Sicurezza by Della Rocca e D'Angelo

A.A. 2022/2023

Indice

Capitolo 1
Sistemi distribuiti e consenso

Un sistema distribuito è sostanzialmente un software in esecuzione su più componenti eterogenei, concorrenti e appartenenti a diversi domini organizzativi/amministrativi. Ogni componente è autonomo, ovvero possono avere diverso hardware e software. Ciò implica che hanno diversi modelli di fallimenti. Comunicano mediante lo scambio di messaggi sulla rete a cui sono connessi. In questi sistemi non esiste il concetto di clock globale per via del **drift dei clock** (gli orologi tra i diversi sistemi non sono sincronizzati, un motivo è il diverso fuso orario nel mondo). Infine i guasti in questi sistemi sono indipendenti cioè un guasto in un nodo non implica un guasto in tutta la rete (guasti hardware, software, rete).
Un sistema distribuito deve possedere diverse proprietà

- **Eterogeneità**: Varietà e differenze nelle macchine (hardware, software, sistemi operativi);
- **Aperti**: I sistemi distribuiti sono aperti ed interagiscono tra di loro. Un sistema distribuito aperto offre servizi secondo regole standard per descrivere la sintassi e la semantica del servizio. Le regole sono di solito specificate attraverso delle interfacce, questo garantisce interoperabilità e portabilità;
- **Sicurezza**: Un sistema distribuito è più semplice da attaccare data la sua apertura. Un attaccante può pianificare facilmente un attacco poiché può accedere alle specifiche. Le caratteristiche da rispettare sono quelle della triade CIA;
- **Concorrenza**: L'utilizzo delle risorse e servizi deve avvenire in maniera concorrente;
- **Trasparenza**: Un sistema distribuito deve nascondere all'utente tutta sua eterogeneità. Le trasparenze che deve garantire sono: Accesso, locazione, migrazione, duplicazione, concorrenza, fallimenti, persistenza.
- **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 tre dimensioni: numero di utenti e risorse, geograficamente (distanza tra utenti e risorse) e dal punto di vista amministrativo.
- **Guasti** I sistema distribuito deve essere tollerante ai guasti, ovvero mentre viene risolto un guasto le risorse ed i servizi vengono sempre forniti.
- **Modelli** I sistemi distribuiti adottano un modello che descrive le caratteristiche essenziali. Specifica quali sono le principali entità del sistema, e come interagiscono tra di loro.
  Un modello fondamentale è un'astrazione delle caratteristiche essenziali e delle specifiche di un sistema. Tale modello viene utilizzato per progettare ed implementare il sistema.

Un sistema distribuito è composto da processi ciascuno dotato di proprio clock ed in esecuzione sui diversi nodi del sistema. Possiedono uno stato, ma esso è locale, cioè può essere visto solo dal processo che lo possiede. Questi processi per raggiungere un obiettivo comune utilizzano algoritmi distribuiti.

I sistemi distribuiti possono essere di due tipi:

- **Sincrono**: Un sistema distribuito sincrono è un sistema distribuito in cui i nodi lavorano in modo coordinato e sincronizzato. In un sistema distribuito sincrono, tutti i nodi devono attendere l'arrivo di un segnale di sincronizzazione prima di continuare a eseguire le proprie operazioni. Questo assicura che tutti i nodi abbiano la stessa visione del sistema e che le operazioni vengano eseguite in modo coerente. Si possono individuare i fallimenti di un nodo utilizzando un time-out sui tempi di risposta di quest'ultimo. Il problema di questi sistemi è che più sono grandi più è difficili controllarli.
- **Asincrono**: Un sistema si dice asincrono se non esistono limiti alla velocità di esecuzione dei processi, al ritardo di trasmissione dei messaggi e alla deviazione degli orologi. Di tali limiti non è possibile nemmeno effettuarne una stima. Per questi motivi alcuni problemi non hanno soluzione nei sistemi distribuiti.

Un sistema distribuito può subire fallire secondo la catena fault-error-failure:

- **Failure (GUASTO)**: comportamento indesiderato o non corretto del sistema;
- **Fault (Difetto)**: causa originaria del guasto;
- **Error (Errore)**: è lo stato in cui si trova il sistema dopo un fault.

Approfondimento: Un guasto (fault) determina un errore (error), ossia uno stato erroneo del sistema. Se un errore si propaga al di fuori della interfaccia del sistema e diviene osservabile, prende il nome di fallimento (failure).

Questi fallimenti sono poi catalogati in:

- **omission failure**: un processo esegue un azione indesiderata;
- **fallimento bizantino**: comportamento arbitrario di un processo o canale;
- **timing failure**: non viene rispettata una scadenza, capita nei sistemi sincroni.

1.1 Consenso

1.1.1 Problema del consenso

Consiste nel far si che tutti i processi in esecuzione su più nodi del sistema convergano su un valore comune. Un possibile esempio è il seguente: abbiamo N processi in esecuzione su M nodi, ognuno possiede una variabile. Se non c'è consenso alle variabili non sarà assegnato nessun valore. Se c'è un consenso avranno tutti lo stesso valore.

- Abbiamo N processi che comunicano con lo scambio di messaggi;
- i canali si assumano affidabili;
- i processi possono fallire;
- il processo p_i ha una variabile di decisione e propone un suo valore;
- Ogni processo parte da uno stato di indecisione e mira ad ottenere uno stato di decisione per assegnare un valore alla sua variabile cosìché diventi immutabile.

Un algoritmo di consenso distribuito deve garantire:

- **terminazione**: prima o poi tutti i processi corretti decidono che valore assegnare alla propria variabile;
- **accordo**: due processi corretti non decidono valori diversi;
- **integrità**: se tutti i processi corretti propongono lo stesso valore, allora quel valore sarà al decisione finale;

1.1.2 Problema dei generali bizantini

Il comportamento bizantino è una condizione di un sistema distribuito in cui i componenti possono guastarsi e compiere azioni discordanti che compromettono il raggiungimento del consenso. In un errore bizantino un componente può sembrare guasto ma anche funzionante ad un sistema di rilevamento degli errori. Quindi non è possibile escludere con certezza una componente guasta. Il termine prende il nome dal "problema dei generali bizantini". In questo problema abbiamo 3 o più generali che comunicano tra di loro e devono decidere se attaccare o ritirarsi. Ogni generale comunica con gli altri tramite dei messaggeri se uno di questi messaggeri è bizantino potrebbe recapitare un messaggio diverso facendo così che non si raggiunga un accordo su che azione prendere. Se invece il generale ha un comportamento bizantino darà informazioni discordanti agli altri generali.
In un sistema distribuito sincrono il problema ha soluzione se e solo se il numero di nodi N è maggiore di tre volte il numero dei fallimenti. In questo problema la terminazione e l'accordo sono lo stesso del problema del consenso, l'integrità invece richiede che il generale non sia bizantino.

1.1.3 Problema della consistenza interattiva

Si tratta di una variante del problema del consenso. In questo caso ogni processo propone un solo valore e l'obiettivo è che tutti i processi concordino su un vettore di valori. I requisiti di terminazione e accordo sono i classici. Invece, il requisito di integrità richiede che se p_i è corretto allora tutti i processi corretti hanno il valore i-esimo del vettore uguale. Se hanno valori discordanti allora p_i non è corretto.

1.1.4 Consenso in sistema sincrono

In questo tipo di sistema il consenso si può raggiungere utilizzando la primitiva del multicast.

- La primitiva multicast(g, m) è invocata da un processo per inviare un messaggio m ad un gruppo g.
- La primitiva di deliver(m) viene invocata da un processo per ricevere il messaggio m.

Esistono altre primitive B-multicast e il B-deliver, che garantiscono che un processo corretto prima o poi riceverà un messaggio se invoca la B-deliver, questo a patto che il mittente non fallisca. Queste sono sfruttate nell'algoritmo di Dolev:

- Si inizializza un vettore V contenente tutti i valori proposti dai vari processi;
- Si effettua un ciclo fino a f+1 iterazioni supposte in cui si raggiunge il consenso;
- Ad ogni iterazione un processo p_i effettua il B-multicast, inviando i valori che no ha inviato nei cicli precedenti;
- Nello stesso momento i processi p_i effettuano un B-deliver per ricevere i messaggi dagli altri processi;
- La durata del ciclo viene limitata da un timeout visto che si conoscono i tempi di riposta di un processo. Se non si ottiene risposta entro un limite di tempo allora quel processo è crashato;
- Dopo f+1 cicli ogni processo sceglie il valore minimo del vettore V, e questo sarà uguale per tutti.

La **terminazione** è garantita perché il numero dei cicli è finito. L'**accordo** e **integrità** sono garantite perché tutti i processi che non crashano arrivano ad avere i medesimi valori nel vettore, ed estraendo tutti il minimo da quest'ultimo avranno tutti la stessa variabile di decisione.

1.1.5 Consenso in un sistema asincrono

Non esistono algoritmi deterministici per garantire il consenso in sistemi asincroni. per raggiungere quest'ultimo possiamo:

- **Indebolire la condizione di terminazione**;
- **Indebolire la condizione di accordo** individuando un insieme finito di possibili valori di decisione;
- **Irrobustire il modello del sistema** rendendolo più sincrono possibile.

