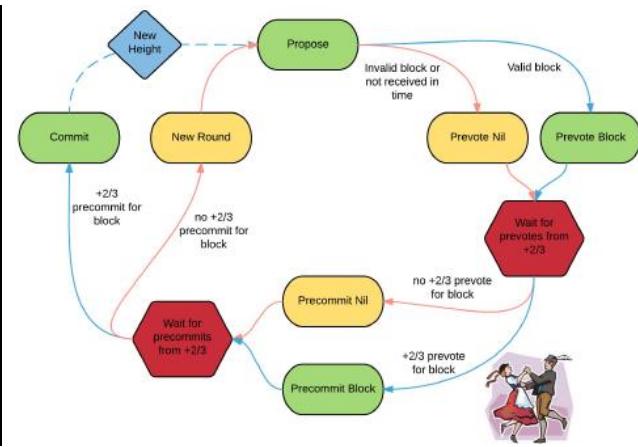
TENDERMINT:

È un algoritmo di consenso ispirato a PBFT che sfrutta la PoS per il consenso in un ambiente permissioned, dove il gruppo dei validatori è un gruppo chiuso e precostituito.

Ogni nuovo blocco viene validato o meno in una iterazione dell'algoritmo, che si compone di molti round per poter giungere a consenso. Si tratta di un consenso di tipo CP, dove la consistenza viene fortemente garantita.

- **[Propose]** Inizialmente, viene scelto a caso un validatoro che propone un nuovo blocco.
- **[Prevote Nil] [Prevote Block]** Il nuovo blocco viene distribuito agli altri validatori che ne verificano la validità, e ne votano l'accettazione.
- **[Precommit Nil] [Precommit Block]** Se sono arrivati i 2/3 ed oltre voti allora si manifesta l'intenzione di accettare il blocco, altrimenti di rigettarlo.
- **[Commit] [New Round]** Se sono arrivati i 2/3 dei voti ed oltre allora si conferma la decisione e si passa ad un'altra iterazione con un nuovo validatore, altrimenti si realizza un nuovo round.
- **[Wait for prevotes from +2/3]** È un algoritmo parzialmente sincrono, quindi si smette di aspettare dopo che è trascorso un timeout.

Non tutti i validatori avranno lo stesso "peso", ma la PoS viene usata per dare peso maggiore a chi ha uno stake maggiore. La condizione di 2/3 non è sul numero di votanti ma sulla quantità di criptovaluta totale nel sistema.



4. PROGRAMMAZIONE SMART CONTRACTS

Un contratto è un accordo legalmente vincolante che riconosce e disciplina i diritti e i doveri delle parti del contratto.

Uno **smart contract** è la “trasposizione” in codice di un contratto, mediante funzioni “if/then” incorporate in software o protocolli informatici, per verificare in automatico l’avverarsi di determinate condizioni e di autoeseguire azioni quando le condizioni sono raggiunte e verificate.

È un programma deterministico che elabora le informazioni in una blockchain, a parità di input i risultati restituiti saranno identici (garantendo la consistenza). Ciò garantisce alle parti una assoluta “certezza di giudizio oggettivo” escludendo qualsiasi forma di interpretazione.

Uno smart contract per blockchain deve soddisfare i seguenti obiettivi:

- **Osservabilità**, l’output deve essere legato sempre all’input, ovvero non bisogna avere dati nascosti e che i risultati vengano tutti restituiti come output (nessuna variabile globale o statica);
- **Verificabilità**, si deve avere la possibilità di testare il software;
- **Riservatezza**, non si devono esporre dati riservati;
- **Applicabilità**, uno smart contract deve essere progettato in maniera tale che è applicabile alla blockchain.

Su una blockchain, uno smart contract non può essere modificato, in quanto è visto come un blocco della blockchain e quindi non mutabile, ma può essere facilmente osservato, verificato, auto-applicato e, a seconda della modalità di accesso della blockchain, è possibile ottenere la privacy.

Ecco un processo graduale di vita di uno smart contract:

1. Gli sviluppatori scrivono la logica per lo smart contract in un linguaggio di programmazione supportato dalla piattaforma blockchain che desiderano utilizzare. Utilizzando un compilatore specifico (di solito fornito dalla piattaforma blockchain), compilano il codice sorgente del loro smart contract e ottengono una sua rappresentazione in bytecode;
2. Dopo aver ottenuto il codice byte, lo smart contract è pubblicato sulla piattaforma blockchain ed archiviato. Una volta pubblicato lo smart contract, sarà di sola lettura o modificabile. Nel caso in cui sia di sola lettura, per fornire un aggiornamento, gli sviluppatori dovranno pubblicare una nuova versione dello smart contract e reindirizzare gli utenti ad essa. Una volta caricato, lo smart contract è al suo stato iniziale, pari ai valori iniziali delle variabili interne;
3. L’accesso a un programma di smart contract pubblicato dipende dalla piattaforma blockchain, come un indirizzo restituito quando lo smart contract è caricato nella piattaforma. Tale indirizzo può essere utilizzato per interagire con lo smart contract, inviando le transazioni contenenti la funzione che desiderano utilizzare e gli argomenti della funzione. Se è necessaria una quantità di valuta della piattaforma per avviare l’esecuzione della funzione, tale importo sarà trasferito insieme alla transazione. La transazione verrà archiviata nel pool di transazioni della piattaforma blockchain che attendono di essere eseguite e convalidate;
4. La piattaforma blockchain selezionerà le transazioni da eseguire e convalidare. Durante l’esecuzione, le funzioni nella transazione verranno eseguite dai nodi. Durante la validazione, i nodi confronteranno i propri risultati e selezioneranno quello da mantenere secondo un protocollo di consenso;
5. Una volta selezionato il risultato valido, verrà inserito in un blocco da aggiungere alla blockchain. Se una transazione validata ha alterato le variabili interne di uno smart contract, i nuovi valori saranno considerati come valori iniziali da transazioni future sullo smart contract.

L’implementazione di smart contract non è standardizzata e ogni piattaforma blockchain propone una propria soluzione.

Le funzioni di uno smart contract possono essere chiamate direttamente dai client o indirettamente da altri smart contracts. Per garantire che le chiamate terminino, al client viene addebitata una fee ad ogni chiamata e sua durata. Se l’addebito supera quello che il cliente è disposto a pagare, il calcolo viene interrotto e le operazioni annullate.

L’attuale approccio di sviluppo di smart contract limita il throughput perché non ammettono concorrenza, infatti:

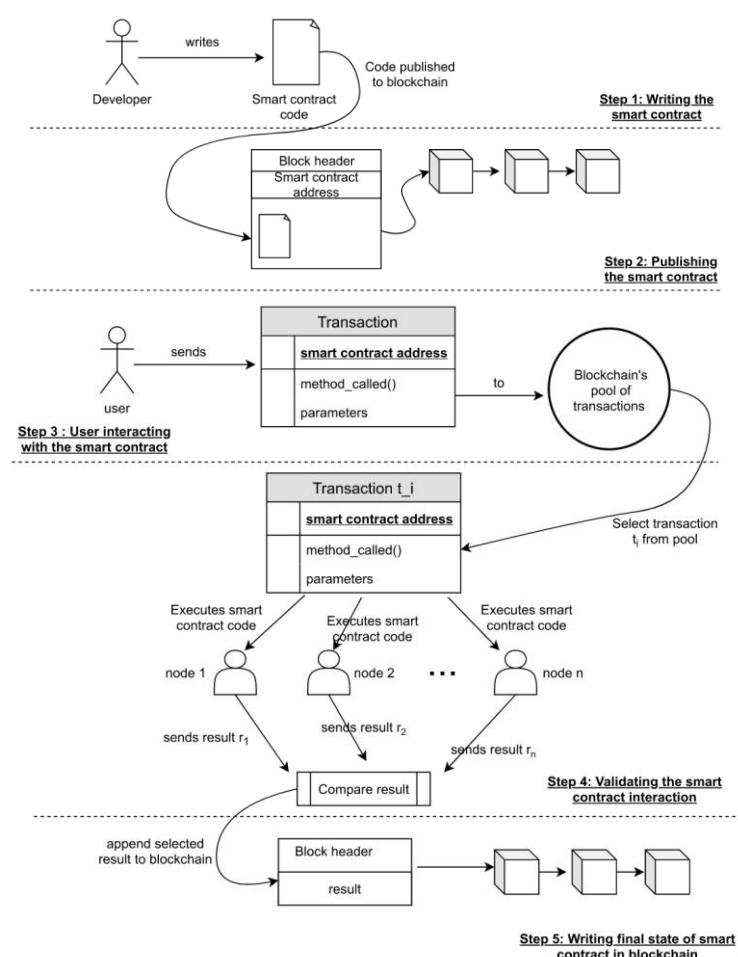
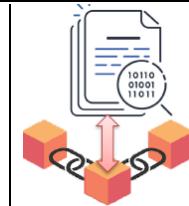
- Quando un miner crea un blocco, assembla una sequenza di transazioni e calcola un nuovo stato provvisorio eseguendo gli smart contract di tali transazioni in serie, nell’ordine in cui si verificano nel blocco.
- Un miner non può eseguire gli smart contracts in parallelo, perché potrebbero sussistere degli accessi in conflitto a dati condivisi e un interleaving arbitrario potrebbe produrre uno stato finale non coerente. Spesso per gli smart contract, non è possibile dire in anticipo se le esecuzioni di contratti siano in conflitto.

4.1 PRINCIPALI PIATTAFORME

La blockchain della criptovaluta **Bitcoin** contiene blocchi di transazioni codificate secondo il modello UTXO (Unspent Transaction Output).

Ogni transazione ha dei Bitcoin in ingresso e in uscita. In ingresso, si ha l’indirizzo delle transazioni mai usate in precedenti transazioni e la cui somma degli output equivale l’output della transazione che le contiene. I destinatari di una transazione sono identificati da un **indirizzo Bitcoin**, un identificatore di 26-35 caratteri. Gli indirizzi possono essere generati gratuitamente da qualsiasi utente.

Spesso è difficile avere transazioni che sommate risultano esattamente pari alla quantità da trasferire. L’eccedenza deve essere trasferita all’autore della transazione.



e163a9244cd9f14098047502169334ad9337ca71039e0f94fb0d8a939e7447

10wkrR59ufUaPAVANACKTVJLk5hChpX0xE

2018-12-04 16:13:58

34E9e81PPNnchkaXWVf0cnd0P8anVnPgEq

0.5 BTC

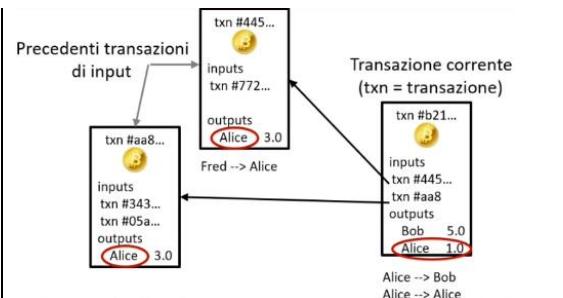
1MD5wKX0XhCwpxu8T0X1Bqz4GAn3zPht5

2.39998054 BTC

3.49998054 BTC

Ogni transazione contenuta nel wallet ha dei Bitcoin in input e in output. In input si ha l'indirizzo delle transazioni mai usate in precedenza e la cui somma degli output equivale all'output della transazione che lo contiene.

Ad esempio, se si hanno due transazioni aventi un identificativo (Alice) ed un output (3.0 o 2.0), se Alice deve inviare soldi a Bob gli invia transazioni che hanno come destinatario Alice (ovvero le precedenti transazioni in input) e la cui somma è maggiore o uguale ai soldi richiesti. Se rimane della crypto-valuta, nel caso che la somma da inviare è maggiore, Bob riceve la quota richiesta e il rimanente Alice viene segnata come secondo destinatario, ovvero Alice da a sé stessa 1.0.



Il modello UTXO non permette di ritrovare sulla blockchain il saldo corrente di un determinato utente, che deve essere ricostruito raccogliendo tutte le transazioni che presentano un determinato utente in output e quelle con tali transazioni in input. UTXO consente transazioni parallele perché non esiste alcun account, inoltre consente l'anonymato dal momento che un utente può disporre di multipli indirizzi BitCoin. Essendo **stateless** rende complesso la realizzazione di applicazioni come smart contracts, in quanto, ad esempio il saldo di un utente non può essere salvato.

La **natura stateless** della rete consente di ottenere resilienza ed elasticità, oltre che la possibilità per qualsiasi istanza di eseguire qualsiasi attività.

Gli utenti hanno molti indirizzi Bitcoin, che fungono da pseudonimi per garantire la privacy, e si vuole impiegare un indirizzo per ogni transazione:

- Un modo ingenuo per accettare pagamenti in Bitcoin è dire ai propri clienti di inviare denaro a un determinato indirizzo (mancanza di identità);
- Essendo le transazioni Bitcoin pubbliche, se un cliente Alice invia Bitcoin, un agente dannoso Bob potrebbe vedere tale transazione e inviare un'email affermando che è stato l'autore del pagamento;
- Il destinatario del pagamento non è in grado di distinguere se il trasferimento è stato effettivamente svolto da Bob o Alice;
- Per questo motivo si usa indicare un indirizzo per ogni cliente che deve effettuare un pagamento.

È possibile impiegare una transazione conoscendo il suo identificativo, che è pubblico. Sorge il problema di evitare che utenti maliziosi impieghino transazioni che non sono proprie, e per formulare dei meccanismi di controllo di autenticazione della legittimità di utilizzo, è stato realizzato un sistema di scripting su BitCoin, precursore degli smart contract. **Script** è un programma imperativo, basato su stack ed elaborato da sinistra a destra:

- Uno script è essenzialmente un elenco di istruzioni registrate con ogni transazione che descrivono come la prossima persona che desidera spendere i Bitcoin trasferiti può accedervi;
- Una transazione è valida se nulla genera un errore e l'elemento in cima allo stack è True (diverso da zero) alla fine dello script;
- Gli stack contengono vettori di byte.

Il linguaggio di scripting ha due tipi di istruzioni:

- Le **istruzioni di dati** contengono semplicemente un valore e sono racchiuse tra parentesi angolari (cioè <data>);
- Gli **OP_CODE** sono operazioni specifiche appartenenti al linguaggio Bitcoin Scripting che agiscono sul valore in cima allo stack e pone il loro risultato anche in cima allo stack.

Lo script si compone di due parti, una parte iniziale, che contiene l'ID della transazione (è una chiave) e la parte finale dove ci sono altre informazioni.



Il funzionamento dello script per la transazione precedente è il seguente:

1. Mette nello stack le prime due parti, ovvero la transazione e il dato:
2. Effettua il duplaco (DUP) della chiave:
3. Esegue l'HASH160 della chiave che prende e la inserisce nello stack:
4. Effettua il confronto tra le due chiavi:
5. Esegue il controllo della firma risolvendo così l'accesso:

Altri esempi sono i seguenti:

Per spendere questo output, è necessario fornire due numeri la cui somma è 8.



Per spendere questo output, vanno fornite due diverse stringhe di dati che producono lo stesso risultato hash (trovare una collisione).



Per spendere questo output, va dato qualcosa che ha lo stesso hash di ciò che è all'interno dello script di blocco.



Ogni codice operativo ha una corrispondente rappresentazione esadecimale, che viene usata per fornire la sua codifica.

Lo script ha un campo "scriptPubKey" che contiene la parte iniziale dello scripting, ovvero le operazioni da effettuare.

Uno **locking script** viene posizionato su ogni output creato in una transazione per determinare come la transazione deve essere usata. Mentre è necessario fornire uno **unlocking script** per ogni input che si desidera spendere in una transazione.

```
"vout": [
  {
    "value": 0.10000000,
    "n": 0,
    "scriptPubKey": {
      "asm": "OP_DUP OP_HASH160",
      "hex": "76a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_EQUALVERIFY OP_CHECKSIG",
      "reqSigs": 1,
      "type": "pubkeyhash",
      "addresses": [
        "1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK"
      ]
    }
  ]
]
```

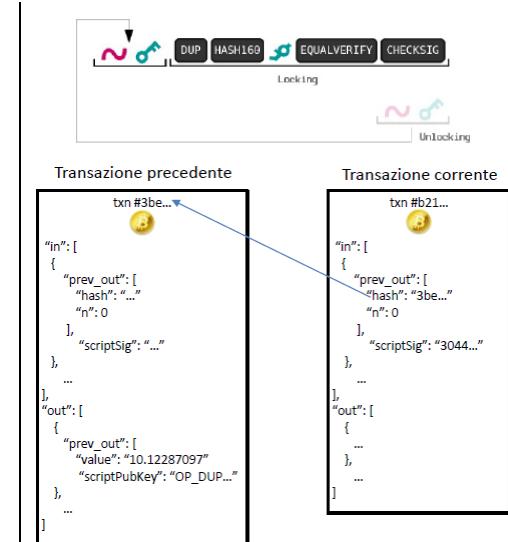


Quindi nella *transazione precedente* nel campo "out" e in "scriptPubKey" (la parte finale della transazione, ovvero Locking) è presente lo script di unlocking. Nella *transazione corrente* che intende usare la precedente, nel campo "scriptSig" (la prima parte della transazione, ovvero Unlocking) si mettono i dati su cui lavora lo script che sta in output.

La parte Locking + Unlocking viene eseguito dal validatore per determinare che la transazione può essere utilizzata.

Anche se l'unlocking script è fornito dopo del locking script iniziale, in realtà viene messo per primo quando si eseguono entrambi gli script.

Il linguaggio non è Turing-completo, in quanto non ci sono istruzioni condizionali e cicli, perché si vuole predicitività dei tempi di esecuzione, che dipende deterministicamente dal numero di istruzioni in esse contenute. Inoltre, è senza stato, non hanno informazione della valuta sbloccata, o non accede alle informazioni del blocco che li contiene.



4.1.1 ETHEREUM

Ethereum è la più grande piattaforma software decentralizzata ed open-source che consente lo sviluppo di smart contracts e applicazioni implementate al di sopra di una Blockchain permissionless. La blockchain crea un **world state**, ovvero uno stato distribuito tra tutti i nodi. Ethereum è una rete blockchain (con PoW è in transizione verso PoS) ed ha una criptovaluta come BitCoin, chiamata **Ether**. Usato per pagare **gas**, un'unità di calcolo utilizzata nelle transazioni e in altre transizioni di stato. Pertanto, ogni operazione che viene fatta sulla blockchain ha un costo, questo perché l'algoritmo di consenso basato sulla PoW deve dare delle reward, quando si eseguono smart contract la blockchain cresce e il **gas** decresce.

L'algoritmo che viene utilizzato per la PoW è leggermente diverso da quello canonico e viene chiamato **Ethash**, che utilizza un algoritmo di hash appartenente alla famiglia Keccak, la stessa delle funzioni hash SHA-3. Utilizza un set di dati di grandi dimensioni che viene periodicamente rigenerato e cresce lentamente nel tempo. Svolge una serie di intensive letture ad accesso casuale in memoria al set di dati di 2 GB strutturato come un grafo aciclico diretto (DAG). La risoluzione si realizza con 64 cicli a partire dall'hash di informazioni di partenza con pagine estratte dal DAG. Il digest viene confrontato con una soglia target. Se inferiore o uguale, il calcolo è considerato riuscito. In caso contrario, l'algoritmo viene rieseguito con un nonce diverso (incrementando quello corrente o scegliendone uno nuovo a caso).

A differenza di Bitcoin dove si aveva una anonimicità e le transazioni erano da un indirizzo all'altro, in Ethereum si ha il concetto di **accounts**, che sono delle identità sulla blockchain dei suoi utilizzatori che interagiscono tra loro tramite transazioni (messaggi) e sono caratterizzati da un indirizzo a 20 byte che li identifica e viene memorizzato come stato all'interno della blockchain.

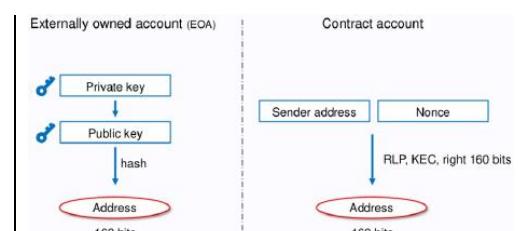
Ethereum gestisce due diversi tipi di account:

- **Externally owned accounts (EOA)** (*utenti umani*), sono in grado di inviare e ricevere Ether, e inviare transazioni agli Smart Contract;
- **Contract accounts** (*Smart Contract veri e propri*), che oltre alle funzionalità degli EOAs hanno del codice associato che implementano delle azioni "triggerate" da EOA o altri Contract accounts.

L'esecuzione del codice modifica le informazioni contenute nel proprio spazio di archiviazione, consentendo di utilizzare la Blockchain per scopi diversi dalla criptovalute.

address	balance	nonce
---------	---------	-------

- **Address:** Gli indirizzi Ethereum contengono 40 cifre esadecimale. Gli indirizzi EOAs incominciano con il prefisso "0x" seguito dai 20 byte più a destra dell'hash della chiave pubblica. Gli indirizzi di contratto sono nello stesso formato, ma sono determinati dal mittente e dal nonce pari al numero di transazione inviato dal mittente.
- **Balance:** Ethereum non si basa sugli output di transazione non spesi (UTXO), ma gli account hanno uno stato che indica il saldo corrente. Lo stato non è memorizzato sulla blockchain, ma in un albero separato di Merkle Patricia. Un wallet di criptovaluta memorizza le chiavi pubbliche e private (un utente può avere più indirizzi), che possono essere utilizzati per ricevere o spendere ether. I wallet sono come applicazioni che consentono di interagire con un account Ethereum, analogamente alle app di e-banking e un conto bancario.
- **Nonce:** Numero di transazioni totali.



I contratti hanno generalmente 4 scopi:

1. Gestire un "data store" che contiene informazioni utili (variabili o dati) per altri contratti o per il mondo esterno;
 2. Comportarsi come un EOA con politiche di accesso più avanzate (una sorta di filtro che consente l'inoltro di "messaggi" a determinate EOA solo se determinate condizioni vengono soddisfatte);
 3. Gestire un contratto o una relazione in corso tra più utenti (un esempio è un contratto che paga automaticamente chi invia una soluzione valida ad alcuni problemi matematici, o dimostra che sta fornendo una risorsa computazionale);
 4. Fornire funzionalità ad altri contratti (una sorta di libreria).

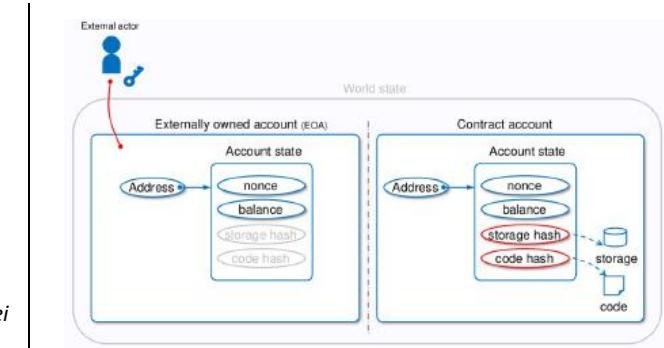
Il codice viene interpretato dalla **Ethereum Virtual Machine (EVM)**, è rappresentato in un EVM bytecode.

Gli **account dei contratti** sono diversi:

	Account Personale	Account di Contratto
address	H(pub_key)	H(addr + nonce del creatore)
code	Ø	Codice
storage	Ø	Dati
balance	Saldo ETH (in Wei)	Saldo ETH (in Wei)
nonce	# transazioni inviate	# transazioni inviate

**H è la funzione hash;*

*Wei è la più piccola unità di ETH, e sono necessari 10^{18} wei per un ETH. 10^9 wei corrispondono a un gwei.