1.2 Algoritmo del consenso di Paxos

L'algoritmo di consenso di Paxos si basa sulla storia del parlamento di Paxos.
Il Parlamento ha il compito di determinare le leggi (consenso). Ciascun membro del parlamento possedeva un registro sul quale annotare i vari decreti approvati, numerati in ordine crescente. I parlamentari però non comunicavano in modo diretto tra di loro, ma tramite scambio di messaggi spediti da messaggeri. Sia i parlamentari che i messaggeri potevano entrare o uscire dal parlamento. I messaggeri potevano anche uscire prima di consegnare un messaggio affidatogli, "magari per un viaggio di sei mesi" o "andar via per sempre e il messaggio non veniva consegnato. Quando lavoravano entrambi, però, poteva succedere di approvare leggi in contrapposizione o leggi aventi lo stesso numero. Ogni legislatore di Paxos manteneva un libro mastro, dove registrava tutto ciò che è succedeva.

Requisiti

- Safety: Può essere scelto solo un singolo valore che è stato proposto. Un nodo non apprende mai che un valore è stato scelto a meno che non lo sia stato effettivamente. (accordo e integrità rispettati)
- Liveness: Alla fine viene scelto un valore proposto. Se è stato scelto un valore, un nodo può eventualmente apprendere il valore (terminazione rispettata).

Proprietà di Paxos

- **P1:** il numero di proposta è unico.
- **P2:** il valore inviato nella fase di accept è il valore della proposta con il numero più alto di tutte le prepare.

Ruoli in Paxos

- **Proposer** ha la facoltà di proporre un valore a tutti gli acceptor. Deve essere unico (evitare concorrenza tra proposer), se crasha si utilizza un timer se entro un tot periodo il proposer non risponde ne viene rieletto un altro.
- **Acceptor** ha la facoltà di accettare un valore proposto, di solito sono più processi (potrebbero fallire);
- **Learner** apprendono e gestiscono la scelta fatta dagli acceptor, di solito sono più processi (potrebbero fallire).

Algoritmo di Paxos

- **Prima fase (Prepare):** Un proposer seleziona un numero di proposta e lo invia agli acceptor tramite un messaggio **prepare(n)**, n è un identificativo. Gli acceptor rispondono al proposer promettendo che non accetteranno altre prepare con identificativi inferiori a n (**promise**).
- **Seconda fase (Accept):** Nel momento in cui il proposer rileva un *numero sufficiente* di promesse, invia alla maggioranza di acceptor il messaggio accept(n,v) dove v è il valore della proposta con n più grande. Se un acceptor riceve un messaggio accept(n,v) accetta la proposta, a meno che non abbia risposto ad una richiesta di prepare avente un numero seriale maggiore di n, informando però i learner. A loro volta i learner, quando ricevono un *numero sufficiente* di messaggi accept, inviano un commit al proposer.

Il «numero sufficiente» precedentemente indicato è pari al 50% degli acceptor.
Paxos funziona solo per lo scenario catastrofico in cui si verifica un fail-stop (il nodo può bloccarsi o non restituire valori) e non tiene conto dei difetti legati alla questione dei nodi Bizantini.

Capitolo 2
Comunicazione di gruppo e consistenza

Nelle comunicazione che siano esse unicast (invio di un messaggio tramite un indirizzo fisico o logico), broadcast o multicast (invio di un messaggio a membri di un gruppo) l'obiettivo che si vuole raggiungere è quello di assicurarsi che tutti i processi possano scambiarsi dei messaggi anche in presenza di:

- **fallimenti dei canali di comunicazione**: un messaggio viene perso nella rete (es. congestione)
- **fallimento dei processi**: I processi comunicano utilizzando un overlay (definito a livello applicativo). Quando un processo invia un messaggio ad un altro processo il messaggio potrebbe passare attraverso altri processi, se uno di questi fallisce allora il messaggio si perde.

Per garantire la consegna di un messaggio si utilizzano diverse strategie:

- **Error masking**: un messaggio viene inviato su più percorsi diversi che giungano alla stessa destinazione (ridondanza spaziale), oppure i messaggi vengono ritrasmessi a intervallo di tempo (ridondanza temporale). La strategia è pro-attiva.
- **Error detection and recovery**: Si controlla se un messaggio non è stato inviato e lo si ritrasmette. Abbiamo due possibili schemi. **Positive ack** dove un messaggio è ritrasmesso se non viene ricevuta una conferma dove t tempo. **Negative ack**: il destinatario invia un ack per avvisare il mittente che non ha ricevuto niente. In questo schema i messaggi devono essere numerati per poter richiedere una ritrasmissione più efficiente.

2.1 B-multicast e R-multicast

Il **Basic multicast** garantische che un processo corretto prima o poi riceverà un messaggio se quest'ultimo invoca una basic-deliver. Il problema di questa primitiva è che soffre dell'**ack-implosion** cioè vengono inviati ad un host un numero elevati di pacchetti.
Per ovviare a ciò si è arrivati al **reliable-multicast**, che garantisce che un messaggio sia consegnato in modo affidabile ad un processo, infatti se un destinatario non riceve un pacchetto può richiederne la ritrasmissione finché non viene correttamente consegnato. Può essere applicato anche includendo processi no corretti in questo caso si chiama **uniform reliable multicast**.

2.1.1 Implementazio R-multicast

La sua implementazione si basa sull B-multicast. Ogni processo contiene i un insieme di messaggi ricevuti. Ogni processo per inviare un messaggio ad un gruppo ultilizza i b-multicast. Alla ricezione ogni processo controlla se non ha già ricevuto il messaggio, se non l'ha ricevuto lo aggiunge al suo insieme, e se non è il mittente del messaggio lo ritrasmette.

2.2 IP-multicast e Applciation-Level Multicast

La soluzione migliore per realizzare un sistema di comunicazione di gruppo è di utilizzare un multicast basato sul protocollo IP. Si identifica un gruppo con un indirizzo multicast (classe D). Si incarica i router di instradare e replicare i pacchetti ai membri. Non garantisce che il messaggio sia consegnato. Molto utile in rete LAN ma non WAN, questo eprchè gli ISP non consento traffico IP multicast per non appesantire il carico sui router, richiede aggiornamenti all'intera infrastruttura della rete ecc.
Per reti più estese si può utilizzare l'Application-level Multicast (ALM). Quest'ultimo costruisce un overlay (rete virtuale che permette di far comunicare più velocemente) tra i processi che si basa sul livello di trasporto. Le connessione tra i vari processi possono essere realizzate come un albero dove un nodo al livello i invia messaggi a nodi al livello i + 1. IP multicast è più efficiente perché l'instradamento è garantito dai router, mentre per l'ALM è possibile che si passi più volte dallo stesso router.

2.3 Ordinamento Messaggi ricevuti

2.3.1 Ordinamento FIFO

Nell'ordinamento FIFO se un processo corretto invia in multicast un messaggio m e m' allora ogni processo riceverà i messaggi nell'ordine m, m'.
Quest'ordinamento si realizza utilizzando dei numeri di sequenza. Ogni gruppo ha un suo contatore che indica i messaggi inviati da un processo p a un processo q. Si utilizza poi un variabile R contenente l'ultimo numero di sequenza consegnato da p. Infine si utilizza una coda con tutti i messaggi fuori sequenza. Quindi ad ogni ricezione di un messaggio si controlla se questo è successivo all'ultimo ricevuto se non lo è lo si aggiunge alla coda.

2.3.2 Ordinameno Causale

Se l'invio di un messaggio m è in relazione happened-before con un messaggio m' allora ogni processo riceverà prima m e poi m'.
Per implementarlo no si utilizza un numero di sequenza, ma un clock-vector che che viene incrementato considerando le attività di ogni processo. L'i-esimo elemento di questo vettore conta i messaggi inviati dal processo i-esimo in relazione happened before con il prossimo messaggio. Ogni processo ha i proprio clock-vector. Ad ogni invio di messaggio un processo p aggiorna la sua componente nel vettore e lo invia insieme al messaggio. Quando un processo riceve un messaggio lo inserisce nella sua coda aspettando che si verifichi un happened-before. Quando si verifica si toglie il messaggio dalla coda e si incrementa la posizione del vettore corrispondente al mittente.

2.3.3 Ordinamento totale

Se un processo riceve i messaggi nella sequenza m e m' allora ogni altro processo riceverà prima m e poi m'. Viene imposto lo stesso ordine di ricezione su tutti i processi. Questo è possibile implementando numeri di sequenza non per un singolo processo ma per tutto il gruppo. Per far si che tutti i processi abbiano lo stesso numero di sequenza si utilizza un sequencer. All'invio di un messaggio si invia anche il sequencer, quest'ultimo serve per comunicare ai membri del gruppo qual'è il numero di sequenza del messaggio su cui bisogna invocare la b-deliver. Anche qui c'è una coda per conservare i messaggio fuori sequenza.

2.4 Gossiping

R-multicast ha il problema di congestionare la rete con l'invio di tutti i messaggi ogni volta. Per ovviare a questo problema si può utilizzare il Gossiping. È un algoritmo distribuito per la consegna affidabile dei messaggi. Le principali varianti di quest'algoritmo sono:

- **modalità push**: ogni nodo periodicamente invia un messaggio ad un sottoinsieme k dei nodi del gruppo.
- **modalità pull**: i nodi si interrogano inviandosi dei messaggi di gossip per capire se hanno per qualche messaggio, anche qui li si invia a un sottoinsieme.

Il contenuto dei messaggi si può dividere in due categorie:

- **positive**: il messaggio di gossiping contiene gli id dei messaggi correttamente ricevuti;
- **negative**: il messaggio di gossiping contiene gli id dei messaggi persi;

Le combinazioni più utilizzate sono **pull/negative** e **push/positive**. Quest'algoritmo può anche essere utilizzato per causare attacchi DoS. Per evitare questo possibile attacco si potrebbe certificare l'identità di chi invia. Per garantire l'integrità dei dati basterebbe cifrare i messaggi.

2.5 Failure Detector

Il compito dei failure detector è quello di rilevare i fallimenti dei vari nodi. Sono utilizzati generalmente per individuare fallimenti di tipo crash. Abbiamo diversi tipi di failure detector:

- **Failure detector semplice**:Invia un messaggio ad un processo ed attende una risposta. Se non la riceve dopo tot tempo considera il processo fallito;
- **Failure detector inaffidabile**: utilizzati nei sistemi asincroni, il loro risultato può non essere corretto:

  - **unsuspected**: il processo non è fallito perché abbiamo ricevuto risposta, ma non si esclude il fallimento dopo la sua risposta;
  - **suspected**: non si ha ricevuto ricevuto risposta quindi si indica il processo come fallito, non è detto che sia corretto.

- **Failure detector affidabili**: capiscono certamente se un processo è fallito:
  - **unsuspected**: il processo non è fallito perché abbiamo ricevuto risposta, ma non si esclude il fallimento dopo la sua risposta;
  - **failled**: il dector ha determinato certamente che il processo è fallito.

Un failure detector ha due caratteristiche:

- **Completezza**: dopo un determinato instante di tempo un qualsiasi processo che è andato in crash viene permanentemente ritenuto fallito: La completezza può essere **forte** cioè un processo fallito è sospettato permanentemente da ogni processo corretto, in quella **debole** invece è sospettato solo da qualche processo;
- **Accuratezza**: esiste un istante di tempo dopo il quale nessun processo corretto è sospettato di fallimento. Anche qui abbiamo due livelli, **forte** in cui i processi corretti non sono mai sospettati falliti, e **debole** dove almeno un processo corretto non sarà mai sospettato fallito. Queste ultime due proprietà sono difficili da soddisfare in sistemi asincroni, quindi si introducono due nuove proprietà: **forte eventuale**, dove solo dopo un determinato instante di tempo i processi corretti non sono sospettati falliti e **debole eventuale** dove solo dopo un instante di tempo almeno un processo non è considerato fallito.

2.5.1 Implementazione failure detector

Per implementare un failure detector abbiamo bisogno di due processi p_i e p_j. Il compito di p_i è quello di monitorare p_j per capire se è fallito o meno. Questo può essere fatto tramite ping-ack o heart beat. Nell'heart beat è p_j ad inviare un messaggio heartbeat a p_i. Quest'ultimo se non riceve messaggi con una certa periodicità lo ipotizza come fallito. Nei sistemi asincroni è preferibile quest'ultimo.
Un failure detector inaffidabile può essere implementato con l'heart beat nel seguente modo:

- Ogni processo p_j invia un messaggio p_j-is-alive ogni T secondi ad un altro processo;
- Il detector calcola una stima massima del tempo per la trasmissione (Δ);
- se il processo p_i non riceve il messaggio is-alive entro T + Δ secondi, sospetterà che p_j sia fallito;
- se successivamente riceverà il messaggio non lo sospetterà più come fallito, altrimenti lo considererà fallito.

In un sistema sincrono si può implementare un detector affidabile, in cui se dopo T + Δ secondi non arriva un messaggio di heart-beat si può considerare il processo sicuramente fallito.

2.5.2 Chi fa il monitoraggio?

Per determinare quali processi devono monitorare chi esistono tre approcci:

- **centralizzato**: un unico processo riceve gli heartbeat e li monitora. Questo rappresenta un single point of failure;
- **Ad anello**: si crea un overlay con tipologia ad anello; Ogni processo invia l'heart beat al suo vicino. Più complessa da realizzare ed è vulnerabile a possibili problemi di rete;
- **Distribuito**: ogni processo invia una attay di heatbeat ad un sottoinsieme di tutti i processi. Genera un forte carito di rete. Per renderlo più efficiente si può utilizzare il gossiping. In questo approccio ogni processo ha un array in cui è contenuto l'ultimo instante in cui un processo ha inviato un heartbeat e il loro contatore. Quindi periodicamente un processo incrementa il proprio contatore heartbeat e invia l'array tramite il gossiping agli altri. Se l'heartbeat di un processo non viene incrementato per più di T secondi allora viene sospetatto, e dopo altri secondi si sospetta come fallito.

2.5.3 Consenso con failure detector

I failure detector possono essere inseriti all'interno di algoritmi di consenso. Il loro compito sarà quello di individuare i nodi falliti e andando ad escluderli. Questo porta ad una velocizzazione del consenso.

2.6 Consistenza dei dati

Nei sistemi di grande dimensione per aumentare la tolleranza agli errori e la velocità di accesso alle risorse si procede a replicare i dati, questo comporta un problema tutte le copie di un dato devono essere consistenti. Per far questo una qualsiasi modifica su una replica deve essere eseguita su tutte le sue repliche. Abbiamo due tipi di conflitti da risolvere:

- **lettura-scrittura**: abbiamo due copie di un dato, su una copia leggiamo e su di un'altra scriviamo;
- **scrittura-scrittura**: come prima ma stavolta effettuiamo due scritture di valori differenti.

Normalmente ci si aspetta che una lettura restituisca il valore scritto dalla scrittura più recente.

2.6.1 Modelli di consistenza

Specifica come determinare quale sia la scrittura più recente:

- **Stretta**: qualsiasi lettura su un dato x restituisce un valore corrispondente al risultato più recente di una scrittura su x, secondo un tempo globale. La scrittura è vista istantaneamente da tutti;
- **Linearizzabile**: gli accessi sono ordinati in base ad un timestamp. Tutti i processi vedono gli accessi condivisi nello stesso ordine;
- **Sequenziale**: Ogni inserimento di operazioni valide di lettura o scrittura è un comportamento accettabile, purché tutti i processi vedano gli stessi inserimenti;
- **Causale**: Tutti i processi vedono gli accessi condivisi correlati causalmente nello stesso ordine.

Un problema dei sistemi distribuiti è che sono partizionabili e questo ostacola la consistenza. Quando un sistema si divide le due parti non possono comunicare tra di loro. Possiamo procedere in due modi:

- Le due parti lavorano in maniera indipendente, il sistema è disponibile ma inconsistente;
- Si blocca tutto finchè non si riunisce il sistema, in questo caso il sistema diventa indisponibile.

2.7 Torema CAP

Il teorema afferma che in un sistema distribuito asincrono di grandi dimensioni, è impossibile garantire contemporaneamente la consistenza dei dati, la disponibilità del sistema e la tolleranza alle partizioni.

- **Consistenza (C)**: le diverse copie dei dati sono aggiornate;
- **Disponibilità (A)**: i dati sono disponibili anche in presenza di fallimenti;
- **Tolleranza alle partizioni (P)**

Visto che non possiamo rinunciare alla tolleranza alle partizioni i sistemi che possiamo ottenere sono di due tipi:

- **CP**: si rinuncia alla disponibilità, quindi le scritture potrebbero non essere eseguite quando il sistema non è disponibile;
- **AP**: si rinuncia alla consistenza dei dati, quindi si può sempre scrivere ma non è detto che una lettura porti ad avere dati aggiornati.

Eventual consistency: In un sistema distribuito, i dati potrebbero non essere immediatamente sincronizzati tra tutte le repliche, ma quando il partizionamento viene risolto man mano tutte le repliche vengono risolte. Un problema è decidere quale copia presente in una delle partizioni mantenere.

Capitolo 3
Introduzione alle blockchainzing

3.0.1 Transazioni

Le blockchain, inizialmente, nascono dall'esigenza di trasferire un bene sulla rete in maniera affidabile e sicura da un nodo all'altro senza mediatore.
Normalmente, infatti, nel trasferimento di denaro c'è sempre un mediatore (banca, ecc) che convalida, supervisiona e preserva le transazioni. Questa soluzione permette di evitare attacchi alle transazioni, ma il mediatore rappresenta il collo di bottiglia del sistema o single-point-of-failure.
Un sistema transazionale senza mediatore prevede una serie di protocolli per eseguire transazioni affidabili e sicure in un ambiente inaffidabile ed insicuro. I problemi da risolvere essenzialmente sono 3: **integrità del messaggio**, **identità dei nodi**, **validità della transazione**.
In generale l'obiettivo attuale della blockchain è quello di consentire in un contesto decentralizzato e senza mediatore il successo delle transazioni, quest'ultime non riguardano più scambio di soldi o criptovalute... ma anche trasferimento di informazioni.