Invece, una **transazione** ha la seguente struttura dati:



- **From**, indirizzo dell'utente di origine della transazione (mittente);
 - **Signature**, firma della nuova transazione usando la chiave privata dell'utente creatore;
 - **To**, indirizzo dell'utente di destinazione della transazione;
 - **Amount**, quantità di ETH trasferita.

Terminale Ethereum mediante cui chiamare funzioni della classe principale della libreria web3.js.

- Web3.fromWei converte in Ether un valore espresso in Wei;
 - Web3.toWei opera l'inverso;
 - La transazione ha richiesto una **commissione** (in gas), per cui il valore associato all'account[0] è inferiore a 90 Ether.

L'avvenuta transazione genera un codice di hash come ricevuta

```
> web3.fromWei(eth.getBalance(eth.accounts[0]))
100
> web3.fromWei(eth.getBalance(eth.accounts[1]))
100
> eth.sendTransaction({
.....  from: eth.accounts[0],
.....  to: eth.accounts[1],
.....  value: web3.toWei(10)
..... })
"0x497913c178f65613035b22340fcf5bc59c7ed474bfa3c1e798c6dffbeda9da5b"
>
> web3.fromWei(eth.getBalance(eth.accounts[0]))
89.99958
> web3.fromWei(eth.getBalance(eth.accounts[1]))
110
```

Nel caso di **transazioni per smart contracts** la struttura cambia per contenere anche i dati per il contratto:



- **Data**, serve per encapsulare i parametri di chiamata della funzionalità dello smart contract e anche a capire qual è la funzionalità invocata.

CICLO DI VITA DI SMART CONTRACT IN ETHEREUM:

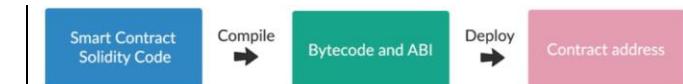
Il **ciclo di vita degli smart contract** si compone di 3 stati:



-FASE DI CREATE:

Nella fase di ***create***, un nuovo contratto viene scritto e caricato sulla rete da parte di un utente. In questa fase, la transazione ha il parametro “to” vuoto in quanto lo smart contract non esiste ancora, mentre il parametro “data” contiene il codice (Bytecode) del contratto.

Gli smart contracts sono scritti in linguaggi di programmazione di alto livello (Solidity), tale codice viene compilato dalla EVM e deployato nella Blockchain sottoforma di EVM Bytecode.



Successivamente, una volta che il contratto viene scritto viene compilato in un Bytecode (contenuto nel campo "data"):

```
pragma solidity ^0.4.0;

contract SimpleStorage {
    uint storedData;

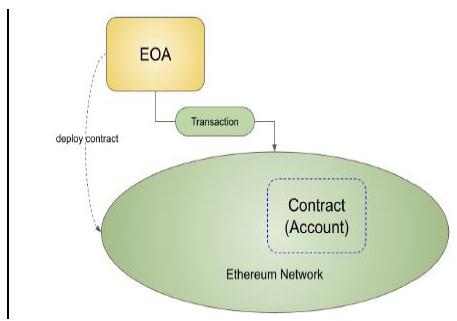
    function set(uint x) public {
        storedData = x;
    }

    function get() constant public returns (uint retVal) {
        return storedData;
    }
}
```

2

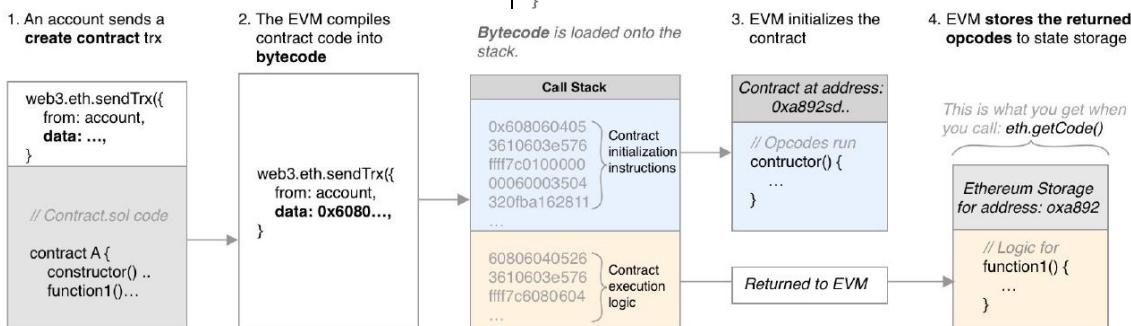
"6808060405234801561001057600080fd5b5060df8061001f6000396000f30060806
04052600436106049576000357c01000
00000000000000000000463fffff761806360fe47b114604e5780636d4ce63c146
078575b600080fd5b34801560957600080fd5b506076600480360381019080080359
0602001909219050505060a56b50b348015608357600080fd5b50608a605a5
b6040518082815260200191505060405180910390f35b80600081905550565b600
080549050905600a165627a7a7320582080122bb351e6e2c021f1c56c0c5933087e7
62eafe73a336b793cbef5a8f10029

Quando viene effettuata la transazione di deploy, viene creato un nuovo account di tipo smart contract che avrà un suo indirizzo dato dall'hash dell'indirizzo + nonce di chi ha creato.



L'EOA crea un contratto con una transazione contenente il relativo bytecode e somministrando una determinata quantità di gas. L'esecuzione della transazione restituisce un codice di hash.

Eseguendo `getTransactionReceipt(hash_value)` si ottiene un "ricevuta", con l'indirizzo del contratto e l'informazione sul gas speso. In ingresso alla transazione è stato inserito un valore di gas pari a 200000, ma poiché il "gaslimit" è pari a 90000, la parte non spesa viene restituita all'EOA.



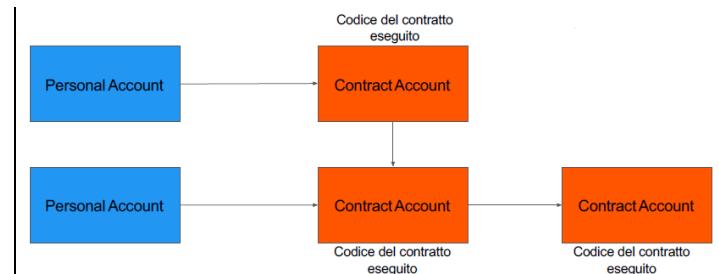
1. Un EOA o un contratto invia una transazione contenente dati, ma nessun indirizzo di destinatario. Questo formato indica all'EVM che è una creazione di contratto, non una normale transazione di invio / chiamata;
 2. EVM compila il codice del contratto in Solidity ottenendo il bytecode, che traduce direttamente in opcode il codice del contratto, che vengono eseguiti in un singolo stack di chiamate;
 3. Durante la creazione del contratto, EVM esegue solo il codice di inizializzazione fino a quando non raggiunge la prima istruzione STOP o RETURN nello stack, il contenuto oltre tale punto è ritornato alla EVM e rappresenta le funzioni che si possono invocare. Durante questa fase, viene assegnato un indirizzo al contratto;
 4. I codici operativi delle altre funzioni vengono copiati in memoria.

-FASE DI INTERACT:

Nella fase di ***interact***, un contratto esistente è attivato, esegue il suo codice, legge/scrive nella memoria interna ed invia transazioni o chiama altri contratti. Un contratto non può avviare nuove transazioni se non in risposta a delle transazioni ricevute.

In questa fase, nella transazione il parametro “to” contiene l’indirizzo del contratto da invocare, mentre il parametro “data” contiene il metodo del contratto da chiamare ed eventuali argomenti.

Possono sussistere delle relazioni tra contratti, con transazioni inviati tra i contratti come effetto dell'esecuzione di codice.



Per poter accedere alle funzioni implementate nel precedente contratto è necessario ottenere la codifica che le è stata assegnata nel bytecode.

Tale codifica è realizzata applicando una funzione di hash alla firma del metodo.

Per ottenere tali valori è necessario considerare i primi 4 bytes dei valori ottenuti dall'hash delle funzioni:

```
> web3.sha3('get()')
"0x6d4ce63caa65600744ac797760506da39ebd16e8240936b51f53368ef9e0e01f"
> web3.sha3('set(uint256)')
"0x6fe47b16ed402aae66ca03d2bfc51478ee897c26a1158669c7058d5f24898f4"
```

Effettuando una call si verifica che il valore contenuto nella variabile storedData è 0.

Viene caricato un nuovo valore tramite la sendTransaction.

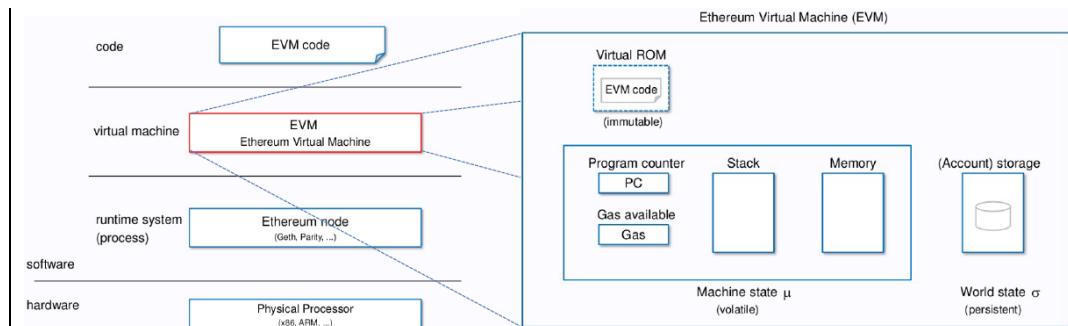
Rieffettuando la call si ottiene il valore aggiornato.

Le call sono transazioni eseguite direttamente dalla VM e non riportate nella Blockchain.

-FASE DI DESTROY:

L'ultima fase è la ***destroy***, dove un contratto esistente viene rimosso (in realtà viene invalidato dato che le informazioni in un a blockchain non possono essere modificate) dalla rete da parte di un utente. In questa fase, nella transazione il parametro “to” contiene l’indirizzo del contratto da invocare, mentre il parametro “data” contiene il nome del metodo di distruzione.

La EVM è un ambiente di esecuzione isolata (sandbox) dei contratti con uno stack, un parallelismo interno a 32-byte, e codici operativi che formano una macchina quasi Turing-completa.



SOLIDITY (ETHEREUM):

Solidity è un linguaggio di programmazione con tipizzazione statica ed orientato agli oggetti per la scrittura di applicazioni che implementano la logica di business incorporata in smart contract su Ethereum e altre piattaforme blockchain. È progettato attorno alla sintassi ECMAScript ma si differenzia per una tipizzazione statica e tipi di ritorno variadici. Supporta inoltre tipi complessi definiti dall'utente, ad esempio struct ed enumerazioni, che consentono di raggruppare i tipi di dati correlati. I contratti supportano l'ereditarietà, inclusa l'ereditarietà multipla con linearizzazione C3. È stata inoltre introdotta un'interfaccia binaria dell'applicazione (ABI) che facilita più funzioni type-safe all'interno di un singolo contratto.

Per **tipizzazione statica** si intende che il controllo del tipo delle strutture dati membro di contratti avviene in fase di compilazione, non in fase di esecuzione come per i linguaggi tipizzati in modo dinamico.

ECMAScript è la specifica tecnica di un linguaggio di scripting, destinato a garantire l'interoperabilità delle pagine Web tra diversi browser Web. L'implementazione più conosciuta di questo linguaggio è JavaScript.

Un **parametro di ritorno variadico** accetta zero o più valori di un tipo specificato, e la funzione restituisce una quantità variabile di valori.

La **linearizzazione C3** è un algoritmo utilizzato per ottenere l'ordine in cui i metodi devono essere ereditati in presenza di ereditarietà multipla. La linearizzazione è la somma della classe più un merge delle linearizzazioni dei suoi genitori e un elenco dei genitori stessi. Il merge viene eseguito selezionando la prima intestazione delle liste che non appare nella coda di nessun'altra. L'elemento selezionato viene rimosso e aggiunto all'elenco di output. Se non è possibile selezionare un'intestazione valida, allora l'unione è impossibile da calcolare a causa di ordinamenti incoerenti delle dipendenze e nessuna linearizzazione è possibile.

Un **ABI** è un'interfaccia tra due moduli di programma binario, e definisce come si accede alle strutture dati o alle routine nel codice macchina. È espresso in un formato di basso livello, dipendente dall'hardware. Al contrario, un'API definisce questo accesso nel codice sorgente, in formato di livello relativamente alto, indipendente dall'hardware.

È possibile scrivere applicazioni in Solidity mediante l'IDE Remix.

Uno smart contract scritto in Solidity inizia sempre con “*pragma*” che specifica la versione di Solidity. Il contratto inizia con la parola chiave “*contract*” più il nome del contratto (proprio come una classe), per poi definire al suo interno la logica del contratto. In Solidity i qualificatori di accesso vanno messi dopo il nome della funzione.

Quando una funzione deve ritornare qualcosa, nel nome della chiamata va specificato cosa ritorna col tipo di dato.

La variabile “*amount*” è una variabile interna al contratto, per tanto non va nella blockchain.

La parola chiave “*view*” significa che la funzione non modificherà lo stato del contratto, rende visibile il dato e quindi ha un ritorno.

```
testContract.sol
1 pragma solidity ^0.5.1;
2
3 contract testContract {
4
5     uint amount=13;
6
7     function set(uint _n) public {
8         amount = _n;
9     }
10
11     function get() view public returns (uint) {
12         return amount;
13     }
14 }
```

Per depolare lo smart contract:

The screenshot shows the Remix IDE interface divided into three main sections: SOLIDITY COMPILER, DEPLOY & RUN TRANSACTIONS, and another DEPLOY & RUN TRANSACTIONS section.

- SOLIDITY COMPILER:** Shows compiler version 0.5.11+commit.c082d0b, language Solidity, and EVM Version compiler default. A button to "Compile testContract.sol" is present.
- DEPLOY & RUN TRANSACTIONS (Left):** Shows Environment as JavaScript VM, Account as 0xCA3...a733c (100 eth), Gas limit at 3000000, and Value at 0 wei. A large blue arrow points from here to the Deploy button in the middle section.
- DEPLOY & RUN TRANSACTIONS (Right):** Shows Environment as JavaScript VM, Account as 0xCA3...a733c (99.999), Gas limit at 3000000, and Value at 0 wei. It also shows a dropdown for "testContract - browser/testContract" and a "Deploy" button.
- Middle Section:** Contains a "Deploy" button and an "At Address" input field with "Load contract from Address". Below it is a "Transactions recorded: 0" section and a "Deployed Contracts" section which currently displays no contracts.
- Bottom Section:** Shows ABI and Bytecode download links.

Secondo esempio di smart contract:

In questo contratto è presente la variabile di tipo “*address payable issuer*”, che memorizza un indirizzo che può ricevere delle transazioni di trasferimento di criptovaluta.

Successivamente è presente un costruttore, che in fase di creazione inizializza lo smart contract e la parte funzionale viene memorizzata.

In tal caso prende l'indirizzo di chi ha inviato la transazione di creazione, memorizzato in “*msg.sender*”. Solidity creerà una variabile di sistema che è il messaggio di transazione.

Nella funzione *getBalance()* è presente “*address*” che è una funzione di sistema che conosce l'indirizzo di qualcosa. Quindi, “*address(this)*” darà l'indirizzo dello smart contract, col “*.balance*” restituisce il saldo dello smart contract.

La funzione *withdrawFunds()* conferma chi ha creato lo smart contract e chi ha invocato la funzione, se corrispondono al possessore dello smart contract viene fatto il trasferimento di criptovaluta con “*issuer.transfer(funds)*” (ricordando che *issuer* è di tipo *payable*).

La funzione *receiveFunds()* è anche essa *payable*, significa che quando si invoca questa funzione, lo smart contract riceve dei fondi, non ha logica perché serve solo a caricare il saldo dello smart contract del valore inserito nella transazione.

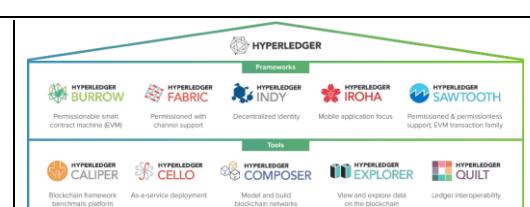
In totale, questo smart contract funge da salvadanaio, ovvero il creatore invia dei fondi che si accumulano nello smart contract, quando poi sono necessari all'*issuer*, invoca la funzione *withdrawFunds()* per prelevare saldo.

```
testContract.sol | financialContract.sol
1 pragma solidity ^0.5.1;
2
3 contract financialContract {
4
5     address payable issuer;
6
7     constructor() public{
8         issuer = msg.sender;
9     }
10
11     function receiveFunds() payable public{
12     }
13
14     function getBalance() view public returns (uint){
15         return address(this).balance;
16     }
17
18     function withdrawFunds(uint funds) public{
19         assert(issuer == msg.sender);
20         issuer.transfer(funds);
21     }
22
23 }
24 }
```

4.1.2 HYPERLEDGER FABRIC

Hyperledger è una famiglia di progetti di blockchain open source avviato nel 2015 dalla Fondazione Linux, per supportare lo sviluppo collaborativo di soluzioni DLT (Distributed Ledger Technology).

Si suddivide in framework per la realizzazione di piattaforme blockchain, e strumenti (tools) a supporto di applicazioni e smart contracts.



Hyperledger Fabric è un progetto di IBM per imprese, e consiste in una piattaforma di DLT **permissioned**, con alcune differenze chiave rispetto ad altri popolari soluzioni:

- ha un'architettura altamente modulare e configurabile;
- supporta smart contracts (detto *Chaincode*) scritti in linguaggi di programmazione comuni, piuttosto che utilizzare linguaggi specifici del dominio;
- i partecipanti sono noti, invece che essere anonimi, adottando un modello di governance costruito sulla fiducia che c'è tra i partecipanti;
- supporto per *protocolli di consenso* innestabili, da scegliere sulla base delle esigenze, e senza una criptovaluta nativa per incentivare costosi mining o per alimentare l'esecuzione di contratti intelligenti.

Le **organizzazioni** che prendono parte alla costruzione della rete Hyperledger Fabric sono chiamate «**membri**», ognuna responsabile di impostare i propri peer per la partecipazione alla rete. Questi **peer** devono essere configurati con materiali crittografici appropriati per autenticare la propria **identità** o proteggere i canali di comunicazione (con SSL/TLS).

- I peer ricevono richieste di transazioni dai client mediante un SDK o servizi Web REST per interagire con la rete Hyperledger Fabric, e sono connessi tra loro da canali che consentono l'isolamento dei dati e la riservatezza.
- Tutti i peer mantengono il loro registro unico per canale a cui sono iscritti, cioè solo se il peer è registrato a quel canale riceve i blocchi pubblicati su quel canale e ogni canale ha un suo registro (ledger). A differenza di Ethereum, i peer hanno ruoli diversi:
 1. **Endorser peer**: ricevuta la richiesta da un client, convalida la transazione ed (eventualmente) esegue il Chaincode simulando l'esito della transazione ma senza aggiornare la blockchain, cioè solo il nodo Endorser esegue il Chaincode per vedere cosa succederebbe se eseguito, quindi non è necessario installarlo in ogni nodo della rete. Al termine, l'Endorser può approvare o disapprovare la transazione.
 2. **Anchor peer**: riceve gli aggiornamenti e trasmette gli aggiornamenti agli altri peer nell'organizzazione. Quindi, quando bisogna aggiornare la blockchain viene avvisato prima l'anchor peer di una organizzazione e questo propaga le informazioni agli altri peer. Tutto ciò viene fatto per ridurre i partecipanti all'algoritmo di consenso. È possibile configurare canali segreti tra i peer e le transazioni tra i peer di quel canale sono visibili solo ai partecipanti.
 3. **Orderer peer**: rappresenta l'elemento centrale di un canale tra peer ed è responsabile dello stato del registro coerente in tutta la rete ordinando le transazioni e collocandole in nuovi blocchi. Sono attualmente disponibili due impostazioni:
 - **Solo**: un singolo orderer da usare per lo sviluppo, non in produzione, perché poco scalabile e resiliente.
 - **Kafka**: soluzione di streaming processing di Apache che garantisce la ricezione di messaggi nell'ordine di invio.

Si possono avere due tipi di **transazioni**:

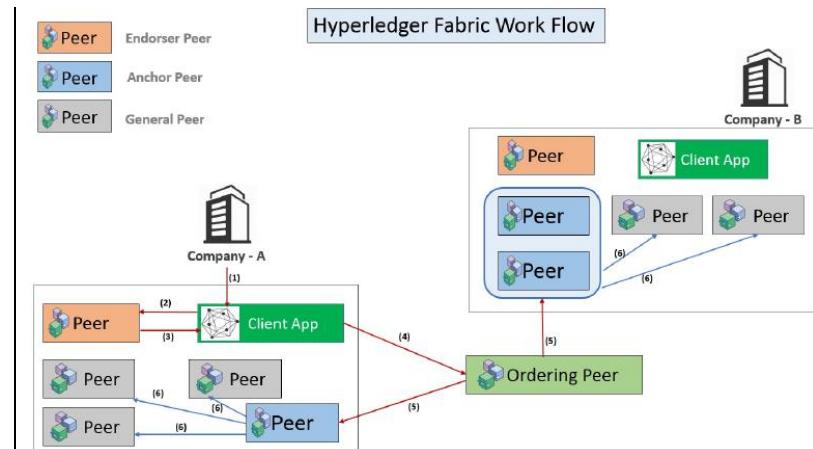
- **Deploy transactions** per creare un nuovo chaincode ed installarlo sulla Blockchain (solo sull'Endorser peer);
- **Invoke transactions** per richiamare le funzioni di un chaincode.

Esempio:

Una infrastruttura su presenta nel modo seguente:

Ci sono (in questo caso) due organizzazioni (Company-A / -B) e un Ordering Peer, in tal caso è centralizzato, se fosse stato con Kafka gli ordering peer erano di più.

Ogni organizzazione ha un Endorser peer che interagisce col client, un Anchor peer che comunica direttamente con l'ordering peer, ed infine i general peer che mantengono la copia del registro, questi non sono direttamente collegati all'ordering peer ma ricevono le informazioni tramite Anchor peer.



Immaginiamo che ogni organizzazione ha una Client App, applicazione con cui dialoga l'utente con la blockchain per mezzo di un SDK:

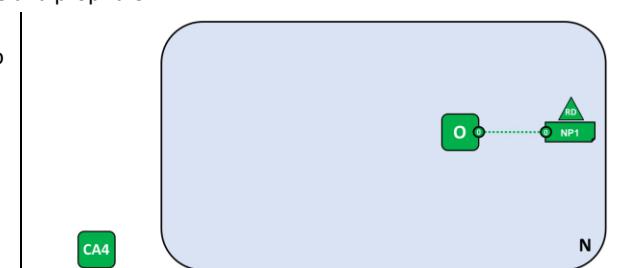
- (1) Un partecipante nell'organizzazione membro effettua una richiesta di transazione tramite l'applicazione client;
- (2) L'applicazione client trasmette la richiesta al Endorser peer, che controlla i dettagli del certificato e altri dettagli per convalidare la transazione;
- (3) Eventualmente esegue il Chaincode e restituisce le risposte di Endorsement, accettando o rifiutando la transazione;
- (4) Il client invia la transazione approvata all'Orderer peer, affinché questa venga ordinata correttamente rispetto ad altre transazioni;
- (5) Il nodo Orderer include la transazione in un blocco, e inoltra il blocco ai nodi Anchor di diverse Organizzazioni membri della rete, che eseguono un consenso distribuito PBFT;
- (6) Al raggiungimento del consenso, gli Anchor peer trasmettono il blocco agli altri peer all'interno della propria organizzazione, così che questi aggiornano il proprio registro locale con l'ultimo blocco. Così tutta la rete ottiene il registro sincronizzato.