3.0.2 Criptovalute

Una criptovaluta è una rappresentazione digitale di un valore basata sulla crittografia senza nessun intermediario. Si differenzia dalla moneta legale poiché la moneta legale è rilasciata dal governo, è governata da una banca centrale, è necessario un intermediario per effettuare transazioni e l'emissione è riservata ai soli soggetti autorizzati.
Una **valuta digitale** è la rappresentazione virtuale di una moneta legale.
Una **valuta virtuale** è una rappresentazione digitale di un valore che non è emessa da una banca né da un'autorità pubblica ed è accettato come mezzo di pagamento, possono essere archiviate o scambiate elettronicamente. Le criptovalute rientrano in questo caso. Le valute virtuali possono avere diverse caratteristiche, una valuta digitale si dice:

- chiusa: se ha valenza solo in un determinato contesto;
- aperta: se è possibile convertire la valuta legale in digitale ma **non** viceversa;
- bidirezionale: se è possibile convertire la valuta legale in digitale e viceversa.

Le criptovalute sono valute virtuali bidirezionali convertibili e decntralizzate.

3.1 Blockchain

3.1.1 Definizione di Blockchain

Un ledger (o libro mastro) è un registro che contiene tutte le transazioni eseguite tra i partecipanti di un sistema. Le transazioni sono scritte in ordine cronologico indicando l'oggetto a cui si riferiscono, in genere è una struttura dati di tipo **append-only multi-party system of record** (non si può cancellare ma solo scrivere e possono farlo più partecipanti).
La blockchain è un libro mastro distribuito di transazioni o eventi digitali condivisi tra partecipanti. Le informazioni sono memorizzate all'interno dei blocchi e per rendere il tutto più robusto, ogni blocco è collegato al precedente in modo tale da formare un catena di blocchi.
Ogni blocco è formato principalmente da 2 parti:

- **Header del blocco**, si va comporre di ulteriori 6 parti:
  - **Version**: versione delle attuali regole che un blocco deve seguire per ritenersi valido;
  - **Nonce**: nonce sta per **number once used** ed è quel numero che i miner devono calcolare per produrre un hash del blocco valido tramite un doppio hashing;
  - **Hash dell'header block precedente**: che si ottiene hashando l'header del block precedente, permette di collegare i blocchi;
  - **Timestamp**: indica in modo approssimato la data e l'ora in cui il blocco viene creato.
- **Lista delle transazioni**: elenco di tutte le transazioni che quel blocco andrà a contenere.

Siccome è un ambiente distribuito e non c'è un'entità di terze parti bisogna garantire i soliti requisiti ovvero integrità del blocco, identità dei nodi e validità del blocco.

La blockchain non è una sola tecnologia, ma nella blockchain risultano più tecnologie.
Infatti le blockchain si possono dividere in:

- **Blockchain Permissionless**: i partecipanti non devono essere autorizzati e non ci sono dei ruoli definiti. Esempi Bitcoin (BTC), Ethereum (ETH)
- **Blockchain Permissioned**: i vari nodi sono caratterizzati da ruoli. Alcune o tutte le operazioni possono essere svolta solo da ruoli autorizzati. (Hyperledger Fabric).

Un ulteriore classificazione è in:

- Public: tutti i blocchi sono visibili a tutti i nodi ed ogni nodo può partecipare al consenso (Bitcoin);
- Private: la blockchain è gestita da un organizzatore che decide quali nodi possono leggere i blocchi e partecipare al consenso;
- Consorzio: pochi nodi predeterminati possono leggere i blocchi e partecipare al consenso.

3.1.2 Livelli blockchain

Un sistema basato su blockchain è diviso in 3 livelli:

1. Livello di transazione: è il livello superiore della blockchain è vicino alle applicazioni. Definisce la codifica ed i criteri utilizzati per la generazione di transazioni e smart contract. (Transazione e smart contract);
2. Livello di generazione blocchi: è in questo livello che si definisce come una transazione deve essere validata a seconda dell'ambito e quale algoritmo di consenso utilizzare. (Regole validazione e meccanismi di consenso);
3. Livello di distribuzione: in questo livello si affrontano i problemi di inserimento dei blocchi nella catena e di come far sì che tutti i nodi del sistema abbiano la stessa visione globale della catena. (Blocchi minati)

Questo stack di livelli è una vera e propria infrastruttura che ha un valore economico sostanziale. A differenza del modello ISO/OSI la cui parte commerciale è solo il livello composto dalle applicazioni, poichè tutto il resto è standardizzato. **Cosa comporta?** Che nel contesto blockchain un'applicazione ha successo non solo per la logica ma anche per come sono adottati i tre strati sottostanti.

3.1.3 Come verificare l'integrità del blocco?

L'integrità del blocco in una blockchain può essere verificata utilizzando le **funzioni hash**. Infatti la funzione di hash crea una "impronta digitale" del blocco e nel contesto blockchain per verificarne l'integrità basta controllare che il suo hash sia uguale a quello del blocco precedente. Quindi aver legato in questo modo i blocchi ci garantisce che se un blocco è stato accettato, non può essere più modificato, rendendo il ledger IMMUTABILE!.
Tutto ciò potrebbe risultare in termini di risorse costoso, ma è stata prevista una struttura dati che velocizza il controllo di validità di un blocco ovvero il **Merkle Tree**. Infatti, tutte le transazioni all'interno di un blocco vengono riassunte in un albero di Merkle, producendo una sorta di impronta digitale dell'intero insieme di transazioni. In questo modo un utente è in grado di verificare se una transazione è inclusa o meno in un blocco.
Il Merkle Tree funziona in questo modo: il processo inizia con i dati originali divisi in coppie e hashati. I due risultati vengono hashati di nuovo creando così un nuovo livello dell'albero. Il processo continua fino a quando non si arriva ad un unico hash, la radice dell'albero.
Per verificare l'integrità, basta calcolare l'hash di un singolo dato fino alla radice confrontando gli hash calcolati con quelli nell'albero. se tutti corrispondono i dati sono integri.

3.1.4 Come autenticare i nodi?

Le blockchain utilizzano la firma digitale e le autorità di certificazione, tramite l'emissione di documenti elettronici noti come certificati digitali.

3.1.5 Come validare le transazioni?

La validazione delle transazioni è un processo fondamentale nella blockchain che garantisce che ogni transazione sia valida e che rispetti le regole della rete. Questo processo viene eseguito da nodi specifici che verificano la validità di ogni transazione prima che venga inclusa in un nuovo blocco.
La validazione di una transazione include anche la verifica di transazioni non duplicate e che i fondi non siano spesi due volte (double spending).
Per fare ciò bisgona poter identificre univocamente un messaggio contenente un blocco di transazioni eseguite. Per farlo si utilizza il **nonce**, ovvero un valore che cambia con il tempo e che ha una probabilità piccola di essere duplicato.

Proof of Work

Per generare il **nonce** le blockchain, e quindi validare le transazioni, utilizzano la **Proof of Work (PoW)**. Infatti, per proporre un nuovo blocco da aggiungere alla blockchain, un nodo deve dimostrare di aver utilizzato abbastanza risorse di calcolo per risolvere un "enigma crittografico", il cui risultato è utilizzato nel blocco stesso come nonce. Il lavoro richiesto per risolvere il puzzle è abbastanza oneroso così da imporre che chi vuole attaccare il sistema deve spendere molte risorse computazionali. Un attacco praticamente richiederebbe il 51% della potenza computazionale della rete. Ovviamente come incentivare i miner a partecipare alla validazione? Chi partecipa alla validazione viene ripagato con una buona somma.
Il sistema Proof of Stake è preferibile rispetto al sistema Proof of Work per il suo ridotto consumo energetico.

Proof of Stake

Nella **Proof-of-Stake**, letteralmente “prova di partecipazione”, chi vuole diventare un nodo validatore dunque deve depositare i token nativi della blockchain in un wallet adatto allo staking e avere una connessione a internet costante. I partecipanti possono poi puntare monete (“stake”) su un blocco ed è il protocollo ad assegnere a uno di loro il diritto di validare il prossimo blocco per poter ottenere le commissioni di transazione.
In genere, la **probabilità di essere scelti è proporzionale alla quantità di monete che si possiedono**: più monete hai, più alte saranno le probabilità di validare il prossimo blocco.

I criteri possono cambiare da blockchain a blockchain infatti in alcune i validatori possono essere scelti in base alla **quantità di token in staking**, oppure da quanto tempo i token sono in staking, o ancora tramite una funzione randomica. Alcune blockchain stabiliscono un minimo di criptovalute che devono essere depositate in staking, altre no.

La differenza di questi criteri determina variazioni del PoS come il **Delegated PoS**, nel quale i nodi che puntano di più votano per eleggere i validatori (vinco e delego altri).
Lo **stake**, la quantità di criptovalute possedute, funziona come una garanzia posta dal nodo per il corretto svolgimento del proprio lavoro di validatore. Qualora un nodo dovesse compiere errori o attività fraudolente, perderebbe tutti i suoi token.

In generale, il PoS viene considerato il meccanismo di consenso più decentralizzato, richiede minori barriere tecniche per partecipare alla rete, i nodi sono più distribuiti e di conseguenza la sicurezza è maggiore.

Una **critica** che viene mossa al PoS è quella di avvantaggiare i nodi pià ricchi che, avendo più criptovalute in staking, vengono selezionati più spesso per validare i blocchi e guadagnare gli incentivi. Tuttavia la grandezza dello stake è un incentivo a svolgere il lavoro di validazione correttamente e frequentemente. Più la posta in gioco è alta, più alto è il rischio di perderla quando si commettono degli errori nella validazione.

3.2 Consenso

Nella blockchain ogni nodo ha una copia del ledger e conosce quali sono i blocchi precedentemente convalidati che formano la catena. Questo è dovuto all'algoritmo di consenso utilizzato. **Infatti l'obiettivo dei protocolli di consenso è quello di garantire che tutti i nodi partecipanti concordino sullo stato della blockchain.**

Per raggiungere il consenso in contesti asincroni si possono rilassare i vincolo di consenso (Bitcoin) oppure rendere meno asincrono il sistema, sfruttando periodi di sincronia. Rilassando i vincoli di consenso è normale ottenere situazioni di non consenso dove non tutti i nodi hanno la stessa visione della catena. In questa situazione è possibile che due blocchi vengano considerati validi contemporaneamente, così da causare una *biforcazione* (**Blockchain Forking**).
I tipi di forking sono due:

- Il **soft fork** è un aggiornamento retrocompatibile, ovvero i nodi aggiornati possono comunque comunicare con quelli non aggiornati. La situazione è comunque temporanea, infatti, quando verrà prodotto un nuovo blocco questo farà riferimento al ramo più lungo e l'altro verrà considerato invalido. A questo punto si raggiungerà il consenso.
- Gli **hard fork** sono aggiornamenti software non retrocompatibili. Tipicamente, si verificano quando i nodi aggiungono nuove regole in conflitto con le regole dei vecchi nodi.

3.2.1 Consenso di Nakamoto (Bitcoin) o Nagatomo

L'algoritmo di consenso utilizzato in Bitcoin e nelle blockchain che adottano la PoW è il **consenso di Nakamoto**.
La primitiva di comunicazione utilizzata è il **gossiping**.

1. Ogni nuova transazione viene inviata in broadcast a tutti i nodi;
2. Ogni nodo collezione le nuove transazioni in un blocco secondo una propria politica di estrazione;
3. Ogni nodo attraverso la PoW calcola il valore del nonce tale che l'**hash del blocco** + **nonce** inizi con un numero stabilito di zeri. (RISOLVE IL PUZZLE CRITTOGRAFICO);
4. Quando un nodo trova il nonce giusto, invia in broadcast il blocco con le transazioni e il nonce.
   Attenzione: due nodi potrebbero inviare contemporaneamente blocchi differenti e si potrebbe creare un soft fork, che si risolve quando un ramo diventa più lungo di un altro) i nodi sul più corto lo abbandonano e si spostano sul più lungo;
5. Un nodo accetta un blocco se le transazioni sono valide (se chi ha iniziato la transazione può farlo), non ci sono double spending e che il blocco sia valido controllando l'hash;
6. I nodi manifestano l'accettazione del blocco aggiungendolo alla copia locale della blockchain al blocco precedente con l'hash di quest'ultimo.

3.2.2 Il puzzle crittografico in Nakamoto

In Nakamoto il puzzle crittografico (per dimostrare il lavoro, PoW) consiste nella generazione di un hash che soddisfi una certa quantità di zeri iniziali. Questo è un problema che può essere risolto solo attraverso una grande quantità di tentativi e quindi potenza di elaborazione (*bruteforce*). Quella macchina(o miner) che per prima trova il risultato si aggiudica la possibilità di scrivere un nuovo blocco in coda agli altri ed ha anche un riscontro economico in criptovalute.

3.2.3 Limitazioni e Vantaggi di Nakamoto

Vantaggi

Il puzzle garantisce la **sicurezza** della rete poiché per falsificare un blocco e far sì che tutta la rete creda in un blocco fake tendenzialmente servirebbe avere una capacità superiore alla metà di calcolo presente, quindi il classico 50% + 1 (o per semplicità **51%**) per poter creare più velocemente i blocchi degli altri e rendere approvato di fatto il blocco fake. In poche parole Nakamoto riesce a tollerare un numero di nodi bizantini finché la somma del loro potere computazionale non supera 50%.

Limitazioni

Dal momento in cui dalla creazione del blocco di una transazione fino al suo inserimento nella blockchain sono richiesti 10 minuti, l'algoritmo soffre di bassa frequenza di trasmissione dei dati (throughput). Per ovviare a ciò si potrebbe aumentare la dimensione del blocco, aumentando la latenza. Il limite più importante è che la PoW ha un sostanziale consumo energetico (PoW consuma 132 TWh di energia all'anno).

3.2.4 PBFT: Pratical Byzantine Fault Tollerance

L'algoritmo PBFT è uno schema **SMR** (di replicazione) che tollera i guasti bizatini a differenza dell'algortimo di Paxos. La comunicazione avviene tramite la primitiva del **gossiping**. PBFT è utilizzato in ***Hyperledger Fabric***.
L'algoritmo è simile a Paxos, ma ha una fase in più:

- **Fase di pre-prepare**: Il learner(client) invia una richiesta al leader (proposer), che la trasmette a tutti i nodi secondari di backup (acceptor) assegnando un numero di sequenza.
- **Fase di prepare**: I nodi secondari (acceptor) si scambiano informazioni ricevute dal proposer e si accordano;
- **Fase di commit**: quando i nodi secondari si accordano sull'ordine delle richieste, la richiesta esse viene eseguita e la risposta viene inviata al client (learner). Se il client (learner) riceve *f+1* risposte identiche si raggiunge il consenso, altrimenti no.

La replica delle informazioni inviate tra i nodi permettono di tollerare i guasti bizantini.
Infatti, siccome i nodi sono ordinati in base ad un identificativo, è possibile scegliere un nuovo leader per ogni round v e sarà quello con identificavo i = v mod(n). In questo modo se un acceptor si accorge che il proposer è bizantino può richiedere di eleggerne un altro (**view change**). L'algoritmo consente di tollerare f guasti anche di natura bizantina se il numero di nodi N è maggiore a *3f+1*.

3.2.5 Limitazioni e Vantaggi di PBFT

Vantaggi

La PBFT garantisce **fortemente** il requisito di **safety** in quanto tutti i nodi accettano la stessa transazione come valida solo se questa è stata effettivamente eseguita dalla maggioranza dei nodi. Tutto ciò si basa sul voto in tre fasi con MAC (Message Authentication Code) per la verifica dei messaggi.
La PBFT garantisce anche la **liveliness**, infatti client che lanciano la loro richiesta devono alla fine ricevere risposte e ciò implica che il il consenso deve essere raggiunto.

Limitazioni

Il **modello PBFT è suscettibile a un attacco Sybil**, in cui un nodo dannoso crea molti account duplicati nella rete e esegue azioni non autorizzate. Ciò potrebbe superare i nodi onesti, compromettendo così la rete.

PBFT è una promettente soluzione di consenso quando il gruppo di nodi è piccolo ma diventa **inefficiente per reti di grandi dimensioni**. Per questo motivo, la scalabilità e la capacità di throughput elevato del modello PBFT sono ridotte. Pertanto deve essere ottimizzato o utilizzato in combinazione con un altro meccanismo di consenso.

Approfondimento utile

PBFT non è generalmente utilizzabile in reti *permissionless* perché richiede un gruppo ben definito di nodi autorizzati che hanno il compito di partecipare al processo di consenso.

SMR - State Machine Replication (Accenno)

Lo schema SMR è un metodo per implementare un servizio tollerante ai guasti basato sulla replicazione e il coordinamento dei server. Un approccio del genere implica appunto la replicazione di una macchina a stati su più istanze di un sistema. Ciascuna delle repliche della macchina a stati viene inizializzata con lo stesso stato iniziale. Quando il sistema riceve una richiesta client, ciascuna delle repliche del sistema elabora la richiesta e, in base all'output generato, aggiorna i propri stati individuali. In uno scenario normale, lo stato risultante di ciascuna delle repliche sarà identico. In caso di guasto o anomalia, non ci sarà consenso sul nuovo stato del sistema.

SMR è spesso realizzato in maniera leader-based, con un server primario 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.

Note: Paxos è uno schema SMR tollera i guasti di tipo crash ma non quelli bizantini.

Capitolo 4
Smart contracts, Chaincode e alcune blockchain

4.1 Smart Contracts

Inizialmente la blockchain non era programmabile perchè nasceva per lo scambio di criptovalute e il controllo dell'esito della transazione. Il contratto è un accordo tra due o più parti che definisce i loro obblighi e diritti, è vincolante per le parti che lo hanno stipulato e stabilisce come verranno risolte eventuali dispute.

Uno **smart contract** è un contratto digitale automatizzato che esegue automaticamente (*deterministico*) le condizioni concordate tra le parti. Scritto in un linguaggio di programmazione ed immutabile, il che significa che una volta creati, i termini del contratto non possono essere modificati (sono considerati proprio blocchi).