CERTIFICATE AUTHORITY (CA):

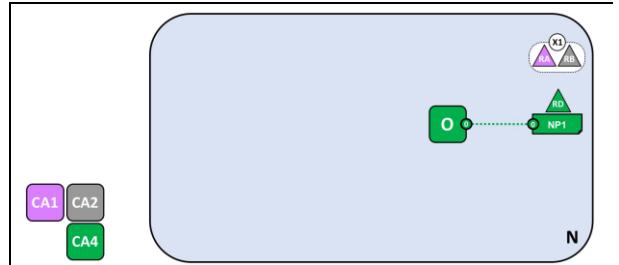
Una **Certificate Authority (CA)** emette i certificati per consentire alle organizzazioni di autenticarsi sulla rete, alle applicazioni client di autenticare le proposte di transazione e ai peer di approvare le proposte e caricare le transazioni nel libro mastro se valide.

Possono esserci una o più CA sulla rete e le organizzazioni possono scegliere di utilizzare una propria CA.

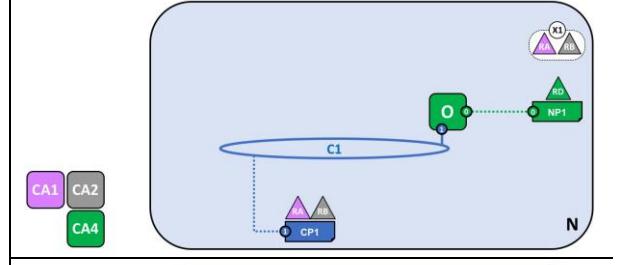
La rete viene creata dalla definizione del servizio di ordinazione (O) con la configurazione per i canali all'interno della rete (NP1), includendo le politiche di accesso e le informazioni sull'appartenenza (come certificati radice X509) per ogni membro del canale (CA4).



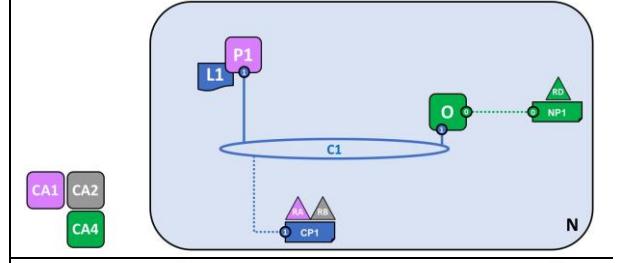
Il consorzio (X1) alla base della rete viene istaurato definendo i certificati delle organizzazioni membro (RD, RA e RB) e le relative CA.



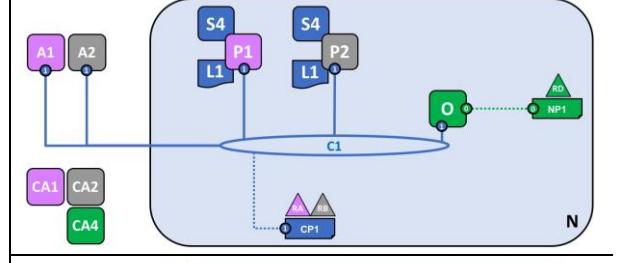
Un canale C1 è creato generando il blocco di configurazione sul servizio di ordinazione (CP1), che valuta la validità della configurazione del canale. L'uso dei canali sono regolati dai criteri con cui sono configurati. Ad esempio, solo le organizzazioni RA e RB posso partecipare al canale.



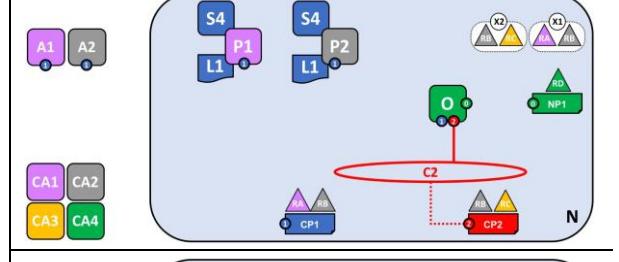
I peer vengono uniti ai canali dalle organizzazioni che li possiedono e possono esserci più nodi peer sui canali all'interno della rete. P1 è il peer che mantiene copia dello stato della blockchain L1 per le transazioni su C1.



Il libro mastro contiene anche delle informazioni private, accessibili sono ad alcune delle organizzazioni sul canale. Sono contenute in un SideDB (*offchain*) e seguono lo stesso processo di endorsement e commit dei dati pubblici, ma la blockchain contiene solo il suo hash.

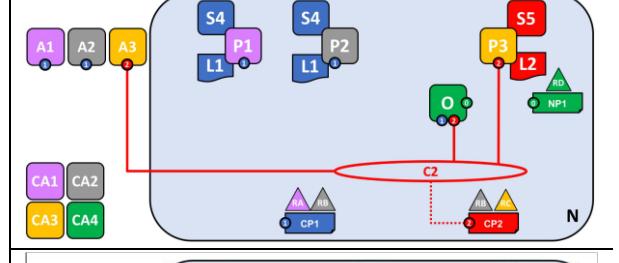


Non vi siano limiti teorici alla dimensione di una rete, e il protocollo gossip è usato per accogliere un gran numero di nodi peer sulla rete e scambiare informazioni sulla rete. Sul canale sono collegate anche le applicazioni client (A1 e A2).



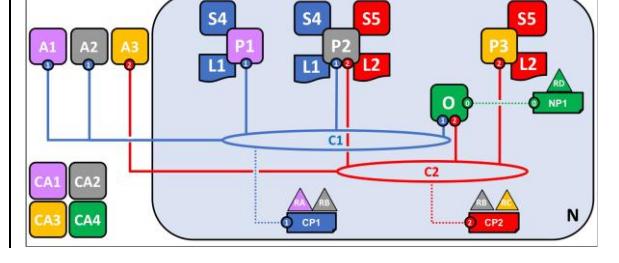
È possibile definire un altro consorzio, dove una terza organizzazione (RC) interagisce con la seconda RB.

Un secondo canale (C2) nell'ambito del secondo consorzio è definibile, con proprie politiche di accesso.



Un peer (P3) per la terza organizzazione può essere istanziato e collegato al secondo canale, dispiegando il chaincode dello smart contract S5.

Nota che le informazioni del canale 1 e le informazioni del canale 2 sono indipendenti.



Ora l'organizzazione RB si trova sia in C1 che in C2, il suo peer P2 ha sia il ledger L1 che L2, questo è il meccanismo di isolamento delle porzioni della blockchain.

Nel complesso l'architettura di Hyperledger è composta da tre macro-aree (o layer):

1. Una per la gestione delle **identità**, dell'appartenenza di peer ad associazioni e l'iscrizione del peer ai canali;
2. Un **libro mastro** con lo stato corrente dei dati e uno storico delle transazioni;
3. L'ambiente di esecuzione dei **chaincode** per gli smart contract.

Trasversalmente sono disponibili delle primitive crittografiche a supporto di questi tre strati.

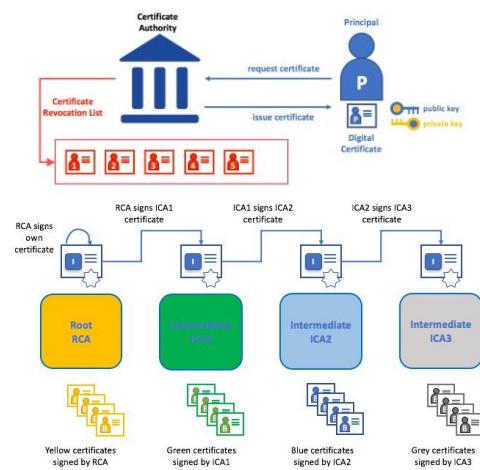
IDENTITÀ DIGITALE:

Ogni elemento ha un'**identità digitale** encapsulata in un certificato digitale X.509, esprimendo anche le autorizzazioni necessarie per le risorse e l'accesso alle funzionalità nella rete blockchain.

Affinché un'identità sia verificabile, deve provenire da un'autorità fidata, come la CA con un modello gerarchico PKI.

Il certificato digitale contiene molte informazioni in merito ad un'identità in SUBJECT, ma anche la chiave pubblica legata all'identità, mentre la sua chiave di firma non lo è e viene mantenuta privata. Il certificato è controfirmato dalla CA che lo emette e lo rende impossibile da modificare.

Esiste una gerarchia di CA, con una Root CA come radice e una serie di CA intermediari i cui certificati sono firmati dalle CA di livello superiore. Fabric offre una propria CA radice.

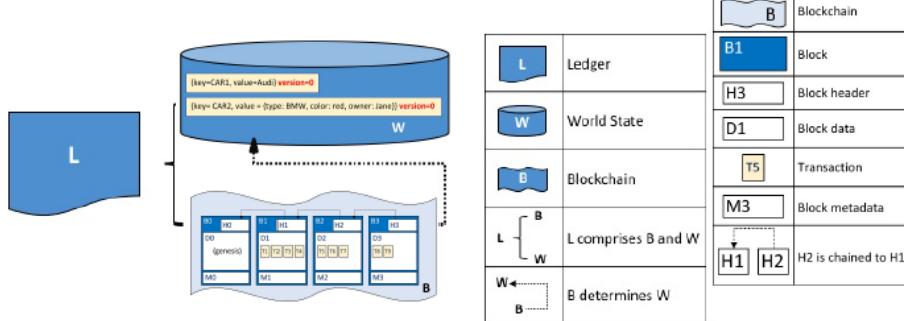


LEDGER:

Il **libro mastro** blockchain è costituito da due parti distinte:

- Uno **stato globale (world state)** in un database che contiene i valori correnti di un insieme di dati indirizzabili per mezzo di una chiave;
- Uno **storico delle transazioni (ledger)** come blockchain, con tutti i cambiamenti dello stato globale.

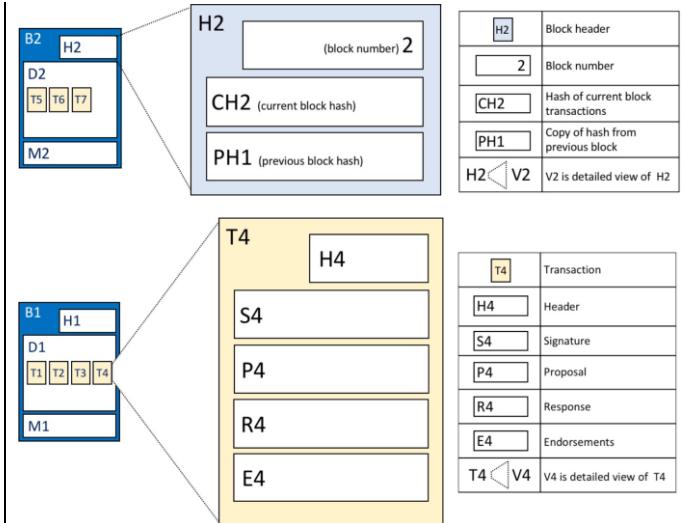
Il numero di versione viene incrementato ogni volta che lo stato cambia, e anche verificato ogni volta che lo stato viene aggiornato.



W	Ledger world state
(key=K, value = V) version=0	A ledger state with key=K. It contains a set of facts expressed as a simple value, V. The state is at version 0.
{key=K, value = {KV}} version=0	A ledger state with key=K. It contains a set of facts expressed as a set of key-value pairs (KV). The state is at version 0.

BLOCCHI E TRANSAZIONI:

Un **blocco** ha un header con un numero di sequenza, l'hash del blocco e del blocco precedente, il corpo con l'insieme ordinato delle transazioni e dei metadati con l'ora di creazione del blocco, il certificato, la chiave pubblica e la firma del creatore del blocco.



Le **transazioni** hanno un header come il nome del chaincode pertinente e la sua versione, la firma del creatore, la proposta e la risposta con l'endorsement.

CHAINCODE:

Chaincode è un programma scritto in Go, node.js o Java che implementa un'apposita interfaccia e viene eseguito in un contenitore Docker isolato dal processo dell'Endorser peer. Lo stato creato da un chaincode è limitato esclusivamente a quel chaincode e non è possibile accedervi direttamente da un altro chaincode. Tuttavia, un chaincode può invocare un altro chaincode per accedere al suo stato.

L'interfaccia Chaincode specifica le funzioni da implementare:

- Init()** viene chiamato quando un chaincode riceve un'istanza o una transazione di aggiornamento in modo da eseguire qualsiasi inizializzazione necessaria, inclusa l'inizializzazione dello stato dell'applicazione;
- Invoke()** viene chiamato in risposta alla ricezione di una transazione invoke per elaborare le proposte di transazione.

L'altra interfaccia è ChaincodeStubInterface, che viene utilizzato per accedere e modificare la blockchain e per effettuare invocazioni tra i chaincode. Dato che il supporto Java è stato introdotto di recente, le API Node.js e Go lang sono a un livello più maturo. JavaScript non si presta bene nel caso di calcoli numerici.

LINGUAGGIO GO:

Il linguaggio più usato per realizzazione chaincode è Go. Questo linguaggio non presenta il concetto di classe, ma Go offre struct, una versione leggera delle classi, a cui è possibile aggiungere metodi.

Realizziamo uno smart contract in Go per la gestione delle auto e dei loro proprietari. Implementiamo le struct per questi dati.

```
type Car struct{
    modelName string
    color string
    serialNo string
    manufacturer string
    owner Owner //composition
}
```

```
type Owner struct{
    name string
    nationalIdentity string
    gender string
    address string
}
```

Il metodo è definito esternamente, indicando nel primo caso se le modifiche nel corpo del metodo si riflettono sul chiamante.

```
//attached by reference ==> called as pointer receiver
func (c *Car) changeOwner(newOwner Owner) {
    c.owner = newOwner
}

/** attached by value ==> called as value receiver
func (c Car) changeOwner(newOwner Owner) {
    c.owner = newOwner
}
*/
```

L'interfaccia da implementare è la seguente con le due principali funzioni delle 36 attualmente specificate -->

Non sussiste una sintassi di derivazione, ma basta solo implementare i metodi di interesse.

Inseriamo la logica di inizializzazione:

```
func (t *CarChaincode) Init(stub shim.ChaincodeStubInterface)
pb.Response {

//Declare owners from Owner struct
tom := Owner{name: "Tom H", nationalIdentity: "ABCD33457", gender:
"M", address: "1, Tumbbad"}
bob := Owner{name: "Bob M", nationalIdentity: "QWER33457", gender:
"M", address: "2, Tumbbad"}

//Declare carfrom Car struct
car := Car{modelName: "Land Rover", color: "white", serialNo:
"334712531234", manufacturer: "Tata Motors", owner: tom}

// convert tom Owner to []byte
tomAsJSONBytes, _ := json.Marshal(tom)
//Add Tom to ledger
err := stub.PutState(tom.nationalIdentity, tomAsJSONBytes)
if err != nil {
    return shim.Error("Failed to create asset " + tom.name)
}

//Add Bob to ledger
bobAsJSONBytes, _ := json.Marshal(bob)
err = stub.PutState(bob.nationalIdentity, bobAsJSONBytes)
if err != nil {
    return shim.Error("Failed to create asset " + bob.name)
}

//Add car to ledger
carAsJSONBytes, _ := json.Marshal(car)
err = stub.PutState(car.serialNo, carAsJSONBytes)
if err != nil {
    return shim.Error("Failed to create asset " + car.serialNo)
}

return shim.Success([]byte("Assets created successfully."))
}
```

```
type Chaincode interface {
    // Init method accepts stub of type ChaincodeStubInterface as
    // argument and returns peer.Response type object
    Init(stub ChaincodeStubInterface) peer.Response
}
```

```
type CarChaincode struct{
}
```

```
//Init implemented by CarChaincode
func (t *CarChaincode) Init(stub shim.ChaincodeStubInterface)
pb.Response {
```

```
}
```

```
//Invoke implemented by CarChaincode
func (t *CarChaincode) Invoke(stub shim.ChaincodeStubInterface)
pb.Response {
```

```
}
```

Inseriamo la logica di invoke:

```
func (c *CarChaincode) Invoke(stub shim.ChaincodeStubInterface)
pb.Response {

    // Read args from the transaction proposal.
    // fc=> method to invoke
    fc, args := stub.GetFunctionAndParameters()
    if fc == "TransferOwnership" {
        return c.TransferOwnership(stub, args)
    }
    return shim.Error("Called function is'n't defined in the chaincode")
}

func (c *CarChaincode) TransferOwnership(stub
shim.ChaincodeStubInterface, args []string) pb.Response {
    // args[0]==> car serial no
    // args[1]==> new owner national identity
    // Read existing car asset
    carAsBytes, _ := stub.GetState(args[0])
    if carAsBytes == nil {
        return shim.Error("car asset not found")
    }

    // Construct the struct Car
    car := Car{}
    _ = json.Unmarshal(carAsBytes, &car)

    //Read newOwnerDetails
    ownerAsBytes, _ := stub.GetState(args[1])
    if ownerAsBytes == nil {
        return shim.Error("owner asset not found")
    }

    // Construct the struct Owner
    newOwner := Owner{}
    _ = json.Unmarshal(ownerAsBytes, &newOwner)

    // Update owner
    car.changeOwner(newOwner)

    carAsJSONBytes, _ := json.Marshal(car)

    // Update car ownership in the
    err := stub.PutState(car.serialNo, carAsJSONBytes)
    if err != nil {
        return shim.Error("Failed to create asset " + car.serialNo)
    }
    return shim.Success([]byte("Asset modified."))
}
```

```
package main
```

```
import (
    "encoding/json"
    "github.com/hyperledger/fabric/core/chaincode/shim"
    "github.com/hyperledger/fabric/protos/peer"
)
func main() {

    logger.SetLevel(shim.LogInfo)

    // Start chaincode process
    err := shim.Start(new(CarChaincode))
    if err != nil {
        logger.Error("Error starting PhantomChaincode - ", err.Error())
    }
}
```

Per completare lo smart contract è necessario specificare il preambolo:

E la funzione principale:

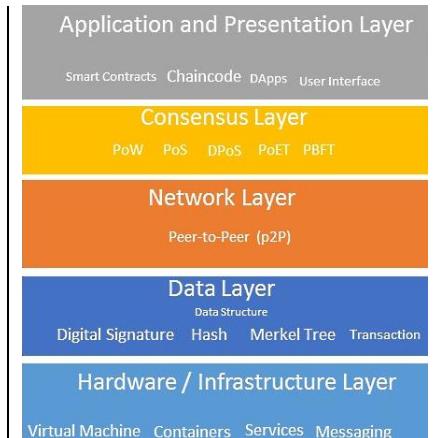
5. SICUREZZA E PRIVACY IN BLOCKCHAIN

Al giorno d'oggi, ci sono sempre più attacchi alle criptovalute. Spesso questi attacchi vengono lanciati su applicazioni blockchain a causa della loro popolarità o del capitale coinvolto nel loro sistema. A causa di una natura pubblicamente verificabile, le criptovalute basate su Blockchain sono vulnerabili a diverse attività fraudolente.

La blockchain ha una stratificazione di layer e gli attacchi possono avvenire in ogni livello:

- **Attacchi a livello applicativo (Application and Presentation Layer)**, associati al contesto applicativo che utilizza la tecnologia Blockchain, ad esempio se si vuole rubare cripto è più facile fare phishing o social engineering su un cellulare che contiene un wallet anziché entrare nella rete blockchain;
- **Attacchi a livello rete (Network Layer)**, associati all'architettura peer-to-peer utilizzata nel sistema Blockchain (delay of service, man in the middle, ecc...);
- **Attacco a livello dati (Data Layer)**, associati alle tecniche matematiche utilizzate per la creazione del libro mastro.

Il modello di fiducia debole espone le blockchain pubbliche (Bitcoin ed Ethereum) a un'ampia varietà di attacchi, consentendo agli avversari di compromettere facilmente il sistema. Per affrontare le carenze delle Blockchain pubbliche e ridurre le opportunità di attacco, le blockchain private (come Fabric) e con autorizzazione (esempio la CA in Fabric) vengono ora utilizzate per varie applicazioni.



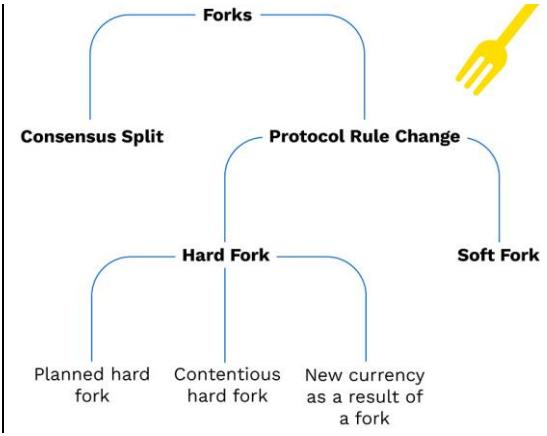
ATTACCHI ALLA STRUTTURA:

La prima tipologia di attacchi è quella che causano dei **fork**. I fork possono essere creati involontariamente a causa di malfunzionamenti o incompatibilità negli aggiornamenti. Le fork possono anche essere causate con intenti dannosi mediante "nodi Sybil" o il "mining egoistico".

Un'altra forma di fork si verifica quando gli utenti di un'applicazione blockchain creano un'applicazione figlia dall'applicazione padre.

I **fork intenzionali** possono essere:

- Una **soft fork** si ha quando la modifica è retrocompatibile, ovvero i nodi aggiornati possono comunque comunicare con quelli non aggiornati;
- Una **hard fork** sono aggiornamenti software non retrocompatibili. Tipicamente, si verificano quando i nodi aggiungono nuove regole in conflitto con le regole dei vecchi nodi. I nuovi nodi possono comunicare solo con altri che usano la nuova versione. Di conseguenza, la blockchain si divide, creando due network separati: uno con le vecchie regole e uno con le nuove regole.



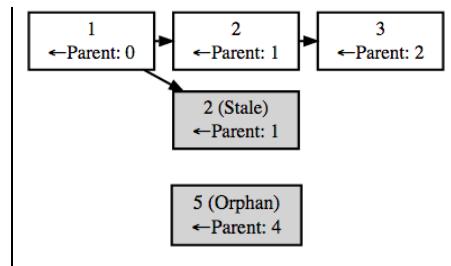
Un fork rappresenta uno stato incoerente che può essere sfruttato dagli avversari per causare confusione, transazioni fraudolente e sfiducia all'interno della rete.

Gli hard fork possono portare a una divisione della criptovaluta. Quando degli attacchi sono riusciti a compromettere con successo una criptovaluta, si è impiegato un hard fork per annullare le transazioni maliziose. Tuttavia, questo richiede il consenso della maggior parte dei nodi.

Se si verifica un ritardo del consenso a causa di un attacco di maggioranza o di un evento DDoS, le attività fraudolente diventano alquanto difficili da gestire e ritardi prolungati possono infine causare la svalutazione della criptovaluta.

Due forme di incongruenze possono verificarsi con il processo di consenso che può lasciare blocchi validi fuori dalla Blockchain:

- **Stale block - blocco obsoleto**, se un blocco è stato «mined» con successo ma non è sulla catena principale accettata. Si verifica principalmente nelle Blockchain pubbliche a causa delle race conditions tra i miners: due o più miners possano trovare una soluzione valida, tra cui la rete ne accetta solo uno e scarta il resto, che diventano blocchi obsoleti. Il "selfish mining" può anche portare a blocchi obsoleti.;
- **Orphaned Block - blocco orfano**, se un blocco valido ha il campo hash del blocco genitore che punta a un blocco non autentico.



Il **consenso** utilizzato determina molto le vulnerabilità verificabili:

- Uno dei maggiori problemi con **PoW** è l'eccessivo spreco di energia per trovare una valida soluzione. La centralizzazione della capacità di hashing tra pochi gruppi di miners rende l'applicazione Blockchain vulnerabile agli attacchi, inclusi gli attacchi della maggioranza e la doppia spesa: se un miner acquisisce la maggioranza dell'hash rate di una rete, sarà in grado di prendere il controllo del sistema;
- **PoS** implica un consenso deterministico poiché un validatore viene scelto prima e i blocchi vengono validati senza ritardi. Inoltre, per lanciare un attacco di maggioranza, l'attaccante deve acquisire più del 50% dei token di criptovaluta, più difficile da ottenere rispetto al 50% di hash rate in PoW. Il meccanismo si polarizza verso i validatori più ricchi che tengono ad arricchirsi progressivamente per la validazione dei blocchi;
- **PBFT** è considerato efficiente dal punto di vista energetico con un elevato throughput di transazione. Tuttavia, assume che la replica primaria esegua fedelmente il protocollo e non manometta l'ordine di transazioni e blocchi. Questa ipotesi può portare a una vulnerabilità nelle Blockchain permissioned, ma siccome l'identità della replica primaria è nota, è possibile monitorarne il comportamento.

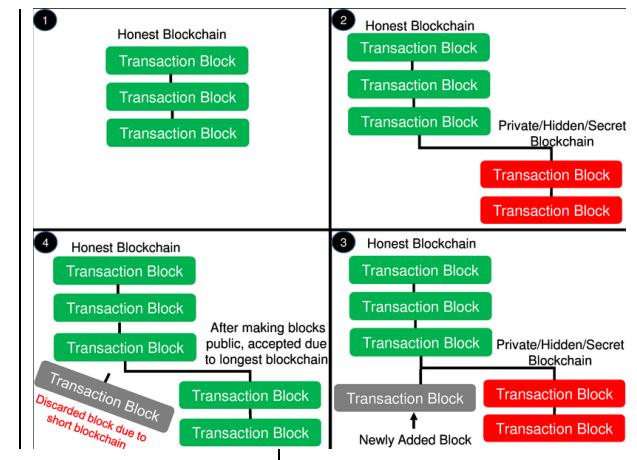
Ma PBFT è affetto da una limitata scalabilità e bassa tolleranza ai gusti bizantini, che aumenta la possibilità per un avversario di inserire repliche dannose. Attualmente, Bitcoin ha oltre 10.000 nodi completi attivi, e può tollerare fino a 5.000 nodi difettosi. In una blockchain privata basata su PBFT che consiste di 100 nodi, un aggressore può avere successo solo controllando 33 nodi.

ATTACCHI AL SISTEMA P2P:

L'attacco del **selfish mining** è una strategia scelta da alcuni minatori che tentano di aumentare le loro ricompense mantenendo deliberatamente privati i loro blocchi e determinando una block race tra la catena pubblica degli onesti e la catena privata degli egoisti.

Ciò causa risultati indesiderabili invalidando i blocchi onesti con tutte le loro ricompense che vengono rifiutate e la possibilità di occorrenza di fork che possono causare un ritardo del consenso e il sopraggiungere di altri potenziali attacchi come "doppia spesa" e "fork after withholding".

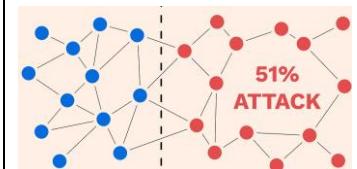
Questo attacco viene adoperato inizialmente per poi effettuare altri attacchi.



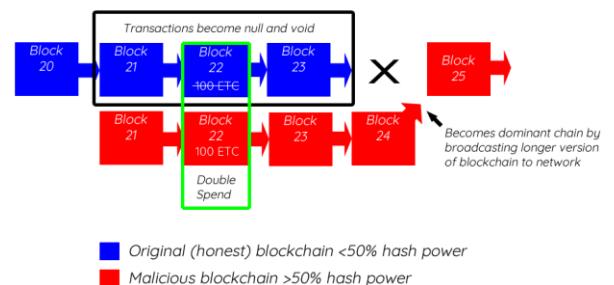
L'attacco della **maggioranza**, o **attacco del 51%**, è una vulnerabilità che può essere sfruttata quando un singolo aggressore, un gruppo di nodi Sybil (nodi che si comportano in modo deviante, compromessi in maniera indiretta) o un pool di mining nella rete raggiunge la maggior parte dell'hash rate della rete per manipolare la blockchain ed essere in grado di:

1. impedire la verifica di transazioni o blocchi (rendendoli non validi);
2. invertire le transazioni durante il tempo in cui hanno il controllo per consentire la doppia spesa;
3. forkare la Blockchain principale e dividere la rete;
4. impedire ad altri miners di validare blocchi per un breve periodo di tempo.

Si consideri lo scenario in cui un pool di mining malizioso con un potere di hash significativo esegue una transazione T_x valida. Allo stesso tempo, genera una transazione fraudolenta T_y a doppia spesa dalla stessa transazione genitore. Il destinatario attende k conferme della fork prima di rilasciare il prodotto al miner, ovvero che k blocchi successivi sono stati estratti dalla rete dopo aver validato la transazione T_x . Durante questo processo, il miner malizioso continua a estrarre blocchi. Forzando la catena, il miner dannoso sarà in grado di invalidare la catena con transazione T_x e la sostituirà con la propria catena con T_y .



51% Attack (double-spend)

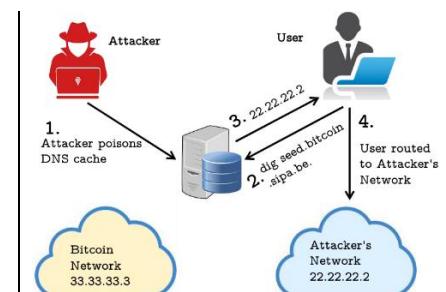


Quando un nodo si unisce alla rete Bitcoin per la prima volta, non è a conoscenza dei peer attivi. Per scoprirli, è necessario un meccanismo di bootstrap (o di Discovery) come il Domain Name System (DNS). I seed DNS vengono interrogati per ottenere informazioni sui peer attivi con cui stabilire delle connessioni. La query DNS iniziale restituisce uno o più records DNS con gli indirizzi IP corrispondenti ai peer che accettano le connessioni in entrata.

Il DNS rappresenta una criticità per blockchain siccome è vulnerabile ad attacchi man-in-the-middle.

Per impostazione predefinita, il client Blockchain ha un elenco di seeder che consentono la scoperta della rete. Se l'aggressore inserisce un falso elenco, l'utente verrà compromesso e condotto a una rete contraffatta.

Un nodo nella rete Bitcoin ha un indirizzo IP di 33.33.33.3, mentre il nodo dell'aggressore in una rete contraffatta ha un indirizzo IP 22.22.22.2. L'autore realizza un attacco DNS cache poisoning per indurre l'utente a entrare nella rete contraffatta. L'utente esegue la query DNS su seed.bitcoin.sipa.be, e invece di rispondere con 33.33.33.3, il resolver DNS restituisce 22.22.22.2. Di conseguenza, l'utente si connette a nodi maliziosi nella rete contraffatta e i nodi dannosi potrebbero fornire falsi blocchi all'utente.

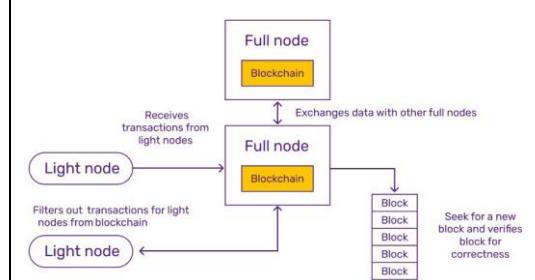


Ci sono due tipi di nodi nella maggior parte delle applicazioni blockchain (come Bitcoin):

- I **full nodes** sono partecipanti effettivi alla rete e mantengono una copia aggiornata della blockchain;
- I **lightweight nodes** utilizzano solo i servizi dei nodi completi per ottenere l'accesso alla rete.

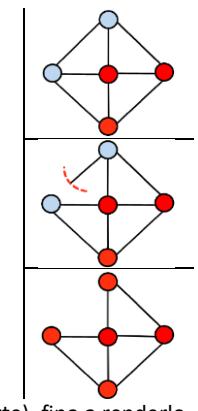
Poiché i nodi leggeri traggono la loro visione della Blockchain dai nodi completi, quando un nodo completo viene compromesso, anche tutti i suoi nodi leggeri associati vengono compromessi. La concentrazione spaziale dei nodi completi all'interno di un AS o di un ISP li rende vulnerabili agli attacchi di routing come il dirottamento BGP (Border Gateway Protocol). Un AS antagonista può dirottare il traffico per un AS target che ospita la maggior parte dei nodi dell'applicazione blockchain.

Trasmettendo percorsi di rete maliziosi tramite BGP, l'attaccante è in grado di reindirizzare il traffico da server di mining legittimi su una rete a server fasulli su un'altra rete mascherata come originale. Questi server di mining fasulli hanno permesso ai clienti di continuare a estrarre criptovaluta ma non hanno mai ricevuto alcuna ricompensa per il lavoro svolto. Tutti i proventi dell'attività di mining sono invece andati direttamente verso i nodi fasulli dell'attaccante.



Il sistema peer-to-peer di Blockchain è anche vulnerabile a una forma di attacco nota come **Eclipse attack**, in cui un gruppo di nodi dannosi isolano i nodi vicini, compromettendo così il traffico in entrata e in uscita. In Bitcoin, i nodi formano dei cluster, dove ogni peer è a conoscenza dell'indirizzo IP di tutti. Con un numero sufficiente di nodi compromessi in un cluster, si può isolare i nodi onesti e modificarne la copia locale dello stato della blockchain, controllando il loro traffico in entrata e in uscita e fornire loro informazioni false.

I nodi blu rappresentano i nodi onesti che memorizzano il vero stato della blockchain mentre i nodi rossi rappresentano i nodi maliziosi che formano un cluster attorno ai nodi blu.



Se la connessione tra i nodi onesti è compromessa, i nodi dannosi possono inviare blocchi falsi ai nodi onesti e partizionarli dalla rete, perdendo la connessione tra nodi onesti.

Di conseguenza, i nodi onesti finiscono per avere una visione sbagliata dello stato della blockchain.

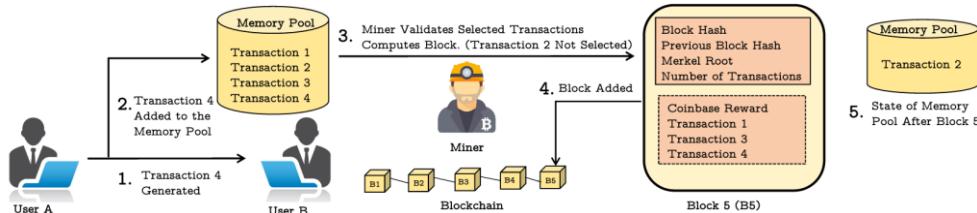
Uno degli attacchi più comuni è **Distributed Denial of Service (DDoS)**, consiste nel tempestare di richieste un sistema (o una sua parte), fino a renderlo irraggiungibile. Nonostante sia un sistema peer-to-peer, blockchain è soggetto ad attacchi DDoS, manifestandosi in diversi modi.

Il numero di transazioni per blocco che un'applicazione blockchain può elaborare in un dato tempo è limitato:

- Un blocco BitCoin ha la dimensione di 1MB, mentre la dimensione media di una transazione è di circa 500 byte, consentendo in media circa 2.000 transazioni per blocco;
- Il tempo medio per il mining di un blocco è di circa 10 minuti. Pertanto, non si può superare le 200 transazioni al minuto, e il totale dei peer attivi serviti dalla rete al non supererà 200;
- Dati questi vincoli, il throughput è di 3-7 transazioni al secondo.

Un avversario può sfruttare varie identità Sybil ed emettere diverse transazioni di "polvere" (ad esempio 0,001 BTC per transazione) tra le varie identità Sybil sotto il suo controllo. Introducendo un gran numero di transazioni di scarso valore in un breve periodo di tempo, la rete sarà congestionata e il servizio agli utenti legittimi nella rete sarà negato.

Un'altra forma di attacco DDoS viene eseguita presso i **mempool** delle criptovalute per aumentare la tariffa di mining. In BitCoin, l'utente A genera una transazione per l'utente B. La transazione viene archiviata nel pool di memoria insieme ad altre transazioni non confermate. Il minatore convalida le transazioni dal pool di memoria e calcola un blocco. Un blocco valido viene aggiunto alla blockchain.



Sebbene la dimensione del blocco sia limitata nelle criptovalute, la dimensione del pool non ha limiti di dimensione.

Più transazioni sono nel mempool, più forte è la concorrenza per accaparrarsi i miners. Per dare maggiore priorità alle proprie transazioni, gli utenti iniziano a pagare maggiori mining rewards, come incentivo per i miners.

Un avversario potrebbe inondare i mempool con transazioni di polvere, portando gli utenti legittimi ad aumentare le commissioni di mining così che le transazioni dell'attaccante non vengonominate.

La natura peer-to-peer alla base di una blockchain può essere sfruttata per creare visioni contrastanti del suo stato globale. I nodi dannosi possono intenzionalmente mascherare, falsificare o nascondere informazioni importanti che devono essere trasmesse attraverso la rete.

L'**attacco Finney** è una variante del double spending in cui un minatore ritarda la propagazione dei blocchi per spendere due volte la sua transazione.

1. Il miner genera una transazione, computa un blocco e sceglie di non ritrasmettere il blocco;
2. Genera un duplice della sua transazione precedente e lo invia a un destinatario;
3. Dopo che il destinatario accetta la transazione e consegna il prodotto, il miner pubblica il suo blocco precedente con la transazione originale;
4. La transazione precedente diventa non valida e il minatore spende con successo il doppio.

Nei pool di mining decentralizzati, tutti i partecipanti consumano elettricità e potenza della CPU per trovare un nonce il cui valore di un hash con il blocco è inferiore alla soglia di destinazione. Una volta trovata la soluzione valida, tutti i partecipanti vengono ricompensati in base al loro impegno.

Nel **block withholding attack**, un miner compromesso risolve il puzzle crittografico della PoW, ma sceglie di non rivelarla al pool server. Ignari di ciò, il resto dei miner del pool sprecano le loro risorse per trovare il nonce e alla fine perdono la gara. Infatti, il miner malizioso può condividere il nonce trovato con nodi in altri pool per una ricompensa maggiore.

Un'variante di questo attacco è il **Fork After Withholding (FAW) attack**:

1. Un miner dannoso si unisce a due pool di mining;
2. Il miner calcola un PoW valido nel primo pool;
3. Egli trattiene la soluzione e pubblica il blocco solo quando anche il secondo pool pubblica il blocco;
4. La rete risolve la fork selezionando un blocco tra i due;
5. Il minatore dannoso viene ricompensato in entrambi i casi.

Se l'attacco FAW viene lanciato tra due o più pool di pool, quello più grande vincerà sempre nel caso di competizione. Pertanto, l'attacco FAW è più redditizio del selfish mining e block withholding attack. Nelle blockchain permissioned, se l'avversario controlla la replica primaria, può trattenere blocchi e transazioni per compromettere il sistema e ritardare l'elaborazione delle transazioni.

Un altro attacco associato coinvolge il **ritardo del consenso**. Un utente malintenzionato può iniettare falsi blocchi per aggiungere latenza o impedire ai peer di raggiungere il consenso, propagando blocchi obsoleti o transazioni doppie. Nelle Blockchain private basate su PBFT, un avversario può causare un ritardo del consenso anche se controlla un basso numero di repliche (meno del 33%).

Dei nodi Sybil possono anche inviare firme fasulle alle altre repliche durante la fase di prepare e commit. Poiché ogni replica è tenuta a verificare le firme, si ha un sovraccarico aggiuntivo. Se si continua a inviare tali firme, è possibile rallentare o bloccare il completamento della fase di commit.

La replica primaria non riceverà il numero di approvazioni richiesto per la verifica della transazione.

Ogni blocco memorizza un timestamp che rappresenta il tempo approssimativo della sua creazione. In Bitcoin, ad esempio, un nodo rifiuta un blocco se il suo timestamp supera il tempo di rete di 120 minuti. Allo scopo, i full nodes mantengono un contatore interno che denota l'ora della rete, e si sincronizzano mediante un algoritmo di sincronizzazione interna.

Un attaccante, con nodi Sybil, può rallentare il tempo di rete di un nodo target, inviandogli timestamp variabili fasulli. Ciò può portare a una differenza tra il tempo di blocco e il contatore del nodo maggiore di 120 minuti, con il conseguente rifiuto del blocco e di tutti quelli successivi. Il nodo target alla fine viene isolato.

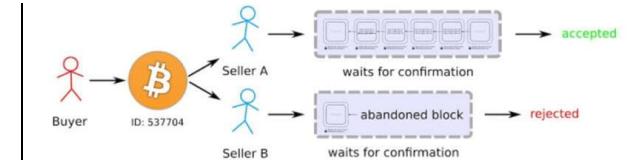
ATTACCHI ALLE APPLICAZIONI:

Le blockchain pubbliche hanno una nozione debole di anonimato e forniscono accessibilità ai dati aperti al pubblico. L'analisi della Blockchain pubblica o **block ingestion** può rivelare informazioni utili a un avversario, e potrebbe non essere desiderabile per un'applicazione o per i suoi utenti.

Nelle blockchain, una volta che una transazione è stata approvata, non può essere annullata. Ciò ha portato a varie attività di truffa irreversibili online, in cui gli utenti sono indotti a inviare denaro. L'assenza di un'autorità centrale rende più difficile denunciare la frode e aspettarsi il rimborso.

Nelle criptovalute, il double-spending si riferisce all'uso di una transazione due o più volte, firmando la stessa transazione con una chiave privata e inviandola a due diversi destinatari.

Un compratore ha la transazione con id 537704 nel suo saldo. Usandola come input, genera due transazioni per pagare due venditori. Quando un miner interroga il mempool, può selezionare una delle due per creare un nuovo blocco inclusi in un ramo della catena accettata. L'altra transazione è contenuta in un blocco rigettato, pertanto il suo venditore non vedrà corrisposto il pagamento.

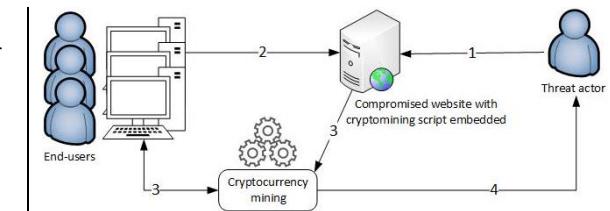


Il **cryptojacking** è una forma di attacco che viene lanciata su servizi web e cloud per eseguire illegalmente PoW, senza che l'utente ne sia consapevole o dia il proprio consenso, per criptovalute basate su blockchain permissionless.

Il cryptojacking consiste nel rubare le risorse di elaborazione dai dispositivi delle loro vittime. Combinando tutte queste risorse, gli hacker sono in grado di competere con sofisticate operazioni di cryptomining senza gli elevati costi associati.

Un avversario può compromettere un computer in vario modo come facendo clic su un link dannoso in un'e-mail, e caricare il codice di cryptomining direttamente sul computer vittima.

Il codice di cryptomining può ricevere richieste di lavoro di mining e svolgerle, rallentando le elaborazioni sulla macchina vittima. Il frutto di tali elaborazioni sono blocchi validati che vengono inseriti nella blockchain, e le relative reward sono girate all'attaccante.



Laddove le credenziali, come le chiavi, associate ai peer nel sistema sono archiviate in un portafoglio digitale, l'attacco **"furto del portafoglio"** diventa particolarmente critico e probabile.

In Bitcoin, il portafoglio viene archiviato di default non crittografato, consentendo a un avversario di apprendere le credenziali associate e quali transazioni esso ha emesso.

Anche quando protetto in modo sicuro sull'host, attacchi alla macchina ospitante consentirà all'avversario di rubarlo.

Sussistono molti servizi di terze parti che consentono l'archiviazione di portafogli, anche questi servizi possono essere compromessi e i relativi portafogli esposti a un avversario.

Un problema ben noto nelle criptovalute basate su Blockchain è l'esposizione e il **furto di chiavi private**. Se l'aggressore acquisisce la chiave privata di un utente, può firmare e generare una nuova transazione al suo posto, spendendo il suo saldo. Le blockchain pubbliche dispongono di applicazioni client open source che consentono agli utenti di connettersi alla rete. Nel tempo vengono rilasciate nuove versioni del software, implementando nuove regole e aggiornamenti, o patch per risolvere vulnerabilità. Non tutti i nodi scaricano la versione appena rilasciata, ma continuano a usare il vecchio software e rimangono esposti alle sue vulnerabilità.

Il codice open source può essere sfruttato da un avversario per rilasciare un nuovo aggiornamento con codice dannoso. Se un utente installa il software, può fornire l'accesso all'attaccante che può lanciare vari attacchi e ghermire dati sensibili dell'utente.

Nuove applicazioni sono realizzate sfruttando un apposito linguaggio di programmazione, e caratterizzate da una serie di vulnerabilità e creano una nuova superficie di attacco. Un certo numero di progetti hanno identificato la sicurezza degli smart contract come un'esigenza del mercato e hanno proposto soluzioni per l'audit e la revisione degli smart contract mediante la loro **analisi statica**.

5.1 PROGRAMMAZIONE SICURA DI SMART CONTRACT

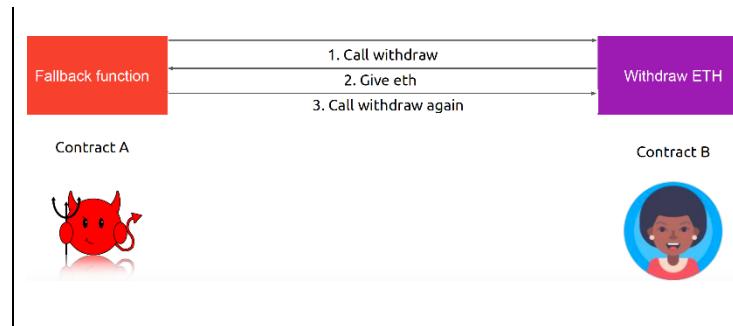
Una delle caratteristiche degli smart contract di Ethereum (scritti in Solidity) è la possibilità di **chiamare e utilizzare il codice di altri contratti esterni**, e ciò richiede una **chiamata esterna**. Queste chiamate esterne possono essere dirottate dagli aggressori per eseguire ulteriore codice (cioè una funzione di **fallback**), comprese le chiamate in sé stesso. Così l'esecuzione del codice "rientra" nel contratto.

Un contratto non è monolitico, cioè non dispone di tutti i servizi e la logica per la realizzazione di un servizio preciso, ma molto spesso si può avere che il contratto A interagisce/esegue un contratto B (può essere una libreria o un altro contratto che offre un diverso servizio) per fornire il servizio finale.

Esempio:

Il contratto A è un servizio di vendita di asset e il contratto B è il token ERC20 che si può utilizzare come moneta di scambio. Queste chiamate esterne possono essere dirottate per poter eseguire altro codice che un attaccante è interessato a poter svolgere, come una chiamata di fallback.

1. Il contratto A invoca una funzionalità del contratto B (esempio un withdraw di cripto);
2. Per effetto della chiamata, il contratto B restituisce della criptovaluta al contratto A;
3. Il contratto A può, in realtà, chiamare di nuovo withdraw sul contratto B ed avere di nuovo della criptovaluta, creando un possibile loop.



- [righe 6-8] La funzione **depositFunds()** incrementa semplicemente il saldo del mittente.
- [righe 10-19] La funzione **withdrawFunds()** consente al mittente di specificare la quantità di wei da prelevare. Avrà successo solo se l'importo richiesto da prelevare è inferiore al saldo e a 1 Ether e che non si è verificato un prelievo nell'ultima settimana.
- [riga 16] è presente la vulnerabilità, dove si invia all'utente la quantità di Ether richiesta.