Uno smart contract per essere definito tale deve rispettare i seguenti requisiti:

- **osservabilità**: l'output deve essere sempre legato all'input;
- **verificabilità**: bisogna poter testare il codice;
- **riservatezza**: i dati riservati devono essere privati;
- **applicabilità**: è possibile applicarlo ad una determinata blockchain;

4.1.1 Ciclo di vita

- **Fase 1 (Scrittura smart contract)**: Gli sviluppatori scrivono gli smart contract in un linguaggio di prorammazione desiderato, lo si compila e si ottiene il bytecode;
- **Fase 2 (Pubblicazione smart contract)**: lo smart contract viene pubblicato sulla piattaforma blockchain come una transazione, quindi viene aggiunto ad un blocco da validare;
- **Fase 3 (Interazione con smart contract)**: Uno smart contract viene richiamato come una transazione in cui si indica il metodo da invocare ed eventuali parametri. La transazione viene inserita nel pool delle transazioni della blockchain in attesa di essere eseguita e convalidata. Per garantire che le chiamate terminino al client viene addebitata una **fee** ad ogni chiamata (se l'addebito supera quello che il cliente è disposto a pagare, il calcolo viene interrotto).
- **Fase 4 (Validazione dell'interazione con lo smart contract)**: La blockchain selezionerà la transazione da eseguire e convalidare e nel caso di una transazione che richiama uno smart contract, i nodi eseguiranno la funzione richiesta ed otterranno un risultato, Questi risultati vengono comparati e il determinismo ne implica l'uguaglianza.
- **Fase 5 (Scrittura risultato finale in un blocco)**: il risultato verrà inserito in un blocco da aggiungere alla blockchain.

Uno smart contract può essere **deterministico ma non concorrente** in quanto nel caso di esecuzione di due smart contract in parallelo non è detto che uno smart contract veda le modifiche apportate dall'altro. Conseguenza del fatto che il risultato prodotto dall'esecuzione dello smart contract deve essere messo in un blocco e bisogna comunque aspettare la sua approvazione. Rientra nel caso anche l'esecuzione di due smart contract in maniera sequenziale (il secondo non deve supporre che il primo ha finito).

4.2 Chaincode

Gli smart contracts di Hyperledger Fabric sono scritti in **chaincode**, programmi scritti in Go e Node.js che implementano un'apposita interfaccia ed eseguiti in un contenitore Docker isolato dal processo dell'endorser peer. Lo stato creato da un chaincode è limitato esclusivamente a quel chiancode e non è possibile accedervi direttamente da un altro. 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 viene deployato per la prima volta o riceve un'istanza o una transazione di aggiornamento in modo da eseguire qualsiasi inizializzazione necessaria, inclusa quella dello stato dell'applicazione;
- *Invoke()*: viene chiamato in risposta alla ricezione di una richiesta di accesso ad una delle funzionalità offerte dal chiancode.

4.3 Bitcoin

La blockchain della cryptovaluta **Bitcoin** contiene blocchi di transazioni codificate secondo il modello UTXO (Unspent Transaction Output). Ogni volta che viene effettuata una transazione Bitcoin, l'importo totale inviato viene diviso in più output, ognuno dei quali rappresenta una quantità specifica di bitcoin. Questi output possono essere utilizzati come input in transazioni successive.

IMPORTANTE: Bitcoin non ha un saldo

Perchè avere saldo vuol dire avere uno stato e in un sistema asincrono avere uno stato è una limitazione, inoltre in Bitcoin il consenso non è sempre garantito.
Per rappresentare una sorta di bilancio si utilizzano gli **output di transazione non spesi** (UTXO) che **rappresentano il "bilancio"** disponibile di un indirizzo Bitcoin. Se un indirizzo ha un output di transazione non speso, significa che c'è ancora una quantità di bitcoin associata a quell'indirizzo che non è stata ancora utilizzata.

ESEMPIO DI TRANSAZIONE IN BITCOIN

Supponiamo che Alice voglia inviare 5 bitcoin a Bob. Ecco come potrebbe svolgersi la transazione:

1. Alice inserisce l'indirizzo Bitcoin di Bob come destinatario della transazione e specifica la quantità di bitcoin che desidera inviare, in questo caso 5 bitcoin.
2. Il portafoglio di Alice cerca un output di transazione non speso associato al suo indirizzo che contenga almeno 5 bitcoin. Se non trova un solo output che soddisfi questa condizione, potrebbe utilizzare più output per sommare a 5 bitcoin;
3. Una volta individuati gli output adeguati, il portafoglio di Alice crea una nuova transazione che specifica gli output selezionati come input e l'indirizzo di Bob come destinatario;
4. La transazione viene quindi inviata alla rete Bitcoin, dove viene verificata e confermata da diversi nodi;
5. Una volta che la transazione è stata confermata e registrata nella blockchain, i 5 bitcoin vengono trasferiti dal bilancio di Alice a quello di Bob.

4.4 Ethereum

Ethereum è una piattaforma software open-source per lo sviluppo di smart contracts e applicazioni implementate attraverso una blockchain permissionless. La blockchain crea un world state, ovvero l'insieme di tutti gli stati degli smart contract e dei vari account all'interno della blockchain Ethereum. La valuta digitale gestita è l'Ether ed ogni operazione all'interno della blockchain richiede un certo costo **(gas**) espresso in **wei**. Questi costi sono utilizzati per le reward. L'algoritmo di consenso utilizzato, dopo il merge è un algoritmo PoS. In Ethereum c'è il concetto di accounts, a differneza di Bitcoin ogni account ha uno stato che indica il saldo corrente, lo stato non è memorizzato nella blockchain ma in un albero esterno (**Merkle Patricia**).
Ethereum ha due tipi di account:

- **Account di proprietà esterna (EOA)**: controllato da chiunque disponga di una coppia di chiavi pubbliche e private. Possono essere utilizzati per avviare transazioni e firmarle, per inviare e ricevere Ether/token e per richiamare contratti;
- **Account del contratto (contract account):** un contratto intelligente distribuito in grado di inviare transazioni solo in risposta alla ricezione di una transazione. Inoltre, possono essere utilizzati per inviare e ricevere Ether e interagire con altri contratti intelligenti. Non hanno chiavi private, infatti per autorizzare un azione su uno smart contract specifico si utilizza la chiave privata del proprietario.

Ogni account è identificato da un indirizzo di 20 byte. L'indirizzo di un account EOA ha un prefisso "0x" seguito dagli ultimi 20 byte dell'hash della chiave pubblica. Invece, gli indirizzi dei contract accounts sono l'hash dell'indirizzo del creatore più il numero di transazioni inviate dal creatore al momento del deploy (nonce).

Invece, una transazione ha la seguente struttura:

- **from**: l'indirizzo del mittente;
- **signature**: la firma della transazione generata con la chiave privata del mittente. Tale firma è verificabile utilizzando la chiave pubblica del mittente;
- **to**: l'indirizzo del destinatario;
- **amount**: quantità di Ether o Wei trasferita.

Se la transazione è verso uno smart contract si aggiunge il parametro data. Tale parametro può contenere l'hash del metodo dello smart contract che si vuole invocare più eventuali parametri. Nel caso in cui la transazione è di ritorno da uno smart contract ad un account personale allora in data c'è il valore di ritorno. L'invio di una transazione viene pagato solo dal mittente.

4.5 Hyperledger Fabric

Hyperledger Fabric si differenzia da altri sistemi blockchain è che è **privato e permissioned**. Invece che essere un sistema aperto permissionless che permette a identità sconosciute di partecipare alla rete.
Hyperledger Fabric inoltre offre la possibilità di creare **canali**, permettendo a un gruppo di partecipanti di creare un ledger di transazioni separato. Questa è un'opzione particolarmente importante per le reti in cui alcuni partecipanti potrebbero essere concorrenti e non volere che ogni transazione fatta da loro sia nota a tutti i partecipanti. Se due partecipanti formano un canale, questi due partecipanti, e nessun altro, possiedono le copie del ledger per quel canale.

Hyperledger Fabric ha un sottosistema ledger comprendente due componenti: il **world state** e il **transaction log**. Ogni partecipante ha una copia del ledger di ogni rete Hyperledger Fabric a cui appartiene.

Il componente **world state** descrive lo stato del ledger ad un dato punto nel tempo. E' il database del ledger. Il componente **transaction log** registra tutte le transazioni che sono risultate nel valore attuale del world state; è lo storico degli aggiornamenti per il world state. **Il ledger, quindi, è una combinazione del database world state e dello storico del log delle transazioni.**

Capitolo 5
Attacchi e vulnerabilità delle blockchain

La maggior parte degli attacchi alla blockchain ha lo scopo di svalutare la criptovaluta ad essa collegata. I punti critici della struttura della blockchain che subiscono maggiormente questi attacchi sono:

- Attacchi alla struttura della blockchain;
- Attacchi alla rete P2P;
- Attacchi alle applicazioni che utilizzano la blockchain;

Le blockchain pubbliche sono quelle più vulnerabili, quelle private per via che richiedono un autenticazione sono meno vulnerabili.