- [righe 3-7] L'attaccante può creare un nuovo contratto (diciamo all'indirizzo 0x0 ... 123) con l'indirizzo del contratto EtherStore come parametro del costruttore.
- [righe 9-16] L'attaccante può così chiamare la funzione **pwnEtherStore()**, con una certa quantità di Ether (≥ 1).
- [righe 12-15] La funzione **depositFunds()** del contratto EtherStore verrà chiamata con un valore di 1 Ether (e molto gas). Il mittente (**msg.sender**) sarà il nostro contratto dannoso (0x0 ... 123). Pertanto, **balance[0x0..123] = 1 Ether**. Il contratto dannoso chiamerà quindi la funzione **withdrawFunds()** del contratto EtherStore con un parametro di 1 Ether. Questo soddisferà tutti i requisiti della funzione e il contratto EtherStore invierà quindi 1 Ether al contratto dannoso.
- [righe 22-26] L'Ether inviato al contratto dannoso eseguirà quindi la funzione di fallback. Siccome un solo Ether è stato prelevato (il saldo totale del contratto EtherStore era di 10 Ether ed ora è 9 Ether), la condizione dell'**if** è soddisfatta, quindi questa istruzione è soddisfatta. La funzione di fallback chiama quindi di nuovo la funzione **withdrawFunds()** di EtherStore e "rientra" nel contratto EtherStore. In questa seconda chiamata a **withdrawFunds()**, il saldo dell'utente è ancora 1 Ether poiché la riga [17] non è stata ancora eseguita. Quindi, abbiamo ancora un saldo disponibile e pari a 1 Ether. Anche la riga [18] non è stata eseguita, superando tutti i requisiti e viene prelevato un altro Ether. L'esecuzione della fallback function continua fino a quando **EtherStore.balance >= 1**. Una volta che nel contratto EtherStore è rimasto meno di 1 Ether, questa istruzione **if** non sarà soddisfatta. Ciò consentirà quindi di eseguire le righe [17] e [18] del contratto EtherStore (per ogni chiamata alla **funzionedrawFunds()**). I saldi e le mappature **lastWithdrawTime** verranno impostati e l'esecuzione terminerà.

Il risultato finale è che l'attaccante ha ritirato tutti gli Ether dal contratto EtherStore, istantaneamente con una singola transazione.

Esistono numerose tecniche per evitare potenziali vulnerabilità di rientro negli smart contract:

- [riga 23] Utilizzare la funzione **transfer()** (invece di call) quando si invia Ether a contratti esterni, che invia solo 2300 gas che non sono sufficienti per chiamare un altro contratto (ovvero reinserire il contratto di invio);
- [righe 19-23] Assicurarsi che tutta la logica che cambia le variabili di stato avvenga prima che l'Ether venga inviato dal contratto. Nell'esempio EtherStore, le righe [17] e [18] di EtherStore.sol devono essere inserite prima della riga [16]. È buona norma inserire qualsiasi codice che esegua chiamate esterne a indirizzi sconosciuti come ultima operazione in una funzione, secondo il *checks-effects-interactions pattern*;
- [riga 3/13/22] Introdurre un mutex, così da bloccare il contratto durante l'esecuzione del codice, impedendo le chiamate di rientro.

NOTA: la funzione **call** trasferisce la valuta, mentre **transfer()** ha un costo in gas.

```

1  contract EtherStore {
2      uint256 public withdrawalLimit = 1 ether;
3      mapping(address => uint256) public lastWithdrawTime;
4      mapping(address => uint256) public balances;
5
6      function depositFunds() public payable {
7          balances[msg.sender] += msg.value;
8      }
9
10     function withdrawFunds (uint256 _weiToWithdraw) public {
11         require(balances[msg.sender] >= _weiToWithdraw);
12         // limit the withdrawal
13         require(_weiToWithdraw <= withdrawalLimit);
14         // limit the time allowed to withdraw
15         require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
16         require(msg.sender.call.value(_weiToWithdraw)());
17         balances[msg.sender] -= _weiToWithdraw;
18         lastWithdrawTime[msg.sender] = now;
19     }
20 }
21
22 import "EtherStore.sol";
23 contract Attack {
24     EtherStore public etherStore;
25     // initialise the etherStore variable with the contract address
26     constructor(address _etherStoreAddress) {
27         etherStore = EtherStore(_etherStoreAddress);
28     }
29
30     function pwnEtherStore() public payable {
31         // attack to the nearest ether
32         require(msg.value >= 1 ether);
33         // send eth to the depositFunds() function
34         etherStore.depositFunds.value(1 ether)();
35         // start the magic
36         etherStore.withdrawFunds(1 ether);
37     }
38
39     function collectEther() public {
40         msg.sender.transfer(this.balance);
41     }
42
43     // fallback function - where the magic happens
44     function () payable {
45         if (etherStore.balance > 1 ether) {
46             etherStore.withdrawFunds(1 ether);
47         }
48     }
49 }
50
51 contract EtherStore {
52     // initialise the mutex
53     bool reEntrancyMutex = false;
54     uint256 public withdrawalLimit = 1 ether;
55     mapping(address => uint256) public lastWithdrawTime;
56     mapping(address => uint256) public balances;
57
58     function depositFunds() public payable {
59         balances[msg.sender] += msg.value;
60     }
61
62     function withdrawFunds (uint256 _weiToWithdraw) public {
63         require(!reEntrancyMutex);
64         require(balances[msg.sender] >= _weiToWithdraw);
65         // limit the withdrawal
66         require(_weiToWithdraw <= withdrawalLimit);
67         // limit the time allowed to withdraw
68         require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
69         balances[msg.sender] -= _weiToWithdraw;
70         lastWithdrawTime[msg.sender] = now;
71         // set the reEntrancy mutex before the external call
72         reEntrancyMutex = true;
73         msg.sender.transfer(_weiToWithdraw);
74         // release the mutex after the external call
75         reEntrancyMutex = false;
76     }
77 }
```

La **Ethereum Virtual Machine (EVM)** specifica i tipi a dimensione fissa per gli interi con un intervallo limitato di rappresentazione.

- Un **uint8** (2⁸) può memorizzare solo numeri nell'intervallo [0,255]. Il tentativo di memorizzare 256 in un uint8 risulterà in 0.

Se non si fa attenzione, possono sopravvenire problemi se l'input dell'utente non è controllato e vengono eseguiti calcoli che risultano in numeri che si trovano al di fuori dell'intervallo di rappresentazione del tipo della variabile che li memorizza:

- ***Underflow*** - sottraendo 1 ad una variabile uint8 che memorizza 0 si otterrà il numero 255;
 - ***Overflow*** - aggiungendo 257 a una variabile di tipo uint8 che attualmente ha un valore zero si otterrà il numero 1;
 - Aggiungendo $2^8 = 256$ a una variabile di tipo uint8 non ha alcun effetto di modifica del valore memorizzato.

Questi tipi di vulnerabilità consentono agli attaccanti di utilizzare in modo improprio il codice e creare flussi logici imprevisti.

Questo contratto è progettato per agire come un salvadanaio a tempo, in cui gli utenti possono depositare Ether e il deposito sarà bloccato per almeno una settimana. L'utente può estendere il tempo per più di una settimana se lo desidera, ma una volta depositato, l'utente può essere certo che il proprio Ether sia bloccato in modo sicuro per almeno una settimana.

- [riga 4] Il valore di questa variabile ha visibilità pubblica;
 - [riga 12] Un attaccante può sfruttare l'overflow per aprire il salvadanaio. Può passare come argomento di *increaseLockTime()* il numero $2^{256} - \text{userLockTime}$, e la somma alla riga 12 causerebbe un overflow, resettando `lockTime[msg.sender]` a 0. L'attaccante può quindi chiamare semplicemente la funzione di ritiro per ottenere la ricompensa.

Nella funzione withdraw(), si controlla prima che il saldo sia maggiore di 0, si controlla che si è superato il tempo di lock, si fa il transfer (per evitare l'attacco di rientro) e poi si imposta la variabile a 0 in quanto si è trasferito tutto il saldo.

Questo contratto è un token che consente agli utenti di trasferire Ether mediante la funzione transfer(), ma presenta un errore.

- [righe 9-10] Si consideri un utente che non ha alcun saldo e invoca

la funzione transfer() con qualsiasi _value diverso da zero. L'istruzione require alla riga [9] viene superata perché balances[msg.sender] è zero (e un uint256) quindi la sottrazione con un qualsiasi importo positivo (escluso 2^{256}) risulterà in un numero positivo a causa dell'underflow. Ciò vale anche per la riga [9], dove al saldo verrà accreditato un numero positivo.

```
1  contract TimeLock {
2
3      mapping(address => uint) public balances;
4      mapping(address => uint) public lockTime;
5
6      function deposit() public payable {
7          balances[msg.sender] += msg.value;
8          lockTime[msg.sender] = now + 1 weeks;
9      }
10
11     function increaseLockTime(uint _secondsToIncrease) public {
12         lockTime[msg.sender] += _secondsToIncrease;
13     }
14
15     function withdraw() public {
16         require(balances[msg.sender] > 0);
17         require(now > lockTime[msg.sender]);
18         msg.sender.transfer(balances[msg.sender]);
19         balances[msg.sender] = 0;
20     }
21 }
22
23 pragma solidity ^0.4.18;
24
25 contract Token {
26
27     mapping(address => uint) balances;
28     int public totalSupply;
29
30     function Token(uint _initialSupply) {
31         balances[msg.sender] = totalSupply = _initialSupply;
32     }
33
34     function transfer(address _to, uint _value) public returns (bool) {
35         require(balances[msg.sender] - _value >= 0);
36         balances[msg.sender] -= _value;
37         balances[_to] += _value;
38         return true;
39     }
40
41     function balanceOf(address _owner) public constant returns (uint balance) {
42         return balances[_owner];
43     }
44 }
```

La soluzione per proteggersi dalle vulnerabilità di under/overflow consiste nell'usare librerie matematiche sicure come SafeMath di Open Zeppelin.

```
La soluzione per proteggere dalle vulnerabilità di divide, overview consists
1 library SafeMath {
2     function mul(uint256 a, uint256 b) internal pure returns (uint256) {
3         if (a == 0) {
4             return 0;
5         }
6         uint256 c = a * b;
7         assert(c / a == b);
8         return c;
9     }
10    function div(uint256 a, uint256 b) internal pure returns (uint256) {
11        // assert(b > 0); // Solidity automatically throws when dividing by 0
12        uint256 c = a / b;
13        // assert(a == b * c + a % b); // There is no case in which this doesn't hold
14        return c;
15    }
16    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
17        assert(b <= a);
18        return a - b;
19    }
20    function add(uint256 a, uint256 b) internal pure returns (uint256) {
21        uint256 c = a + b;
22        assert(c >= a);
23        return c;
24    }
25 }
```

```
26 contract TimeLock {
27     using SafeMath for uint; // use the library for uint type
28     mapping(address => uint256) public balances;
29     mapping(address => uint256) public lockTime;
30
31     function deposit() public payable {
32         balances[msg.sender] = balances[msg.sender].add(msg.value);
33         lockTime[msg.sender] = now.add(1 weeks);
34     }
35
36     function increaseLockTime(uint256 _secondsToIncrease) public {
37         lockTime[msg.sender] = lockTime[msg.sender].add(_secondsToIncrease);
38     }
39
40     function withdraw() public {
41         require(balances[msg.sender] > 0);
42         require(now > lockTime[msg.sender]);
43         balances[msg.sender] = 0;
44         msg.sender.transfer(balances[msg.sender]);
45     }
46 }
```

Quando una quantità di Ether viene inviato a un contratto, deve eseguire la funzione di fallback o un'altra funzione nel contratto. Ci sono 2 eccezioni:

1. Qualsiasi contratto può implementare la funzione `selfdestruct(address)`, che rimuove tutto il bytecode dall'indirizzo del contratto e invia tutti gli Ether lì memorizzati all'indirizzo specificato dal parametro. Se l'indirizzo specificato è di contratto, non viene chiamata nessuna delle sue funzioni (incluso il fallback). Qualsiasi attaccante può creare un contratto con una funzione `selfdestruct()`, inviargli Ether, chiamare `selfdestruct(target)` e forzare l'invio di Ether a un contratto target;
 2. Un utente può precaricare un nuovo contratto con Ether, cosicché dopo la sua creazione avrà un saldo diverso da zero.

Rappresenta un semplice gioco semplice (che causerebbe situazioni di race conditions) in cui i giocatori inviano 0,5 Ether al contratto nella speranza di essere il giocatore che raggiunge per primo uno dei tre traguardi. Il primo a raggiungere il traguardo può rivendicare una parte dell'Ether accumulato quando il gioco è finito. Il gioco termina quando viene raggiunto il traguardo finale (10 Ether) e gli utenti possono richiedere i loro premi.

La funzione `claimReward()`, controlla se si è arrivati a 10 (`finalMileStone`) e controlla se chi ha fatto la richiesta può fare il claim di qualcosa, se lo può fare si setta 0 il suo claim e gli si trasferisce quanto ha chiesto, ma c'è un errore.

- [righe 14/16/32] Il cattivo uso di `this.balance` nel codice può dare problemi se un malintenzionato potrebbe forzare il contratto mandando una piccola quantità di Ether tramite la funzione `selfdestruct()` per impedire a futuri giocatori di raggiungere un traguardo. In aggiunta, un attaccante più aggressivo potrebbe inviare forzatamente 10 Ether o più per spingere il saldo del contratto al di sopra del `finalMileStone`, così da bloccare per sempre tutti i premi nel contratto.

```

1  contract EtherGame {
2
3      uint public payoutMileStone1 = 3 ether;
4      uint public mileStone1Reward = 2 ether;
5      uint public payoutMileStone2 = 5 ether;
6      uint public mileStone2Reward = 3 ether;
7      uint public finalMileStone = 10 ether;
8      uint public finalReward = 5 ether;
9
10
11     mapping(address => uint) redeemableEther;
12
13     // users pay 0.5 ether. At specific milestones, credit their accounts
14     function play() public payable {
15         require(msg.value == 0.5 ether); // each play is 0.5 ether
16         uint currentBalance = this.balance + msg.value;
17         // ensure no players after the game as finished
18         require(currentBalance <= finalMileStone);
19         // if at a milestone credit the players account
20         if (currentBalance == payoutMileStone1) {
21             redeemableEther[msg.sender] += mileStone1Reward;
22         }
23         else if (currentBalance == payoutMileStone2) {
24             redeemableEther[msg.sender] += mileStone2Reward;
25         }
26         else if (currentBalance == finalMileStone ) {
27             redeemableEther[msg.sender] += finalReward;
28         }
29         return;
30     }
31
32     function claimReward() public {
33         // ensure the game is complete
34         require(this.balance == finalMileStone);
35         // ensure there is a reward to give
36         require(redeemableEther[msg.sender] > 0);
37         redeemableEther[msg.sender] = 0;
38         msg.sender.transfer(redeemableEther[msg.sender]);
39     }

```

La logica implementata nello smart contract, quando possibile, dovrebbe evitare di dipendere dai valori esatti del saldo del contratto perché può essere manipolato artificialmente.

- [righe 9/15/27/33] Una nuova variabile, `depositedWei` è stata introdotta per tenere traccia dell'Ether depositato. Si noti che non abbiamo più alcun riferimento a `this.balance`.

```

1  contract EtherGame {
2
3      uint public payoutMileStone1 = 3 ether;
4      uint public mileStone1Reward = 2 ether;
5      uint public payoutMileStone2 = 5 ether;
6      uint public mileStone2Reward = 3 ether;
7      uint public finalMileStone = 10 ether;
8      uint public finalReward = 5 ether;
9      uint public depositedWei;
10
11     mapping (address => uint) redeemableEther;
12
13     function play() public payable {
14         require(msg.value == 0.5 ether);
15         uint currentBalance = depositedWei + msg.value;
16         // ensure no players after the game as finished
17         require(currentBalance <= finalMileStone);
18         if (currentBalance == payoutMileStone1) {
19             redeemableEther[msg.sender] += mileStone1Reward;
20         }
21         else if (currentBalance == payoutMileStone2) {
22             redeemableEther[msg.sender] += mileStone2Reward;
23         }
24         else if (currentBalance == finalMileStone ) {
25             redeemableEther[msg.sender] += finalReward;
26         }
27         depositedWei += msg.value;
28         return;
29     }
30
31     function claimReward() public {
32         // ensure the game is complete
33         require(depositedWei == finalMileStone);
34         // ensure there is a reward to give
35         require(redeemableEther[msg.sender] > 0);
36         redeemableEther[msg.sender] = 0;
37         msg.sender.transfer(redeemableEther[msg.sender]);
38     }
39 }

```

`CALL` e `DELEGATECALL` sono utili per modularizzare il codice degli smart contracts. Questi due codici operativi realizzano la chiamata di codice esterno ma hanno sostanziali differenze:

1. Le chiamate gestite dall'opcode `CALL` implicano che il codice viene eseguito nel contesto del contratto/funzione esterna. Ad esempio, quando D invoca `CALL` su una funzione f di E, `msg.sender` vista da f è D.
2. Il codice operativo `DELEGATECALL` si differenzia per il fatto che il codice viene eseguito nel contesto del contratto chiamante. Ad esempio, quando D (la cui funzione è stata invocata dall'utente/contratto C) invoca `DELEGATECALL` su f di E, `msg.sender` vista da f è C.

Queste due opcode consentono l'implementazione di librerie in cui gli sviluppatori possono creare codice riutilizzabile per contratti futuri. L'uso di `DELEGATECALL` può portare a un'esecuzione imprevista del codice e vulnerabilità.

Preservando il contesto, DELEGATECALL non rende facile la creazione di librerie prive di vulnerabilità. Sebbene sicuro e privo di vulnerabilità, il codice eseguito nel contesto di un'altra applicazione può determinare l'insorgere di nuove vulnerabilità.

La libreria *FibonacciLib* genera la sequenza di Fibonacci, e offre una funzione per ottenere il numero di posizione ennesima nella successione.

Il contratto *FibonacciBalance* consente a un partecipante di ritirare l'Ether dal contratto, con la quantità pari al numero di Fibonacci corrispondente all'ordine di prelievo del partecipante.

- [righe 9/19 in *FibonacciBalance*] Selettore di funzione e contiene i primi 4 byte dell'hash Keccak (SHA-3) della stringa "setFibonacci (uint256)", e viene passata come primo argomento di DELEGATECALL.
- [righe 12-14 in *FibonacciBalance*] L'indirizzo del contratto esterno da chiamare è passato mediante il costruttore. Non sussiste alcun controllo se il contratto passato è quello corretto, e ciò può dare problemi. Inoltre, i parametri di input per gli smart contract sono visibili sulla blockchain e rilevare scelte di implementazione. Una soluzione consiste nell'usare la parola chiave new per creare contratti:

```
constructor() public payable {
    fibonacciLibrary = new FibonacciLib();
}
```

Ma è presente un errore:

- [righe 6/24-26 in *FibonacciBalance* e riga 4 in *FibonacciLib*] Entrambi hanno la variabile start, e la funzione di fallback in *FibonacciBalance* consente di passare tutte le chiamate al contratto di libreria, e potenzialmente chiamare *setStart()*.

Le variabili di stato o di archiviazione (variabili che persistono nelle singole transazioni) vengono inserite in slot dello stack in modo sequenziale man mano che vengono dichiarate nel contratto.

- [righe 4/5 in *FibonacciLib*] La prima variabile è start, e viene memorizzata allo slot[0]. La seconda variabile, calculatedFibNumber, viene posizionata nel successivo slot, slot[1]. La funzione *setFibonacci()* pone calculatedFibNumber pari a *fibonacci(n)*, ovvero pone slot[1] pari al ritorno di *fibonacci(n)*.
- [righe 2-7 in *FibonacciBalance*] slot [0] corrisponde all'indirizzo *fibonacciLibrary* e slot[1] corrisponde a *calculatedFibNumber*. Qui è presente la vulnerabilità siccome DELEGATECALL preserva il contesto del contratto chiamante.
- [riga 19] La funzione *setFibonacci()* viene chiamata, modificando lo slot[1], come previsto. Tuttavia, la variabile start nel contratto *FibonacciLib* si trova a slot [0], che nel contesto del chiamante contienete l'indirizzo di *fibonacciLibrary*. Ciò significa che la funzione *fibonacci()* darà un risultato inaspettato.

Il contratto *FibonacciBalance* consente agli utenti di chiamare tutte le funzioni di *FibonacciLibrary* tramite la funzione di fallback, tra cui la funzione *setStart()*. Tale funzione è richiamata mediante DELEGATE CALL [riga 25], che preserva il contesto; quindi, questa funzione consente a chiunque di modificare o impostare il valore in slot[0], ovvero l'indirizzo per *FibonacciLibrary*. Pertanto, un utente malintenzionato potrebbe creare un contratto dannoso, e chiamare *setStart()* passando l'indirizzo del contratto dannoso. Questo farà sì che ogni volta che un utente chiama *withdraw()* o la funzione di fallback, verrà eseguito il contratto dannoso (che può rubare l'intero saldo del contratto).

Per ovviare a questa vulnerabilità, Solidity fornisce la parola chiave *library* per l'implementazione dei contratti di libreria. Ciò garantisce che il contratto con library sia senza stato (per attenuare le complessità del contesto di archiviazione) e non autodistruttibile.

Le funzioni in Solidity hanno **specificatori di visibilità**: una funzione può essere chiamata esternamente dagli utenti, o da altri contratti derivati, o solo internamente o solo esternamente.

- *Private*, si limita la visibilità al contratto;
- *Internal*, si limita la visibilità al contratto alle derivate del contratto;
- *External*, si limita la visibilità solo ad un altro contratto;
- *Public*, si limita la visibilità a tutti e tre.

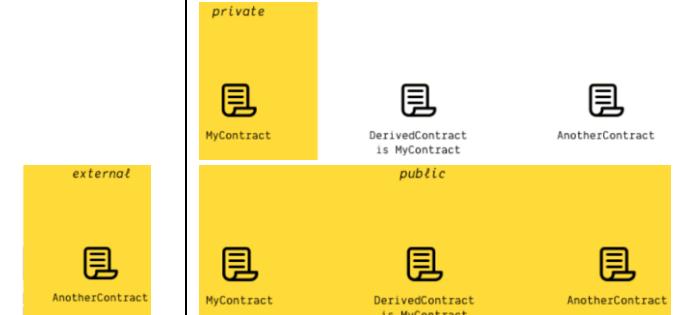
```
1 // library contract - calculates fibonacci-like numbers;
2 contract FibonacciLib {
3     // initializing the standard fibonacci sequence;
4     uint public start;
5     uint public calculatedFibNumber;
6     // modify the zeroth number in the sequence
7     function setStart(uint _start) public {
8         start = _start;
9     }
10    function setFibonacci(uint n) public {
11        calculatedFibNumber = fibonacci(n);
12    }
13    function fibonacci(uint n) internal returns (uint) {
14        if (n == 0) return start;
15        else if (n == 1) return start + 1;
16        else return fibonacci(n - 1) + fibonacci(n - 2);
17    }
18 }
```



```
1 contract FibonacciBalance {
2     address public fibonacciLibrary;
3     // the current fibonacci number to withdraw
4     uint public calculatedFibNumber;
5     // the starting fibonacci sequence number
6     uint public start = 3;
7     uint public withdrawalCounter;
8     // the fibonacci function selector
9     bytes4 constant fibSig = bytes4(sha3("setFibonacci(uint256)"));
10
11    // constructor - loads the contract with ether
12    constructor(address _fibonacciLibrary) public payable {
13        fibonacciLibrary = _fibonacciLibrary;
14    }
15    function withdraw() {
16        withdrawalCounter += 1;
17        // calculate the fibonacci number for the current withdrawal user
18        // this sets calculatedFibNumber
19        require(fibonacciLibrary.delegatecall(fibSig, withdrawalCounter));
20        msg.sender.transfer(calculatedFibNumber * 1 ether);
21    }
22
23    // allow users to call fibonacci library functions
24    function() public {
25        require(fibonacciLibrary.delegatecall(msg.data));
26    }
27 }
```



```
1 contract Attack {
2     uint storageSlot0; // corresponds to fibonacciLibrary
3     uint storageSlot1; // corresponds to calculatedFibNumber
4
5     // fallback - this will run if a specified function is not found
6     function() public {
7         storageSlot1 = 0; // we set calculatedFibNumber to 0, so that if withd
8         // is called we don't send out any ether.
9         <attacker_address>.transfer(this.balance); // we take all the ether
10    }
11 }
```



L'impostazione predefinita delle funzioni è la visibilità pubblica. Un uso errato degli specificatori di visibilità può portare ad alcune vulnerabilità.

Questo semplice contratto è progettato per fungere da gioco per indovinare gli indirizzi. Per vincere il saldo del contratto, un utente deve generare un indirizzo Ethereum i cui ultimi 8 caratteri esadecimalesi sono 0.

- [righe 9-11] Purtroppo non è stata specificata la visibilità delle funzioni, e chiunque può invocare la funzione `_sendWinnings()`, che è pubblica, e ottenere la taglia non rispettando il gioco.

È buona norma specificare sempre la visibilità di tutte le funzioni in un contratto, anche se intenzionalmente pubbliche.

```

1  contract HashForEther {
2
3      function withdrawWinnings() {
4          // Winner if the last 8 hex characters of the address are 0.
5          require(uint32(msg.sender) == 0);
6          _sendWinnings();
7      }
8
9      function _sendWinnings() {
10         msg.sender.transfer(this.balance);
11     }
12 }
```

Le funzioni `call()` e `send()` restituiscono un valore booleano che indica se la chiamata è riuscita o meno.

Un errore comune si verifica quando il valore restituito non è controllato.

- [riga 9] Viene utilizzata una `send()` senza controllarne la risposta. Un vincitore la cui transazione fallisce (o esaurendo il gas, o perché invoca `throws` intenzionalmente nella fallback o tramite un `call` stack depth attack, impossibile su recenti EVM) consente a `payedOut` di essere impostato su `true` (indipendentemente dal fatto che ether sia stato inviato o no). In questo caso, chiunque può ritirare la vincita tramite la funzione `drawLeftOver()`. Quando possibile, si deve preferire `transfer()` invece di `send()`, poiché quest'ultima propaga errori all'intero verso il chiamante disfacendo la transazione. Se `send()` è richiesto, assicurati sempre di controllare il valore restituito, dove è importante che l'errore venga gestito nel contratto senza annullare tutte le modifiche di stato.

```

1  contract Lotto {
2      bool public payedOut = false;
3      address public winner;
4      uint public winAmount;
5
6      // ... extra functionality here
7      function sendToWinner() public {
8          require(!payedOut);
9          winner.send(winAmount);
10         payedOut = true;
11     }
12
13     function withdrawLeftOver() public {
14         require(payedOut);
15         msg.sender.send(this.balance);
16     }
17 }
```

La combinazione di chiamate esterne ad altri contratti e la natura multiutente della blockchain sottostante dà origine a una serie di potenziali insidie per cui gli utenti competono per l'esecuzione del codice ottenendo esiti imprevisti. Le transazioni sono ordinate dai miners in base al loro `gasPrice`, e ciò può essere usato a proprio vantaggio dagli attaccanti.

- [riga 6] L'utente che trova la stringa per l'hash sha3 nella variabile `hash` può inviare la soluzione e recuperare il 1000 Ether.

Ipotizziamo che un utente capisce che la soluzione è Ethereum, e invoca di conseguenza `solve()`. Un utente malintenzionato può aver controllato il pool di transazioni e vedono questa soluzione, invia una transazione equivalente con un prezzo del gas molto più alto rispetto a quella originale. A causa del gas più elevato, la seconda transazione sarà accettata prima - front-running attack.

L'attaccante otterrà i 1000 Ether e l'utente onesto nulla.

```

1  contract FindThisHash {
2      bytes32 constant public hash = 0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44
3
4      constructor() public payable {} // load with ether
5
6      function solve(string solution) public {
7          // If you can find the pre image of the hash, receive 1000 ether
8          require(hash == sha3(solution));
9          msg.sender.transfer(1000 ether);
10     }
11 }
```

Tra le possibili soluzioni, uno robusto è di utilizzare uno schema di ***commit-reveal***, quando possibile:

- Gli utenti inviano transazioni con informazioni nascoste (in genere un hash):

```
function commit(bytes32 commitment) public payable {}
```

- Dopo che la transazione è stata inclusa in un blocco, l'utente invia un'altra transazione rivelando i dati che sono stati inviati:

```
function reveal(Choice choice, bytes32 blindingFactor) public {}
```

Questo metodo impedisce sia ai miner che agli utenti di effettuare transazioni anticipate poiché non possono determinare il contenuto della transazione.

Esistono vari modi in cui un contratto può diventare indisponibile mediante attacchi DoS:

- Ciclo lungo mappings o array manipolati esternamente:

[righe 8-11/15] Si noti che il ciclo in questo contratto viene eseguito su un array che può essere modificato arbitrariamente. Un utente malintenzionato può creare molti account in modo tale che il gas richiesto superi il limite del gas di blocco, rendendo sostanzialmente inutilizzabile la funzione `distribute()`. La soluzione consiste nell'evitare di ciclare su strutture manipolate esternamente, oppure nell'applicare il `withdraw` al pattern: ogni utente deve invocare una funzione isolata che richiede il trasferimento del token.

```

1  contract DistributeTokens {
2      address public owner; // gets set somewhere
3      address[] investors; // array of investors
4      uint[] investorTokens; // the amount of tokens each investor gets
5
6      // ... extra functionality, including transfertoken()
7
8      function invest() public payable {
9          investors.push(msg.sender);
10         investorTokens.push(msg.value * 5); // 5 times the wei sent
11     }
12
13     function distribute() public {
14         require(msg.sender == owner); // only owner
15         for(uint i = 0; i < investors.length; i++) {
16             // here transferToken(to,amount) transfers "amount" of tokens to the address
17             transferToken(investors[i],investorTokens[i]);
18         }
19     }
20 }
```

- L'owner del contratto deve eseguire alcune attività affinché il contratto possa passare allo stato successivo. Se perde le proprie chiavi private o diventa inattivo, l'intero contratto diventa inutilizzabile;
- A volte i contratti passano a un nuovo stato se ricevono Ether o un input da una fonte esterna. Questi modelli possono portare ad attacchi DOS, quando la chiamata esterna fallisce o viene impedita per motivi esterni.

Una soluzione agli ultimi due punti consiste nell'impostare un timelock: `require(<cond> || now > unlockTime)` che consente a qualsiasi utente di finalizzare dopo un periodo di tempo in cui la condizione non si è verificata.

I timestamp dei blocchi sono utilizzati per una varietà di applicazioni, e i miners hanno la capacità di modificare leggermente i timestamp, il che può rivelarsi piuttosto pericoloso se i timestamp vengono utilizzati in modo errato negli smart contract.

Questo contratto si comporta come una semplice lotteria. Una transazione per blocco può scommettere 10 ether per avere la possibilità di vincere il saldo del contratto. Il presupposto è che block.timestamp sia distribuito uniformemente sulle ultime due cifre. Se così fosse, ci sarebbe una probabilità di 1/15 di vincere questa lotteria.

Il margine d'azione è però limitato: i timestamp aumentano in modo monotono e non si possono scegliere valori arbitrari o non troppo lontano nel futuro poiché il blocco risulterebbe rifiutato dalla rete.

- [righe 11] Se nel contratto è presente una quantità sufficiente di Ether, un minatore che risolve un blocco è incentivato a scegliere un timestamp tale che block.timestamp o now modulo 15 sia 0. In tal modo può vincere il saldo del contratto insieme alla ricompensa per il mining del blocco. Poiché c'è solo partecipante alla lotteria per blocco, si verifica il caso del front-running attack.

A volte è necessaria una logica sensibile al tempo, ma si consiglia di utilizzare block.number, che è più sicuro poiché i minatori non sono in grado di manipolarlo facilmente.

I costruttori sono funzioni speciali che spesso svolgono compiti critici e privilegiati durante l'inizializzazione dei contratti. Prima di Solidity v0.4.22, i costruttori erano funzioni che avevano lo stesso nome del contratto che li conteneva. Quando un nome di contratto viene modificato in fase di sviluppo, ma il nome del costruttore non viene modificato, questo diventa una normale funzione richiamabile.

Il nome del costruttore non corrisponde con quello del contratto, e quest'ultimo può essere invocato da qualunque per configurarsi come owner.

Questo problema è stato risolto nel compilatore Solidity nella versione 0.4.22, introducendo una parola chiave constructor che specifica il costruttore, invece di richiedere che il nome della funzione corrisponda al nome del contratto.

```
1 contract Roulette {
2     uint public pastBlockTime; // Forces one bet per block
3
4     constructor() public payable {} // initially fund contract
5
6     // fallback function used to make a bet
7     function () public payable {
8         require(msg.value == 10 ether); // must send 10 ether to play
9         require(now != pastBlockTime); // only 1 transaction per block
10        pastBlockTime = now;
11        if(now % 15 == 0) { // winner
12            msg.sender.transfer(this.balance);
13        }
14    }
15 }
```

EVM archivia i dati in tre possibili locazioni:

- **Memory**, rappresenta un'area di memoria per le variabili locali alle funzioni, il cui ciclo di vita è limitato alla durata di una chiamata esterna di funzione;
- **Storage**, rappresenta un'area di memoria per le variabili di stato, il cui ciclo di vita è pari a quello del contratto;
- **Calldata**, rappresenta una porzione di area speciale, non modificabile, non persistente, per gli argomenti di funzione.

Capire esattamente come questo viene fatto e i tipi predefiniti (alcuni indicano dei valori, altri definiscono dei riferimenti) per le variabili locali delle funzioni è altamente raccomandato quando si sviluppano i contratti. Questo perché è possibile produrre contratti vulnerabili inizializzando in modo inappropriato le variabili.

Le variabili locali all'interno delle funzioni vengono allocate per impostazione predefinita in storage o memory a seconda del loro tipo. Se non inizializzate possono causare dei comportamenti inattesi.

Quando il contratto è sbloccato, consente a chiunque di registrare un nome (come hash bytes32) e mappare quel nome a un indirizzo.

Sfortunatamente, è inizialmente bloccato e la richiesta alla linea [19] impedisce a register() di aggiungere record di nomi.

I tipi struct definiscono riferimenti e newRecord non è inizializzato con new.

Solidity inizializza per impostazione predefinita istanze di tipi di dati complessi, come gli struct, a storage quando vengono usati come variabili locali. newRecord punta a storage e newRecord.name a slot[0], ovvero ad unlocked, modificabile direttamente mediante il parametro _name della funzione register().

Il compilatore Solidity genera dei warning quando trova delle variabili non inizializzate; quindi, gli sviluppatori dovrebbero prestare molta attenzione a questi avvisi. L'attuale versione di mist (0.10) non consente la compilazione di questi contratti.

```
1 contract OwnerWallet {
2     address public owner;
3     //constructor
4     function ownerWallet(address _owner) public {
5         owner = _owner;
6     }
7
8     // fallback, Collect ether.
9     function () payable {}
10
11    function withdraw() public {
12        require(msg.sender == owner);
13        msg.sender.transfer(this.balance);
14    }
15 }
```

Solidity ha una variabile globale, tx.origin che attraversa l'intero stack di chiamate e restituisce l'indirizzo dell'account che ha originariamente inviato la chiamata (o la transazione). L'utilizzo di questa variabile per l'autenticazione negli smart contract lascia il contratto vulnerabile a un attacco phishing.

Un attaccante può convincere il proprietario del contratto Phishable di inviargli dell'Ether su un indirizzo, senza riconoscere che corrisponde a AttackContract. L'effetto è di chiamare la funzione di fallback, che invoca quella del contratto vittima.

```

1 import "Phishable.sol";
2 contract AttackContract {
3
4     Phishable phishableContract;
5     address attacker; // The attackers address to receive funds.
6     constructor (Phishable _phishableContract, address _attackerAddress) {
7         phishableContract = _phishableContract;
8         attacker = _attackerAddress;
9     }
10
11     function () {
12         phishableContract.withdrawAll(attacker);
13     }
14 }
```

tx.origin non deve essere utilizzato per l'autorizzazione negli smart contract, ma per altri scopi. Ad esempio, se si volesse negare ai contratti esterni di chiamare il contratto corrente, si potrebbe implementare una require (tx.origin == msg.sender).

```

1 contract Phishable {
2     address public owner;
3
4     constructor (address _owner) {
5         owner = _owner;
6     }
7
8     function () public payable {} // collect ether
9     function withdrawAll(address _recipient) public {
10        require(tx.origin == owner);
11        _recipient.transfer(this.balance);
12    }
13 }
```

VULNERABILITÀ NEI CHAINCODE (HYPERLEDGER FABRIC):

Anche i chaincode su **Hyperledger Fabric** hanno delle vulnerabilità. Poiché il chaincode è eseguito da nodi distribuiti e indipendenti, la determinatezza della logica di business è una delle principali preoccupazioni. Se il chaincode non è deterministico, rende incoerente il risultato dell'approvazione di diversi peers. Alcune piattaforme hanno linguaggi come Solidity che non hanno sorgenti di causalità nell'esecuzione. Hyperledger Fabric impiega **linguaggi general-purpose** che presentano tali sorgenti. Gli utenti prevedono un numero generato in modo casuale e, se la previsione è corretta, l'utente riceve un premio. In ogni endorsement peer il numero da indovinare è differente e l'esito della lotteria è differente.

Go ha un ricco supporto alla concorrenza utilizzando **goroutine** (una funzione che può essere eseguita contemporaneamente ad altre funzioni) e channel (un modo per due goroutine di comunicare tra loro e sincronizzare la loro esecuzione).

La sintassi delle goroutine è invocare una funzione con la parola chiave **go**:

```

package main

import (
    "fmt"
    "time"
    "math/rand"
)

func f(n int) {
    for i := 0; i < 10; i++ {
        fmt.Println(n, ":", i)
        amt := time.Duration(rand.Intn(250))
        time.Sleep(time.Millisecond * amt)
    }
}

func main() {
    for i := 0; i < 10; i++ {
        go f(i)
    }
    var input string
    fmt.Scanln(&input)
}
```

```

package main

import (
    "fmt"
    "time"
)

func pinger(c chan string) {
    for i := 0; ; i++ {
        c <- "ping"
    }
}

func printer(c chan string) {
    for {
        msg := <- c
        fmt.Println(msg)
        time.Sleep(time.Second * 1)
    }
}

func main() {
    var c chan string = make(chan string)

    go pinger(c)
    go printer(c)

    var input string
    fmt.Scanln(&input)
}
```

Un canale è indicato dalla parola chiave **chan** seguita dal tipo di dati scambiabili sul canale. L'operatore **<-** viene utilizzato per inviare e ricevere messaggi sul canale.

Ipotizziamo un chaincode che deve eseguire un compito computazionale pesante e restituire una risposta a un richiedente in modo sincrono, come una sequenza di due funzioni:

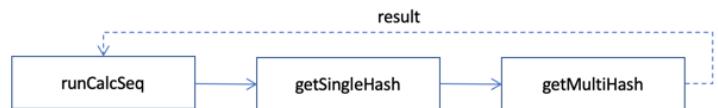
- `getSingleHash()`, che prende dei dati come input e calcola l'output come $\text{crc32}(\text{data}) + \sim + \text{crc32}(\text{md5}(\text{data}))$, dove $+$ significa concatenazione di stringhe;

- `getMultiHash()`, che restituisce l'output come $\text{crc32}(i + \text{data})$, dove i è un numero di iterazione compreso tra 0 e 6.

```
// gets a crc32(data) + "~" + crc32(md5(data)) value
func getSingleHash(data string) string {
    var result string
    crc32Hash1 := getCrc32(data)
    md5Hash := getMd5(data)
    crc32Hash2 := getCrc32(md5Hash)
    result = crc32Hash1 + "~" + crc32Hash2
    return result
}

// gets a crc32(i + data) where i = 0..5
func getMultiHash(data string) string {
    var result string
    for i := 0; i < 6; i++ {
        sI := strconv.Itoa(i)
        result += getCrc32(sI + data)
    }
    return result
}
```

Consideriamo una implementazione sequenziale:



Il codice sarà racchiuso in un metodo Hyperledger invocabile come segue:

```
func runCalcSeq(stub shim.ChaincodeStubInterface, args []string) peer.Response {
    var resAsBytes []byte
    var hash1, hash2 string

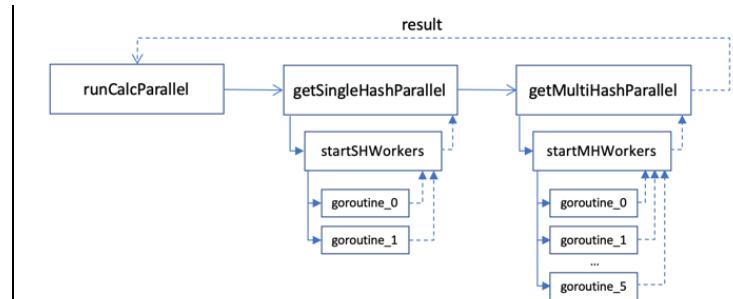
    for _, val := range args {
        hash1 = getSingleHash(val)
        hash2 = getMultiHash(hash1)
    }
    resAsBytes = []byte(hash2)

    return shim.Success(resAsBytes)
}
```

La funzione sopra impiegherà 8.02 secondi per essere eseguita.

```
==== RUN TestSeq
Call: runCalcSeq ( 0 )
Code execution time for 'runCalcSeq' is 8.027733723s
RetCode: 200
RetMsg:
Payload: 29568666068035183841425683795340791879727309630931025356555
Received result:
29568666068035183841425683795340791879727309630931025356555
--- PASS: TestSeq (8.03s)
main_test.go:20: The code execution time is: 8.027805802s
PASS
ok   GoBasicHLpipelines/src  8.810s
```

Ora consideriamo una implementazione parallela:



Un canale chiamato "out" viene creato e passato alla funzione `scheduler`, che istanzia due goroutine per il calcolo delle due porzioni del risultato. Sull'ultima stringa restituiamo solo un valore fornito dal canale.

```
// gets a crc32(data) + "~" + crc32(md5(data)) value in parallel mode
func getSingleHashParallel(in string) string {
    out := make(chan string)

    go func(data string) {
        startSingleHashWorkers(data, out)
    }(in)

    return <-out
}
```

Si usa un oggetto di sincronizzazione WaitGroup per attendere che entrambe le goroutine finiscano attendendo su wg.Wait(). Ogni istanza di WaitGroup ha un contatore interno, che viene aumentato dal metodo wg.Add() e diminuito dal metodo wg.Done(). La parola chiave "defer" ritardare l'esecuzione della funzione o del metodo o di un metodo anonimo fino al ritorno delle funzioni vicine.

```
func startSingleHashWorkers(data string, out chan<- string) {
    wg := &sync.WaitGroup{}
    wg.Add(2)

    var left, right string

    go func() {
        defer wg.Done()
        left = getCrc32(data)
    }()

    go func() {
        defer wg.Done()
        hash := getMd5(data)
        right = getCrc32(hash)
    }()

    wg.Wait()
    result := left + "~" + right
    out <- result
}
```

Il tempo di esecuzione in questo caso è 2.02 secondi.

NOTA: Se un programma concorrente non viene gestito in modo appropriato, si verificherà facilmente un problema di race condition.

```
==== RUN TestParallel
Call: runCalcParallel ( 0 )
RetCode: 200
RetMsg:
Payload: 29568666068035183841425683795340791879727309630931025356555
Received result:
29568666068035183841425683795340791879727309630931025356555
--- PASS: TestParallel (2.02s)
    main_test.go:35: The code execution time is: 2.021541811s
PASS
```

Una delle problematiche maggiori di Fabric è che non supporta la semantica Read-Your-Write, esempio se all'interno di un chaincode si fa un modifica alla blockchain, questa modifica nel chaincode non è visibile. In realtà, l'esecuzione del chaincode non altera la blockchain ma la simula. Fondamentalmente, le modifiche non vengono viste finché il chaincode viene approvato dall'endorsement peer per poi passare per tutto il processo di consistenza.

L'aggiornamento dello stato sulla catena è un metodo asincrono:
`<async> putState(key, value)`

che produce una serie di modifiche che verranno applicate allo stato ma non lo sono in maniera sincrona con il codice.

L'implicazione è che gli aggiornamenti allo stato sicuramente non sono riletti.

```
func addToAccount(stub ..., account *string, amount int) {
    balance, _ := stub.GetState(*account)
    stub.PutState(*account, balance + amount)
}

func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface)
pb.Response {

    args := stub.GetArgs()
    addToAccount(stub, "Tobias", args[0])

    ...
    addToAccount(stub, "Tobias", args[1])

    return shim.Success(nil)
}
```

Solo il primo stato finale è corretto:



Gli sviluppatori devono implementare due metodi (**Init()** e **Invoke()**) per soddisfare l'interfaccia chaincode in Go. Quando i metodi vengono implementati all'interno di una struttura, gli sviluppatori possono definirne dei campi che sono utilizzati con uno stato globale. Tuttavia, poiché ogni peer non esegue ogni transazione, lo stato non mantiene lo stesso valore tra i peer.

È sempre sbagliato usare variabili globali in un chaincode.

```
type BadChaincode struct {
    globalValue string // this is a risk
}

func (t *BadChaincode) Invoke (stub shim.ChaincodeStubInterface) peer.Response {
    t.globalValue = args[0]
    return shim.Success([]byte("success"))
}
```

BLOCKCHAIN VS GDPR:

Per quanto la blockchain dia forti garanzie di sicurezza, non sono più isolati gli episodi di hackeraggio ai danni degli Exchange Coin-to-Coin (scambio di criptovalute).

Ad eccezione di un hardware wallet, violabili mediante un accesso fisico, le chiavi private per accedere ai wallet possono essere soggette ad attacchi, soprattutto phishing, alla stregua delle altre password.

Il Regolamento Generale sulla Protezione dei Dati (GDPR) è un regolamento dell'Unione europea in materia di trattamento dei dati personali e di privacy. Nel contesto della privacy e del GDPR, le blockchain presentano varie sfide: Come garantire i diritti del titolare dei dati immessi nella catena, fino al diritto di cancellazione se i dati sono immutabili? Chi risponde del trattamento dei dati in una rete di registri replicati? Il trattamento dei dati è tecnicamente solo quello che si conclude con la chiusura del primo blocco o è continuo?

Circa l'adeguamento della blockchain al Regolamento UE 2016/679 o GDPR, lo scoglio apparentemente insormontabile è garantire i diritti dell'interessato, disciplinati al capo III, negli artt. 12 – 23. La peculiarità intrinseca della blockchain della non modificabilità e cancellazione di garantire l'integrità dei dati e aumentare la fiducia nella rete risulta essere in forte contrasto con requisiti legali come il "diritto di rettifica" e il "diritto all'oblio" sanciti nell'artt. 16 e 17.

Soluzioni al diritto di oblio potrebbero essere:

- la crittografia dei dati personali e la successiva eliminazione delle corrispettive chiavi, lasciando su blockchain solo i dati indecifrabi;
- mediante l'uso dei cosiddetti modelli di memoria "fuori catena": registrare su blockchain solo un "collegamento", lasciando il dato vero e proprio al di fuori del libro mastro;
- garantire il diritto all'oblio mediante una corretta anonimizzazione.

È in corso un dibattito sul fatto che i dati archiviati in una blockchain, come chiavi pubbliche e dati transazionali, si qualifichino come dati personali ai fini del GDPR, e se i dati personali che sono stati crittografati o sottoposti a hash si qualificano ancora come dati personali.