5.1 Attacchi alla struttura

Gli attacchi rivolti verso la struttura della blockchain mirano a causare dei fork. Si cerca di generare un fork perchè questo rappresenta uno stato di incoerenza all'interno della rete e quindi si può sfruttare per eseguire transazioni fraudolente oppure creare della sfiducia da parte dei nodi verso la rete.
Gli attacchi che generano una **soft-fork** puntano sempre a lasciare dei blocchi validi fuori dalla rete nel momento in cui il soft-fork si risolve. Abbiamo due tipologie di blocchi lasciati fuori:

- **Blocco orfano**: è un blocco valido ma ha come genitore un blocco non pià valido. La sua generazione può causare un ritardo nella conferma delle transazioni;
- **Blocco obsoleto (stale)**: è un blocco minato con successo e valido, ma non si trova sulla catena principale. Presente principalmente nelle reti pubbliche a causa della race condition tra i miner per la pubblicazione di un blocco. Può essere generato anche dall'attacco **selfish mining**.

L'algoritmo di consenso utilizzato determina le vulnerabilità che si possono verificare:

- **PoW**: la possibilità che la capacità di hashing sia centralizzata in pochi nodi rende la rete suscettibile ad **attacchi 51%** o **double spending**, per effettuare uno di questi due attacchi c'è bisogno di avere più del 50% computazionale della rete;
- **PoS**: anch'essa è suscettibile ad un attacco 51% anche se in questo caso si dovrebbe avere il controllo sulla maggioranza della valuta disponibile.
- **PBFT**: la sua bassa scalabilità la rende suscettibile ad attacchi di tipo sybil.

5.2 Attacchi al P2P

Sono attacchi alla rete della blockchain. Gi attacchi possibili sono i seguenti:

- **Selfish mining**: un miner malizioso fa il mining dei blocchi ma non li pubblica subito. Li accumula ed aspetta che si verifichi un fork, a quel punto li pubblica creando così un branch più lungo e prendendo così lui le reward. Si cerca di eseguire **double spending** e **fork-after-witholding**;
- **double spending**: un attaccante invia due transazioni in rapida successione con l'obiettivo di invalidare la prima (A) attraverso la seconda (B); quest'ultima contiene informazioni sul trasferimento degli stessi fondi ad un wallet di sua proprietà;
- **fork-after-witholding**: un miner malizioso si trova su due pool di mining. Calcola la Pow sul primo blocco ma non pubblica la soluzione. La pubblica solo quando il secondo blocco la pubblica. Questo genera un fork che sarà risolta solo quando una delle due pubblicherà un blocco in più. L'attacante vince sempre;
- **Attacco 51%**: è un attacco che si verifica quando nodi sybil (falsi/fittizi) o un pool di miner possiede la maggior parte dell'hash rate/stake. A questo punto si può manipolare la rete per impedire di validare blocchi o per effettuare double spending;

- **DNS Hijacking**: Quando un nodo si unisce alla rete Bitcoin per la prima volta, non è a conoscenza dei peer attivi. Per scoprirli utilizza il DNS. Si potrebbe quindi effettuare un poisoning della cache DNS per sostituire alcuni indirizzi IP del server DNS per reindirizzare le richieste verso una rete gestita dall'attaccante;
- **BGP Hijacking**: si manomettono le tavole BGP per reindirizzare il traffico da server mining leggittimi su una rete con server fasulli. In questo modo le rewards saranno riscosse dai server fasulli dell'attaccante;
- **Eclipse attack**: l'attacco mira ad oscurare la visione di un partecipante della rete. Così da potergli inviare blocchi falsi per rendere maizioso il nodo onesto;
- **Attacco DDos**: un attaccante che controlla diversi nodi sybil inizia a scambiare transazioni dust (transazioni di pochi centesi) tra i suoi nodi. Il suo scopo è quello di congestionare la rete che portare al rifiuto di alcune transzioni e negazioni di servizi verso utenti legittimi;
- **Attacco Finney**: Il miner genera una transazione, calcola un blocco e sceglie di non ritrasmetterlo. Poi genera un duplicato della sua transazione precedente e lo invia a un destinatario. Quando il destinatario valida la transazione , l'attaccante rilascia il suo blocco invalidando la transazione del destinatario. Questo genera un double spending;
- **Ritardare il consenso**: nelle reti private basate su PBFT un utente malizioso può iniettare blocchi falsi per ritardare il consenso.
- **Timejacking attack**: Ogni blocco ha un timestamp che rappresenta il tempo approssimativo della sua creazione, se questot empo di creazione supera una soglia il blocco viene scartato. Un'attaccante può utilizzare dei nodi sybil per trasmettere informazioni errate sull'ora della rete, in questo modo l'orologio dei nodi si distacca da quello della rete e questo porta a uno scarto dei blocchi;

5.3 Attacco Alle applicazioni

Si attaccano le applicazioni che si collegano alla blockchain sfruttando il fatto che quando una transazione è approvata questa è irreversibile.

5.3.1 Criptojacking

Si tratta di un attacco che viene lanciato su servizi web e cloud per eseguire illegalmente PoW senza che l'utente ne sia consapevole. Sostanzialmente vengono rubate risorse di elaborazione ai dispositivi di diversi utenti.

5.3.2 Furto del wallet

I principali attacchi sono volti a rubare le credenziali di wallet. In Bitcoin, di default il wallet viene archiviato non crittografato. Le credenziali di wallet vengono memorizzate sui dispositivi degli utenti, però possono essere rubate mediante attacchi di ingegneria sociale.

5.3.3 Furto della chiave privata

Quello che si cerca di rubare è la chiave privata di un utente. Se un attaccante possiede la chiave privata di un utente allora potrà firmare e generare transazioni spendendo il saldo dell'utente.

Capitolo 6
Trusted Execution Environment

6.1 Trusted Computing (TC)

Il TC è un insieme di tecnologie, protocolli e standard che mirano a garantire la sicurezza e la privacy dei dati e delle informazioni gestiti da un computer o da un sistema informatico. Si basa sulla creazione di un ambiente di elaborazione affidabile, che viene creato utilizzando hardware, software e tecnologie di sicurezza integrate. Oltre al TEE una tecnica per applicare il TC è la **Software Fault Isolation (SFI)** in cui si crea una area di memoria per eseguire un codice non attendibile così se viene rilevato un comportamento anomalo si lancia un errore e non si rischia di intaccare le componenti software.

Un esecuzione isolata si può implementare in due modi virtualizzazione e containerizzazione.

6.1.1 Virtualizzazione

La virtualizzazione è un processo che crea un ambiente virtuale, separato dal sistema hardware al fine di emulare o simulare l'esecuzione di un SO, server, ecc. Questo consente a più sistemi operativi di funzionare sulla stessa macchina fisica e di condividere le risorse, riducendo i costi, migliorando l'utilizzo della risorsa e fornendo maggiore flessibilità e affidabilità. Comporta anche delle limitazioni:

- **Dipendenza da un hypervisor**: un hypervisor è un software che permette di creare e gestire ambienti virtuali(VMs) su un sistema fisico. L'hypervisor funge da intermediario tra le VMs e l'hardware del sistema, assegnando risorse hardware (come CPU, memoria e spazio su disco) alle VMs e gestendo le richieste di accesso alle risorse da parte delle VMs. Gli hypervisor possono essere hardware o software. Per questo rappresentano un **Trusted computing base (TCB)** nella virtualizzazione. Una TCB di un sistema è l'insieme di tutte le componenti hardware e software critiche per la sicurezza e che quindi se intaccate compromettono il sistema.
- **Mancata gestione dell'hypervisor o dei rootkit del firmware**: tramite l'uso di un rootkit (insieme di software dannosi) si può riuscire ad accedere dalla macchina virtuale a quella fisica;
- **Sovraccarico del sistema**: la virtualizzazione sovraccarica il sistema e causa un calo delle prestazioni.

6.1.2 Containerizzazione

La containerizzazione permette di raggruppare tutti i componenti e le dipendenze di un'applicazione in un singolo contenitore isolato, detto "container". In questo modo, l'applicazione, completa di tutte le componenti necessarie, può essere spostata ed eseguita in tutti gli ambienti e infrastrutture. L'indipendenza del container e la caratteristica di estrema portabilità, permette eseguire un'applicazione su qualsiasi sistema operativo, senza riscontrare alcun problema di compatibilità. Un esempio di container è Docker. Un container per gestire l'ambiente di esecuzione utilizza un daemon. Questo daemon rappresenta la parte vulnerabile di un container:

- deployare immagini con codice dannoso;
- deployare immagine benigne che durante l'esecuzione scaricano codice dannoso;
- un attaccante può ottenere informazioni sensibili dal registro docker così da poter compromettere la macchina.

6.2 Soluzioni hardware

L'utilizzo di componenti hardware fornisce strumenti di difesa da compromissioni, consente cambi di contesto più veloci e permette di separare le applicazioni in due contesti di esecuzione:

- normale: dove vengono eseguite le applicazioni e SO normali;
- sicuro: è dove viene eseguito il software critico.