Le blockchain sono un particolare tipo di database di sola aggiunta che crescono continuamente man mano che vengono aggiunti nuovi dati, che vengono replicati su molti computer diversi. Ciò pone problemi circa la minimizzazione dei dati e limitazione delle finalità.

Cosa si intende per "scopo" del trattamento dei dati personali nel contesto della blockchain? Si include solo la transazione iniziale o comprende anche il trattamento continuo dei dati personali (come la sua memorizzazione e il suo utilizzo per consenso) una volta che è stato messo in catena.

In caso di controversie, quali leggi devono essere applicate e di chi è la giurisdizione? In situazioni in cui non è possibile identificare l'entità di elaborazione dei dati personali e il luogo in cui i dati vengono elaborati (probabilmente ci sono tante di queste entità e luoghi quanti sono i nodi di rete), è difficile individuare la giurisdizione cui dovrebbe competere una eventuale valutazione legale del trattamento dei dati (ossia, in parole semplici, la legge nazionale applicabile).

Il GDPR introduce il **Data Protection Officer (DPO)**, che deve assistere colui che li controlla o li gestisce al fine di verificare l'osservanza interna al regolamento. In una Blockchain, chi è il responsabile del trattamento dei dati personali? Nel GDPR, il responsabile del trattamento determina le finalità e i mezzi del trattamento dei dati personali. Può esistere una simile entità nel contesto di una Blockchain distribuita?

Il modello di governance decentralizzato di Blockchain e la molteplicità degli attori coinvolti rendono più ostica la definizione dei ruoli.

- I partecipanti, che scrivono sul canale e inviano dati alla convalida dei miners, possono essere considerati i responsabili.
- Qualora un gruppo di partecipanti decida di attuare un trattamento con uno scopo comune, il responsabile va identificato tra loro. In caso contrario, tutti i partecipanti dovrebbero essere considerati come contitolari del trattamento (ex art. 26 GDPR).

Sicuramente non possono essere ritenuti titolari:

- i miner, in quanto il loro operato è circoscritto alla convalida delle transazioni, senza avere voce in merito all'oggetto di queste transazioni, non determinando né le finalità né i mezzi da attuare;
- le persone fisiche che immettono dati personali nella blockchain, al di fuori da un'attività professionale o commerciale (cioè quando l'attività è esclusivamente personale).

Il responsabile del trattamento andrebbe ricercato tra:

- gli sviluppatori degli Smart Contracts, in quanto trattano i dati personali per conto del titolare;
- i minatori, poiché eseguono le istruzioni del titolare quando verificano che la transazione soddisfano i criteri tecnici.

Entrambi dovrebbero quindi definire, con il titolare del trattamento, un contratto che specifichi gli obblighi di entrambe le parti e che incorpori le disposizioni dell'articolo 28 del GDPR.

Per quanto riguarda l'ambito degli obblighi di **Privacy by Design** (articolo 25), il titolare del trattamento deve pensare, in via preliminare, alla pertinenza della scelta di questa tecnologia per l'attuazione del suo trattamento.

Qualsiasi transazione sulla blockchain implica:

- l'invio di una richiesta a tutti i minatori blockchain per la convalida di una transazione (contenente potenzialmente dati personali);
- un aggiornamento della blockchain aggiungendo il nuovo blocco nella catena di blocchi a tutti i partecipanti.

In aggiunta, i partecipanti possono essere ubicati in paesi al di fuori dell'UE sollevando la questione della conformità con gli obblighi di trasferimento extra UE (capo V GDPR).

Per quanto riguarda l'identificabilità, la blockchain ha il vantaggio che ogni partecipante ha un identificativo costituito da una serie di caratteri alfanumerici apparentemente casuali e non identificabili con reali identità. Quanto ai dati aggiuntivi memorizzati sulla blockchain, nel caso in cui si tratti di dati personali, essi dovrebbero essere registrati preferibilmente in modalità crittografata.

È necessario determinare le finalità del trattamento e la valutazione d'impatto (**DPIA**), al fine di identificare come perseguitibile l'uso di blockchain pubbliche, private, con o senza cifratura del contenuto dei blocchi, così da dimostrare i rischi residui di una scelta tecnologica e la loro accettabilità. La blockchain non è una tecnologia ma è una classe di tecnologie e l'esame di compatibilità con il GDPR va effettuato caso per caso.

Attualmente, è allo studio una duplice linea d'azione per risolvere i problemi tra blockchain e GDPR:

- Le istituzioni dovrebbero condurre un'attività di orientamento regolatorio (una sorta di interpretazione autentica) di alcuni principi del GDPR per renderli applicabili alla tecnologia blockchain.
- I fornitori di servizi blockchain potrebbero studiare congiuntamente con le istituzioni dei codici di condotta e sistemi di certificazione delle catene a seconda dei settori di produzione nei quali la tecnologia è applicata.

Ciò sarebbe condotto in maniera similare al Cloud computing conduct code, ed è previsto dallo stesso GDPR (artt. 40 e 42).

6. TRUSTED EXECUTION ENVIRONMENT E BLOCKCHAIN

Il **Trusted Computing (TC)** ha lo scopo di garantire la sicurezza mediante la realizzazione di una sorta "cassaforte virtuale" attorno ai dati ed ai programmi. Andando a definire due porzioni del sistema Trusted, anche se un nodo viene compromesso non si presenta un comportamento bizzantino, e Non-trusted, un attaccante può comprometterla. Se dati o programmi esterni vogliono avere accesso a questa cassaforte devono ottenere la chiave dal sistema di TC e solamente dati e programmi autentificati possono disporre delle risorse del sistema: hard disk, RAM o CPU.

SOFTWARE FAULT ISOLATION:

SFI è una tecnica di strumentazione software a livello di codice macchina per stabilire domini di protezione logica all'interno di un processo. Si va ad isolare un modulo rispetto ad un altro, così se uno viene compromesso esternamente, il fault di quella parte non va ad intaccare il modulo protetto. Designa una regione di memoria per un componente non attendibile e fornisce istruzioni pericolose per limitare il suo accesso alla memoria. È indicato come **sandbox** del codice.

ESECUZIONE ISOLATA:

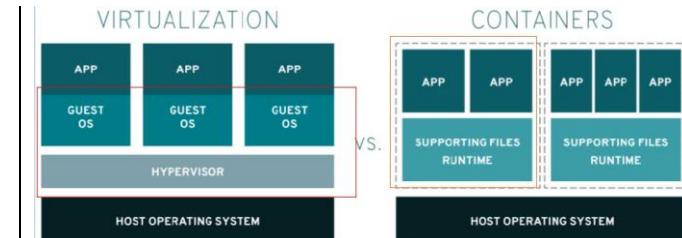
Isolare l'esecuzione del codice è uno degli approcci fondamentali per ottenere la sicurezza. La **virtualizzazione** può creare un ambiente di esecuzione isolato per l'esecuzione di strumenti difensivi. Gli approcci esistenti basati sulla virtualizzazione hanno limitazioni, tra cui:

1. Dipendenza da hypervisor che possono avere un'ampia **Trusted Computing Base (TCB)**. Il TCB di un sistema informatico è l'insieme di tutti i componenti hardware, firmware e/o software critici per la sua sicurezza: bug o vulnerabilità che si verificano all'interno del TCB potrebbero compromettere le proprietà di sicurezza dell'intero sistema.
2. Mancata gestione dell'hypervisor o dei rootkit del firmware. Un rootkit è una raccolta di software, in genere dannoso, progettato per consentire l'accesso a un computer o a un'area del suo software altrimenti non consentito.
3. Soffre di sovraccarico delle prestazioni del sistema (ad esempio, cambi di contesto da una VM a un hypervisor).

Un approccio diverso si realizza con soluzioni **container-based** come **Docker**.

Mentre una macchina virtuale astrae l'hardware, i container limitano il loro livello di astrazione al solo sistema operativo con la condivisione dello stesso sistema operativo, con il kernel, la connessione di rete e i file di base.

Le istanze vengono eseguite in uno spazio separato, garantendo una diminuzione di consumo della CPU e dell'overload associato.



Questa forma di virtualizzazione si basa sui **daemon** per gestire i container e le immagini delle applicazioni. Gli attacchi per compromettere gli ambienti di esecuzione sono tipicamente basati su vulnerabilità di questi demoni che presentano bug software o una cattiva configurazione:

1. **Deployment di immagini in container con codice dannoso:** le immagini dannose vengono prima inviate a un registro pubblico per essere estratte e distribuite sugli host Docker non protetti.
2. **Deployment di immagini benigne che scaricano payload dannosi in fase di esecuzione:** le immagini benigne vengono distribuite sugli host Docker, dove i payload dannosi vengono quindi scaricati ed eseguiti.
3. **Deployment di file dannosi sulla macchina host:** gli avversari montano l'intero file system dell'host su un container e vi accedono dal container.
4. **Ottenere informazioni sensibili dal registro Docker:** gli avversari analizzano i registri Docker per trovare informazioni sensibili.

Gli ambienti di esecuzione isolati assistiti da hardware sono stati realizzati per la protezione dei sistemi, e combinano il concetto di esecuzione isolata con tecnologie assistite da hardware. Entrambi sono fondamentali per proteggere i sistemi informatici:

- Il concetto di esecuzione isolata fornisce un TEE (Trusted Execution Environment) per l'esecuzione di strumenti difensivi su un sistema compromesso.
- L'utilizzo di tecnologie assistite da hardware esclude gli hypervisor dal TCB, raggiunge un alto livello di privilegio (ovvero privilegio a livello hardware) e riduce il sovraccarico delle prestazioni, consentendo ai cambi di contesto di essere eseguiti più velocemente nell'hardware.

Separa le applicazioni in contesti normali e sicuri, dove nel secondo è eseguito il software critico.

AMBIENTE DI ESECUZIONE SICURA:

Un ambiente sicuro consente l'archiviazione e l'esecuzione sicure delle applicazioni.

- **Esecuzione isolata:** ogni applicazione dovrebbe essere eseguita indipendentemente dalle altre applicazioni.
 - un'applicazione dannosa non può accedere ai dati sensibili conservati da altre applicazioni protette nella memoria.
 - l'applicazione dannosa non può accedere al codice o ai dati di un'applicazione mentre è in esecuzione e inoltre non può alterarne l'esecuzione.
- **Archiviazione sicura:** sono garantite l'integrità e la segretezza di tutti i dati, inclusi i file binari che rappresentano le applicazioni da eseguire. I dati più sensibili da proteggere risultano essere le password, le chiavi di crittografia e i certificati.
- **Provisioning sicuro:** questa proprietà garantisce sia la capacità di inviare in modo sicuro i dati a un software specifico nell'ambiente protetto sia la capacità di installare in remoto applicazioni sensibili e trasferire chiavi di crittografia o certificati.

Un'applicazione eseguita in un ambiente che garantisce queste tre caratteristiche è chiamata applicazione sicura.

SOLUZIONI HARDWARE:

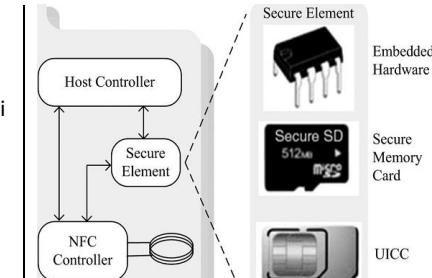
Le soluzioni di sicurezza hardware hanno il vantaggio di ridurre notevolmente le intrusioni e gli attacchi.

Gli avversari dannosi possono eseguire attacchi in grado di ispezionare la memoria di archiviazione o intercettare la chiave durante il processo di esecuzione, rendendo la crittografia non completamente in grado di affrontare i problemi di sicurezza nel mondo IoT.

La massiccia implementazione della scheda SIM che incorpora un Secure Element è l'origine delle soluzioni basate su hardware.

Un **Secure Element (SE)** è un chip a microprocessore in grado di memorizzare dati sensibili ed eseguire app sicure come i pagamenti. Agisce come un vault, proteggendo ciò che è all'interno dell'SE (applicazioni e dati) dagli attacchi malware tipici dell'host (ovvero il sistema operativo del dispositivo).

Solo le applicazioni firmate dal produttore possono essere eseguite in un SE, il che limita le possibilità di un utente malintenzionato di app e di un root attacker. Questa è una grande limitazione per gli sviluppatori e il motivo principale che ha ostacolato lo sviluppo di questa tecnologia.



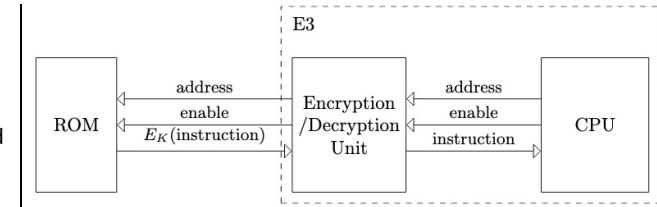
AMBIANTE DI ESECUZIONE CRIPTATO:

Un **Encrypted Execution Environment (E3)** è un ambiente di esecuzione in cui il software è crittografato e consente l'esecuzione senza rivelare le istruzioni che compongono l'applicazione.

È probabile che venga impiegata una società che assegna una grande quantità di valore monetario e tempo per proteggere i propri investimenti dalla contraffazione e da altre duplicazioni non autorizzate.

Ciò non implica necessariamente che le istruzioni possano essere eseguite direttamente dal loro stato crittografato. In effetti, è accettabile che l'E3 decifri ed esegua le istruzioni fintanto che le istruzioni in testo normale non vengono rivelate esternamente.

Se ogni dispositivo fornito con il software ha la propria chiave E3, il software E3 per un dispositivo non verrà eseguito su un altro.



TRUSTED PLATFORM MODULE (SOLUZIONE HARDWARE):

Il **TPM** è costituito da un microcontrollore con funzionalità crittografiche aggiuntive. Il TPM può essere associato al dispositivo utilizzando un bus LPC (Low Pin Count) e può eseguire operazioni crittografiche molto complesse dalla crittografia simmetrica alla crittografia asimmetrica RSA.

Il vantaggio dell'utilizzo di un TPM è che lo sviluppatore non deve sapere nulla sull'implementazione di questi algoritmi (come delle blackbox), poiché il TPM fornisce un'API (Application Programming Interface). Le funzionalità crittografiche di un TPM sono le seguenti:

- **Acceleratore RSA:** un modulo motore esegue le operazioni crittografiche RSA con una lunghezza massima della chiave di 2048 bit, ed è utilizzato durante le operazioni di firma digitale e wrapping delle chiavi;
- In grado di **calcolare valori hash** di piccole porzioni di dati, come chiavi crittografiche e certificati, ma non è sufficiente per grandi quantità di dati;
- **Generazione di numeri pseudo casuali:** questa funzione è molto importante e utile per generare chiavi di crittografia, ad esempio per RSA.

Il TPM è caratterizzato dalla chiave di endorsement, creata casualmente sul chip al momento della produzione, non può essere modificata e non lascia mai il chip, mentre la chiave pubblica viene utilizzata per l'attestazione e per la crittografia dei dati sensibili inviati al chip. Le chiavi di identità dell'attestazione vengono conservative dal TPM per eseguire un'autenticazione con un provider di servizi. Il TPM archivia tre **tipi di certificati**:

- Il **certificato di verifica dell'autenticità** garantisce l'integrità della chiave di verifica dell'autenticità. Questo certificato può essere fornito dallo stesso emittente dell'EK ma non è obbligatorio.
- Il **certificato della piattaforma** viene fornito dal fornitore della piattaforma e garantisce che tutti i componenti di sicurezza forniti con la piattaforma siano autentici. Questo certificato abilita l'attendibilità della piattaforma.
- Il **certificato di conformità** viene fornito da un laboratorio di valutazione di terze parti o dallo stesso fornitore della piattaforma. Attesta che le proprietà di sicurezza dichiarate dal produttore sono autentiche.

Il TPM **protegge i dati** mediante due possibili soluzioni:

- **Memory curtaining** estende le comuni tecniche di protezione della memoria per fornire un isolamento completo delle aree sensibili della memoria, anche al sistema operativo, ad esempio le posizioni contenenti chiavi crittografiche;
- **Sealed storage** protegge le informazioni private legandole alle informazioni di configurazione della piattaforma. I dati possono essere rilasciati utilizzando una chiave che deriva dallo stato del sistema, ovvero il software utilizzato e l'hardware su cui è in esecuzione. Le informazioni possono essere utilizzate solo con la stessa combinazione di software e hardware.

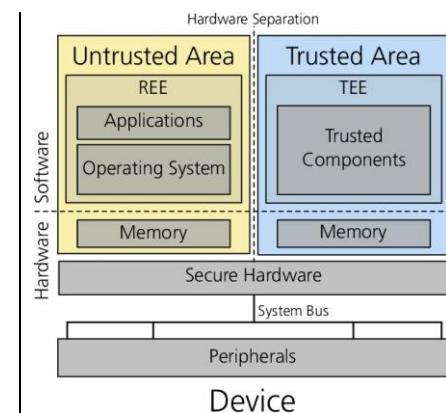
Un TPM può essere utilizzato come **PKI** che fornisce le chiavi e i certificati necessari per stabilire comunicazioni protette e firmare documenti.

TRUSTED EXECUTION ENVIRONMENT:

Avere una soluzione puramente hardware limita la programmabilità dei dispositivi. TEE combina hardware e software, e consentono di dividere il sistema in due ambienti di esecuzione:

- Il **Rich Execution Environment (REE)**, è un sistema operativo tradizionale che ha una notevole superficie di attacco.
- Il **TEE** rappresenta il sistema operativo sicuro responsabile dell'esecuzione di operazioni sensibili. Ha anche la capacità di proteggere il display e l'ingresso utilizzando una modalità sicura dei bus che collegano il processore alle periferiche I/O.

Il TEE divide il processore in due zone con un sistema operativo sicuro e altri meccanismi per migliorare la sicurezza dell'elaborazione dei dati sensibili.



L'altra caratteristica essenziale è l'archiviazione sicura, in cui i dati critici e sensibili sono isolati dai dati dell'utente per migliorarne la sicurezza. In opposizione all'SE, il TEE fornisce un canale di comunicazione sicuro tra il processore e la periferica esterna, in particolare l'ingresso e il display.

Se un'applicazione dannosa può intercettare l'input, può avere accesso a dati sensibili. Anche la visualizzazione sicura è primordiale perché l'utente deve essere sicuro che ciò che vede sullo schermo sia realmente inviato dal mondo sicuro.

È inoltre necessario un processo di avvio sicuro:

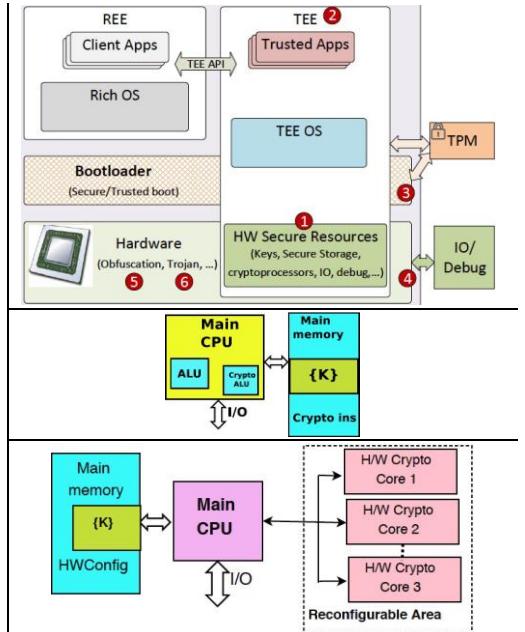
1. Leggere una ROM affidabile (bloccata in produzione);
2. Controllo della firma e dell'integrità del sistema operativo sicuro;
3. Configurazione del sistema operativo sicuro.

TEE ha anche una terza modalità chiamata Monitor, utilizzata per eseguire il salvataggio del contesto e il passaggio tra Rich e Secure OS.

Il sistema operativo sicuro ha un set di istruzioni limitato, necessario per ridurre la superficie di attacco, e pianifica le applicazioni sensibili in esecuzione su di esso per realizzare istruzioni sicure come operazioni crittografiche: generazione di chiavi, criptatura/decrittatura di dati, generazione di firme. SE e il TPM hanno più funzionalità di sicurezza fisica rispetto a TEE. Tuttavia, TEE fornisce una maggiore potenza di calcolo che consente modelli di sicurezza più complessi.

Criteria	SE	TEE	TPM
Tamper resistance	✓		✓
Secure input and display		✓	
High computation power		✓	✓
High storage capacity		✓	
Dependency to manufacturer	✓	✓	✓
Proven security level	✓	✓	✓

La base per la realizzazione a livello hardware di TEE è costituita da coprocessore, che sono un processore specializzato che esegue algoritmi crittografici a livello hardware e accelera il processo di cifratura/decifratura per una migliore sicurezza dei dati e protezione delle chiavi segrete.



Un processore generico (GPP) viene personalizzato e utilizzato per implementare alcuni algoritmi crittografici. La personalizzazione principale sono le estensioni del set di istruzioni (possono essere chiamate istruzioni crittografiche) per applicazioni crittografiche. In questo caso, le chiavi segrete vengono salvate nella memoria principale e utilizzate come altri normali dati.

Un coprocessore crittografico è un modulo esterno al GPP che accelera i calcoli crittografici. Le chiavi segrete normalmente non vengono memorizzate nella memoria del coprocessore, ma nei registri dati del processore o nella memoria principale. Il coprocessore crittografico può essere controllato utilizzando il GPP host.

Ci sono molti modi in cui questi TEE possono essere realizzati:

- La prima realizzazione utilizza un coprocessore di sicurezza per scaricare le attività critiche per la sicurezza dall'ambiente operativo principale. I vantaggi sono che l'operazione può essere generalmente completamente isolata e può essere eseguita simultaneamente con il nucleo principale. Lo svantaggio è che c'è un sovraccarico associato al trasferimento dei dati da e verso il core.

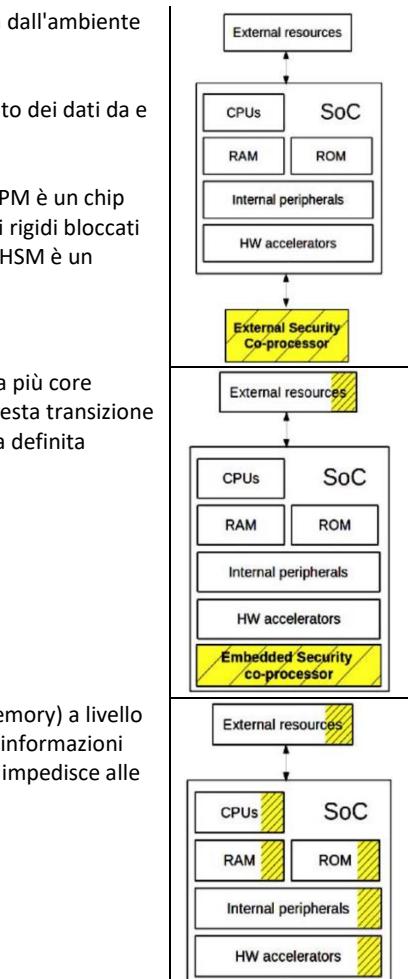
Può avere un coprocessore di sicurezza esterno (con HSM, Hardware Secure Module) o integrato (TPM). Un TPM è un chip hardware sulla scheda madre ed è in grado di fornire la crittografia completa dei dischi. TPM mantiene i dischi rigidi bloccati fino a quando il sistema non completa una verifica del sistema o un processo di autenticazione. D'altra parte, HSM è un dispositivo rimovibile o esterno che genera, archivia e gestisce chiavi crittografiche.

- Molte famose architetture TEE supportano un nuovo tipo di configurazione in cui un singolo core supporta più core virtuali che si escludono a vicenda, ovvero quando uno è in esecuzione l'altro è sospeso. Normalmente questa transizione da uno stato all'altro viene eseguita da una sorta di monitor / trigger. Questa configurazione viene talvolta definita "ambiente protetto del processore".

Esempi sono ARM TrustZone e Intel Trusted Execution Technology.

- Oltre alle tecniche basate sull'hardware, sono stati presentati altri approcci, come XOM (eXecute-Only-Memory) a livello di architettura. Tali piattaforme realizzano il TEE per separazione architettonica, previene la fuoriuscita di informazioni dalle applicazioni XOM utilizzando il compartimento, dove un compartimento è un contenitore logico che impedisce alle informazioni di entrare o uscire da esso.

Ulteriori esempi di software TEE includono i contenitori Docker.



PROCESSORI ARM TRUSTZONE:

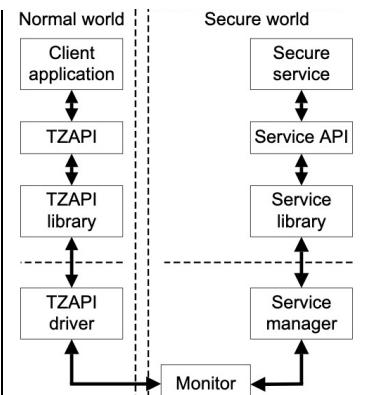
Il processore **ARM TrustZone** ha un'istruzione specifica chiamata **Secure Monitor Call (SMC)** da invocare durante l'esecuzione nel mondo normale per entrare in "modalità monitor" che esegue la transizione al mondo sicuro. Non è necessario scaricare i dati da / verso il mondo sicuro. È previsto un costo aggiuntivo per memorizzare / ripristinare lo stato del dispositivo all'ingresso / uscita da una determinata modalità. Lo svantaggio è che quando un mondo è attivo, l'altro deve essere completamente fermato, complicando così la gestione delle interruzioni. L'innovazione architettonica più significativa è l'aggiunta di un nuovo bit del processore, il 33-esimo, che indica in quale modo il processore è attualmente in esecuzione.

Il **TrustZone address space controller (TZASC)** estende la sicurezza a livello di memoria, abilitando la partizione in differenti regioni.

L'**unità di gestione della memoria (MMU)** che riconosce la zona di fiducia fornisce due interfacce MMU distinte.

L'**API TrustZone (TZAPI)** specifica in che modo le applicazioni non protette (NSA), in esecuzione nell'ambiente ricco, interagiscono con l'ambiente di esecuzione isolato.

Seguendo un modello client-server, l'API definisce un insieme di interfacce software astratte mediante le quali un NSA può interagire con un TA. L'API consente ai client di inviare comandi e richieste a un TA e scambiare dati tra i due mondi. L'API TrustZone non include alcuna specifica su come sviluppare applicazioni in esecuzione all'interno dell'ambiente di esecuzione isolato.



INTEL SGX:

Tecniche come **Intel Software Guard Extensions (Intel SGX)** utilizzano sia hardware che software per implementare TEE.

Le applicazioni Intel SGX sono costituite da una parte affidabile e una parte non attendibile. Quando l'applicazione deve lavorare con un segreto, crea un'enclave che viene collocata nella memoria attendibile. Quindi chiama una funzione attendibile per entrare nell'enclave, dove gli viene fornita una visualizzazione dei suoi segreti in testo chiaro.

Tutti gli altri tentativi di accesso alla memoria dell'enclave dall'esterno dell'enclave sono negati dal processore, anche quelli effettuati da utenti privilegiati. Ciò impedisce che i segreti nell'enclave vengano svelati.

SGX è un insieme di istruzioni della CPU che consentono a un'applicazione di istanziare un contenitore protetto, denominato enclave. Un'enclave è definita come un'area protetta nello spazio degli indirizzi dell'applicazione che non può essere alterata da codice esterno all'enclave, nemmeno da codice con privilegi superiori. Il core non esegue una transizione completa, ma parti di un'applicazione standard sono protette da meccanismi hardware nel core. I vantaggi sono che non è necessario trasferire i dati avanti e indietro tra il core o impostare transizioni complicate da e verso un mondo sicuro e non è necessario un ambiente operativo separato come richiesto in altri stili di configurazione TEE.

Il componente affidabile dovrebbe essere il più piccolo possibile: una grande enclave con un'interfaccia complessa non consuma solo più memoria protetta, ma crea anche una superficie di attacco più ampia. Le enclave dovrebbero anche avere un'interazione minima tra componenti attendibili e non attendibili, e quindi limitare queste dipendenze rafforzerà l'enclave contro gli attacchi.

Untrusted Run-Time System (uRTS) indica il codice che viene eseguito al di fuori dell'ambiente enclave ed esegue il caricamento e la manipolazione di un'enclave, l'esecuzione di chiamate (ECALL) a un'enclave e la ricezione di chiamate (OCALL) da un'enclave.

Trusted Run-Time System (tRTS) rappresenta il codice che viene eseguito all'interno dell'ambiente dell'enclave ed esegue la ricezione di chiamate (ECALL) dall'applicazione e l'esecuzione di chiamate all'esterno (OCALL) dell'enclave o la gestione dell'enclave stessa.

OPEN-TEE:

Agli sviluppatori di applicazioni mancano le interfacce per utilizzare le funzionalità TEE basate su hardware. In effetti, il loro utilizzo è stato limitato principalmente alle applicazioni sviluppate dai fornitori di dispositivi. La recente standardizzazione delle interfacce TEE da parte di **Global Platform (GP)** promette di risolvere parzialmente questo problema consentendo alle applicazioni conformi a GP di essere eseguite su TEE di diversi fornitori.

Gli sviluppatori ordinari che desiderano sviluppare applicazioni affidabili devono affrontare sfide significative. È difficile ottenere l'accesso alle interfacce hardware TEE senza alcun supporto da parte dei fornitori. Gli strumenti e il software necessari per sviluppare ed eseguire il debug di applicazioni affidabili possono essere costosi o inesistenti, con solo tecniche di debug primitive come il "tracciamento di stampa".

Un TEE virtuale chiamato **Open-TEE**, conforme alle specifiche GP, è stato proposto come TEE virtuale conforme agli standard implementato interamente nel software consentirà agli sviluppatori di creare applicazioni TEE utilizzando strumenti e ambienti di sviluppo con cui hanno familiarità.

La specifica GP è composta da:

- **L'API TEE Core** fornisce un ampio set di funzionalità, come un'API crittografica e uno storage sicuro, per implementare un TA;
- **L'API client TEE** è uno strato molto generico e sottile costituito da un numero limitato di funzioni e definizioni per trasferire i dati avanti e indietro da REE a TA.

Tra l'API client TEE in esecuzione su REE e l'API TEE Core in esecuzione su TEE, abbiamo un'efficace RPC (Remote Procedure Call) in cui un processo in esecuzione in REE può richiamare attività nel TEE.

Questi sforzi di standardizzazione nella piattaforma globale potrebbero risolvere il problema dei TEE interoperabili. Tuttavia, non rimuovono l'ostacolo all'accesso all'hardware richiesto né semplificano il compito di sviluppare e testare le TA.

Open-TEE fornisce un'architettura e un **kit di sviluppo software (SDK)** che implementa le specifiche GP come framework e un insieme di strumenti familiari allo sviluppatore, eliminando così la necessità di hardware specializzato e le spese generali che comporta.

▪ **Base**: Open-TEE è progettato per funzionare come un processo demone nello spazio utente. Inizia l'esecuzione di Base, un processo che incapsula la funzionalità TEE nel suo complesso. Una volta inizializzato, Base eseguirà il fork per creare due processi indipendenti ma correlati.

▪ **Manager**: Manager può essere visualizzato come il "sistema operativo" di Open-TEE:

1. Gestione delle connessioni tra le applicazioni;
2. Monitoraggio dello stato del TA;
3. Fornire archiviazione sicura per TA;
4. Controllo delle regioni di memoria condivisa per le applicazioni.

La centralizzazione in un processo di controllo può anche essere visto come un wrapper che astrae l'ambiente in esecuzione e lo riconcilia con i requisiti imposti dagli standard GP TEE.

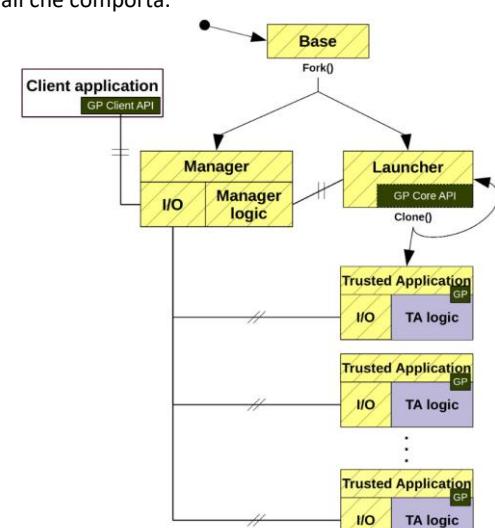
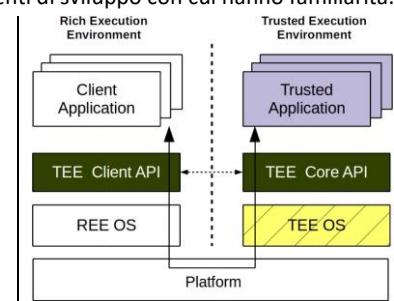
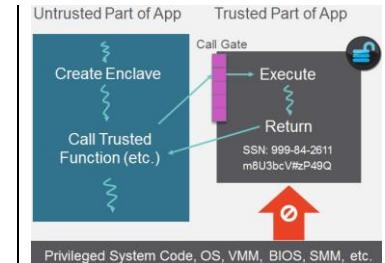
▪ **Launcher**: L'unico scopo di Launcher è creare nuovi processi TA in modo efficiente.

Quando viene creato per la prima volta, Launcher caricherà una libreria condivisa che implementa l'API TEE Core e attenderà ulteriori comandi da Manager. Manager segnalerà Launcher quando è necessario avviare un nuovo TA. Dopo aver ricevuto il segnale, Launcher si clonerà da solo. Il clone caricherà quindi la libreria condivisa corrispondente al TA richiesto.

Il design di Launcher segue il modello di progettazione "zigote", che è un processo speciale del sistema operativo Android che abilita il codice condiviso su Dalvik/ Art VM in contrasto con Java VM in cui ogni istanza ha la propria copia dei file di classe della libreria principale e degli oggetti heap. Un processo TA appena creato viene quindi reimpostato su Manager in modo che sia possibile controllarlo.

▪ **Trusted Application**: I processi TA sono stati divisi in due fili. Il primo gestisce Inter-Process Communication (IPC) e il secondo è il thread di lavoro, indicati rispettivamente come thread IO e TA Logic. Questo modello architettonico consente di interrompere il processo senza arrestarlo e consente una maggiore separazione e astrazione della funzionalità TA dal framework Open-TEE.

▪ **Collegamenti**: L'API client TEE e l'API TEE Core sono implementate come librerie condivise per ridurre il consumo di codice e memoria. Open-TEE implementa un protocollo di comunicazione oltre ai socket del dominio Unix e ai segnali di interelaborazione come mezzo per controllare il sistema e trasferire i messaggi tra CA e TA.



6.1 APPLICAZIONE DI TEE NELLE BLOCKCHAIN

TEE è utilizzata anche in ambito blockchain, non nelle soluzioni in commercio ma in quelle di ricerca con lo scopo di semplificare e migliorare alcune delle funzionalità delle blockchain.

HYPERLEDGER SAWTOOTH:

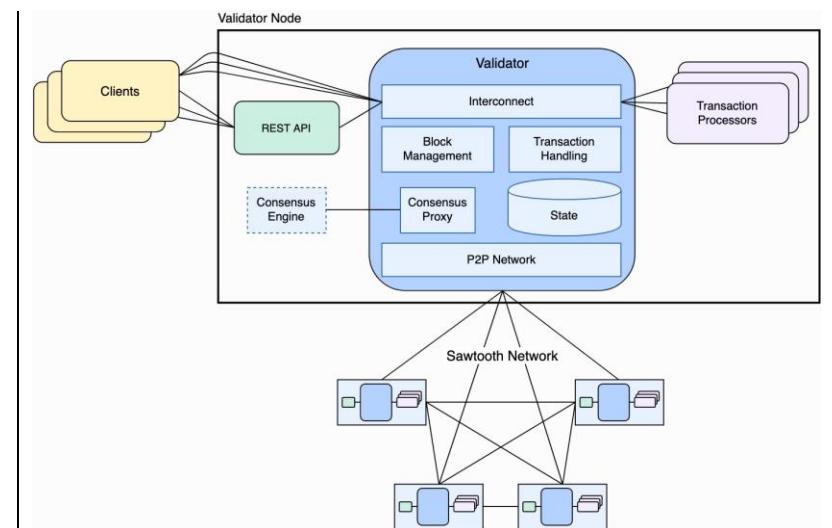
Hyperledger Sawtooth è una piattaforma blockchain aziendale altamente modulare che consente alle applicazioni di scegliere le regole di transazione, i permessi e gli algoritmi di consenso che supportano le loro esigenze aziendali uniche.

Sawtooth semplifica lo sviluppo e la distribuzione di un'applicazione fornendo una chiara separazione tra il livello dell'applicazione e il livello del sistema principale.

Sawtooth fornisce un'astrazione degli smart contracts che consente agli sviluppatori di applicazioni di scrivere la logica del contratto in una lingua a loro scelta, inclusi Python, Javascript, Go, C++, Java e Rust.

Sawtooth rappresenta lo stato per tutte le famiglie di transazioni in una singola istanza di un albero Merkle-Radix su ogni validatore, ovvero un Merkle tree indirizzabile dove gli indirizzi identificano in modo univoco i percorsi ai nodi foglia dell'albero in cui sono archiviate le informazioni.

Il processo di convalida dei blocchi su ciascun validatore garantisce che le stesse transazioni risultino nelle stesse transizioni di stato e che i dati risultanti siano gli stessi per tutti i partecipanti alla rete.



Sawtooth astrae i concetti fondamentali del consenso e isola il consenso dalla semantica delle transazioni. L'interfaccia di consenso Sawtooth supporta il collegamento di varie implementazioni di consenso come *consensus engines* che interagiscono con il validatore attraverso l'API di consenso e di comunicazione P2P. Sawtooth consente di modificare il consenso dopo che la rete è stata creata ed è in esecuzione con una o due transazioni.

Il **consenso Proof of Elapsed Time (PoET)** offre una soluzione al problema dei generali bizantini che utilizza un "ambiente di esecuzione affidabile" per migliorare l'efficienza delle soluzioni attuali come Proof-of-Work.

PoET sceglie stocasticamente i peer per eseguire le richieste, e comportamenti fraudolenti sono evitati da TEE e verifica di identità.

PoET funziona essenzialmente come segue:

- Ogni validatore richiede un tempo di attesa da un'enclave (mediante l'esecuzione di una funzione attendibile);
- Il validatore con il tempo di attesa più breve per un particolare blocco di transazione viene eletto leader;
- Una funzione, come "CreateTimer", crea un timer per un blocco di transazioni che è garantito essere stato creato dall'enclave;
- Un'altra funzione, come "CheckTimer", verifica che il timer sia stato creato dall'enclave. Se il timer è scaduto, questa funzione crea un'attestazione che può essere utilizzata per verificare che il validatore abbia atteso il tempo assegnato prima di rivendicare il ruolo di leadership.

Questo algoritmo rientra nella casistica basata sulla lotteria come quella di Nakamoto.

In generale, le soluzioni BFT possono resistere a comportamenti maliziosi di alcuni nodi. Le soluzioni CFT presumono che nessun nodo sia malizioso, ma può bloccarsi o scomparire dalla rete senza anomalie nel suo comportamento. Le soluzioni CFT sono generalmente meno costose e più scalabili.

- PoET con hardware SGX realizza un consenso BFT. L'enclave SGX genera in modo sicuro il valore del tempo di attesa casuale a prova di manomissione. L'enclave firma quindi un certificato con il valore del tempo di attesa. Dopo la scadenza del timer, l'attestazione SGX viene inviata agli altri nodi di rete. I nodi peer verificano la firma del tempo di attesa generata dal nodo vincente. Il nodo vincente può pubblicare il blocco.
- PoET è disponibile anche senza SGX, che è simulato. Data tale simulazione, il consenso sarà solo CFT non BFT.

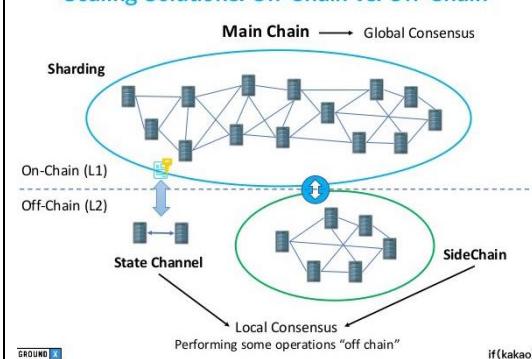
HYPERLEDGER AVALON:

Le blockchain offrono consistenza e trust tramite una replica massiccia dei dati, ma hanno un throughput limitato e privacy e riservatezza imperfette.

- **On-chain** sono le transazioni che consistono in una transazione di stato con consenso globale.
- **SideChain** è una soluzione di pochi nodi che mantiene un consenso locale per determinate informazioni estratte dalla catena principale.
- **Off-chain** sono elaborazioni su dati estratti dalla catena.

L'aggiunta di un'esecuzione off-chain affidabile a una blockchain è proposta come la soluzione per migliorare le prestazioni.

Scaling Solutions: On-Chain vs. Off-Chain



Una blockchain principale mantiene una singola istanza autorevole degli oggetti, applica le politiche di esecuzione e garantisce la verifica delle transazioni e dei risultati, mentre l'elaborazione offchain associata consente una maggiore produttività. Si rende necessario abbinare soluzioni di trusted computing per aumentare l'integrità delle processazioni e proteggere la riservatezza dei dati.

Ogni azienda ha richiedenti che inviano transazioni e i worker li eseguono. Le ricevute sono registrate sulla blockchain.

Le richieste di elaborazioni verso workers ospitati in un enclave TEE sono inviate dai richiedenti tramite l'interfaccia utente front-end o strumenti a riga di comando. Gli ordini di lavoro possono essere inviati anche da smart contract in esecuzione su DLT. Esistono due modelli di funzionamento:

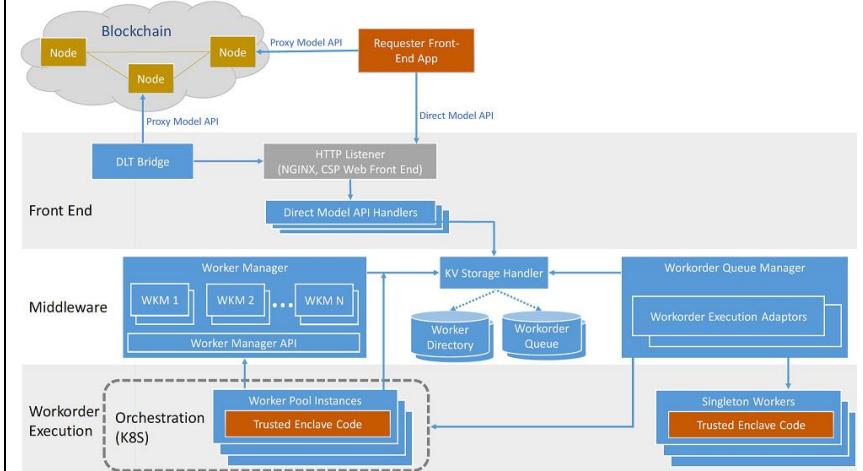
- **Modello proxy** per smart contract con componenti che implementano le interazioni tra DLT e TCS o connettori che astraggono le API specifiche di un DLT o un proxy Avalon invocabile dai richiedenti mediante la sintassi della DLT.
- **Modello diretto**, che fornisce un'API RPC JSON. Può essere utilizzato anche un modello ibrido che combina elementi di entrambi i modelli proxy e diretti.

KV Storage Handler:

Tutte le comunicazioni tra i componenti front-end e middleware vengono effettuate tramite KV Storage Manager, che mantiene:

- Una directory dei worker con tutte le informazioni sull'attestazione, sul tipo (singleton o pool) e i parametri di esecuzione (es. pod K8S, numero massimo consentito di istanze simultanee)
- Coda ordini di lavoro, che contiene richieste di ordini di lavoro, risposte e, facoltativamente, ricevute.

Il KV Storage Manager è un thin wrapper di Lightning Memory-Mapped Database (LMDB) che rappresenta una libreria software per fornire un database transazionale ad alte prestazioni sotto forma di archivio di valori-chiave.



Worker Manager:

Il Worker Manager è responsabile della creazione della chiave crittografica del worker mediante Worker Key Manager (WKM) per i pool di worker, della creazione e manutenzione dei pool di worker.

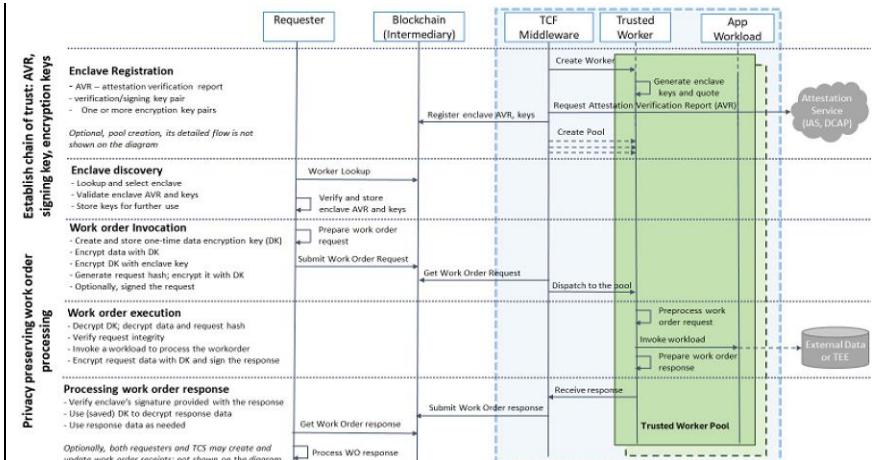
Workorder Queue Manager:

Il gestore della coda limita il flusso di richieste per un dato worker, serializza l'esecuzione delle richieste esprimendo le dipendenze esplicitamente specificate, secondo il modello più letture e una scrittura. Avvia l'esecuzione della richiesta tramite uno o più adattatori di esecuzione e gestisce la dimensione della coda con la rimozione di richieste troppo vecchie.

Workorder Execution:

L'interazione con i worker fondamentalmente avviene con un'interazione diretta con istanze di worker allocati singolarmente o come pool, in maniera statica o dinamica. In aggiunta è possibile prevedere che i worker siano impacchettati come pod K8S e vengono avviati tramite l'API di Kubernetes, così da avere un motore di coordinamento e loadbalancing tra di essi.

- **Enclave Registration:** Inizialmente un nuovo worker viene istanziato in un enclave con apposito manager. Le sue informazioni di attestazione vengono generate e pubblicate sulla blockchain (o/e su un server);
- **Enclave discovery:** Successivamente un richiedente cerca un worker, ne verifica le informazioni di attestazione e le memorizza per un ulteriore utilizzo;
- **Work Order Invocation:** Il richiedente crea una richiesta di lavoro e lo memorizza sulla blockchain. Il manager preleva l'ordine e lo inoltra a un worker nell'enclave.
- **Work Order Execution / processing work order response:** A conclusione, la risposta è posta nella blockchain e viene reperita dal richiedente.



L'implementazione Avalon esistente fornisce l'attestazione e la registrazione dei lavoratori, l'applicazione dell'integrità e della riservatezza degli richieste end-to-end, la loro elaborazione asincrona, il modello diretto e l'integrazione con le blockchain Fabric ed Ethereum.

Open Enclave con supporto SGX è impiegato per l'implementazione dell'enclave dove eseguire i worker.