Un ambiente si dice sicuro se garantisce le seguenti tre proprietà:

- **Esecuzione isolata**: ogni applicazione viene eseguita in maniera indipendente dalle altre. Un'applicazione dannosa non può accedere/modificare i dati sensibili conservati da altre applicazioni protette, e inoltre non può alterarne l'esecuzione.
- **Archiviazione sicura**: l'integrità e la segretezza sono garantite per tutti i dati;
- **Provisioning sicuro**:garantisce la sicurezza delle applicazioni di terze parti.

Alcuni esempi

- **Secure elment (SE)**: le schede SIM incorporano un cip detto SE che memorizza i dati sensibili ed esegue le app in modo sicuro su questi dati.
- **Enrypted 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.
- **Trusted Platform Module (TPM)**: è un chip progettato per offrire funzioni legate alla sicurezza dell'hardware che si sta utilizzando. Si occupa di eseguire operazioni crittografiche complesse partendo dalla crittografia simmetrica a quella asimmetrica. Offre agli sviluppatori delle API per poter invocare le diverse funzioni crittografiche.

6.3 TEE

Un TEE è una componente hardware o software che fornisce un ambiente di esecuzione protetto per l'elaborazione di dati sensibili e garantisce che i dati riservati vengano mantenuti protetti da accessi non autorizzati. Questo ambiente è separato dal sistema operativo principale del dispositivo e utilizza una propria memoria, processore e sistema operativo indipendenti. Il TEE è un componente del TC. Divide il sistema in due ambienti di esecuzione:

- **Trusted**: vengono eseguite solo le operazioni sensibili. Il TCB è estremamente ridotto così da minimizzare gli attacchi;
- **Rich Excecution Environment**: abbiamo il SO e le applicazione tradizionali. Qui il TCB è molto ampio.

La comunicazione tra queste due parti avviene tramite un canale sicuro. Il TEE fornisce anche un archiviazione sicura, i dati critici e sensibili infatti sono isolati dai dati dell'utente così da migliorarne la sicurezza. Infine protegge anche il bootstraping, infatti prima di caricare il bootloader verifica l'autenticità e l'integrità del SO.

6.4 TEE nelle blockchain

Al momento TEE viene utilizzato nelle soluzioni Blockchain di ricerca. Si vuole utilizzare un TEE per semplificare e migliorare alcune funzionalità delle blockchain.

6.4.1 Hyperledger Sawtooth

In Sawtooth oltre a specificare che architettura utilizzare (scelta dei peer, chaincode e collegamenti tra peers), possiamo scegliere le regole di transazione, i permessi e l'algoritmo di consenso da utilizzare. Lo stato è rappresentato su ogni nodo validatore tramite un albero Merkle-Radix. Si può decidere quale algoritmo di consenso utilizzare anche se la rete è già attiva e in esecuzione. La comunicazione avviene attraverso API e protocollo P2P. Uno di questi algoritmi è il **Proof of Elapsed Time (PoET)**. Questo algoritmo è una soluzione del problema dei generali bizantini ed 'e normalmente tollerante ai crash. Eseguendolo in un ambiente TEE diventa tollerante ai fault bizantini, in quanto un TEE evita che il codice sia modificabile esternamente. L'algoritmo rientra tra i vari algoritmi a lotteria come Nakamoto.

6.4.2 Hyperledger Avalon

Le elaborazioni possono essere:

- **On-chain**: sono elaborazioni effettuate in blockchain (esecuzione di smart contract) e con un consenso globale;
- **Sidechain**: sono elaborazioni effettuate in blockchain ma su un consenso locale (pochi nodi) e non globale. Qui si ha una catena locale e bisogna proteggere le interazione tra questa e quella globale;
- **Off-chain**: elaborazioni effettuate fuori dalla blockchain su dati estratti dalla catena.

Le elaborazioni off-chain sono le più efficienti ma anche le più vulnerabili, la blockchain potrebbe riceve dati sbagliati. Possono essere protette con un TEE. Per farlo le elaborazioni sono definite come dei workers all'interno di un enclave, così da non poter essere compromessi dall'esterno. Le richieste verso i workers sono inviate tramite interfacce utente front-end oppure strumenti a riga di comando. Queste richieste possono essere effettuate dagli smart contract o client application. Ci sono due modelli:

- modello proxy: utilizzato dagli smart contract per richiede funzionalità;
- modello diretto: utilizzato dalle client application tramite delle API.

Tutte le comunicazione tra front-end e i workers avvengono tramite il KV Storage Manager, che mantiene una directory dei workers con tutte le informazioni sull'attestazione, il tipo e i parametri di esecuzione. Mantiene anche una coda che contiene tutte le richieste ancora non soddisfatte.
Il **Worker Manager** si occupa della creazione delle chiavi crittografiche dei worker, della creazione e manutenzione dei pool di worker. Il **Worker Queue Manager** gestisce la coda di richieste per un dato worker eliminando quelle vecchie, limita il flusso di richiese per un dato worker ed avvia l'esecuzione di una richiesta tramite uno o pi'u adattatori. I worker possono essere singleton oppure pool.

Capitolo 7
GDPR e Firma elettronica

Quando si parla di **sicurezza** intendiamo la confidenzialità, l'integrità e la disponibilità di dati/servizi utilizzati dagli utenti. Quando invece parliamo di **privacy** invece quello che intendiamo è garantire che non vengano violati i diritti sanciti dalla legge per le persone.
Per questo l'unione europea nel maggio del 2016 ha pubblicato il **GDPR (Regolamento Generale sulla protezione dei Dati)**

7.1 GDPR

Il GDPR chiarisce come i dati personali di un utente debbano essere trattati, incluse le modalità di raccolta, utilizzo, protezione e condivisione. Per **dati personali** nel contesto del GDPR si intende tutte quei dati che possono portare all'identificazione di una persona. Non ha importanza che sia un username o un dato crittografato, se si può risalire alla vera identità di un utente questi devono rispettare le regole sancite dal GDPR.
Trattando solo di dati personali il GDPR non si applica ai dati aziendali (indirizzo di un azienda, nome di un azienda, ecc.), ma si applica ai dati che identificano le persone che lavorano in un azienda.
Il GDPR non si applica anche quando i dati eprsonali sono trattati da autorità competenti duranti indagini, da istituzioni, organi, uffici legati alla UE e quando si utilizzano per attività personali (la nostra rubrica telefonica).
Per la gestione della privacy il GDPR impone due paradigmi:

- **Privacy by Design**: durante la creazione di un sistema informatico il come gestire i dati personali ha la stessa importanza dei requisiti funzionali, quindi la sicurezza dei dati deve essere fondamentale in ogni fase di sviluppo del sistema;
- **Privacy by Default**: alla creazione di un sistema si devono già individuare quali dati devono essere protetti e le impostazioni di privacy devono essere già attive per un utente senza che quest'ultimo faccia qualcosa.

Data breach : il titolare dell'azienda deve avvisare tutti i suoi utenti entro 72 ore dal data breach.

Coockie law : è una legge che regola l'utilizzo dei cookie sul Web. la legge richiede che i siti web informino gli utenti sull'utilizzo dei cookie e ottengano il loro consenso prima di archiviarli sul loro computer.

Storage dei dati : il GDPR impone che i dati sensibili siano memorizzati in storage presenti geograficamente in europa.

7.1.1 Diritti degli utenti

Il GDPR sancisce anche quali sono i diritti degli utenti

- **Diritto ad essere informati**: Le organizzazioni devono chiarire ai loro utenti come intendo utilizzare i dati che raccolgono;
- **Diritto di accesso**: Gli utenti hanno il diritto di accedere ai propri dati personali e alle informazioni relative alle modalità di trattamento degli stessi, inoltre l'organizzazione deve fornire gratuitamente all'utente che ne fa richiesta una copia dei suoi dati personal;
- **Diritto di rettifica**: Gli utenti hanno il diritto di richiedere la rettifica dei loro dati personali se sono imprecisi o incompleti. Questo diritto implica anche che la rettifica debba essere comunicata a tutti i soggetti terzi coinvolti nel trattamento dei dati in questione, a meno che ciò non sia impossibile o particolarmente difficile;
- **Diritto ad opporsi**: gli utenti hanno il diritto ad opporsi a come vengono trattati i loro dati personali presentando un motivazione valida;
- **Diritto alla portabilità dei dati**: l'utente ha il diritto di ottenere (in un formato elettronico leggibile) i propri dati personali allo scopo di trasferirli ad altro titolare, senza che l'attuale titolare crei alcun ostacolo;
- **Diritto all'oblio**: l'utente ha il diritto di chiederne la cancellazione nonché la cessazione di ogni altra forma di diffusione dei suoi dati personali;
- **Diritto alla limitazione**: l'utente ha il diritto di richiedere la limitazione del trattamento dei suoi dati personali;

7.1.2 Data Protection Officer (DPO)

Il suo ruolo è quello di monitorare la corretta esecuzione del GDPR e di comunicare eventuali violazioni. Il DPO dovrebbe inoltre essere competente nella gestione dei processi informatici, nella sicurezza dei dati e in altre questioni critiche relative al trattamento di dati personali e sensibili.

7.1.3 Data Protection Impact Assessment (DPIA)

È un processo utilizzato per aiutare le organizzazioni a rispettare efficacemente il GDPR e a garantire che i principi di responsabilità, privacy by design e privacy by default siano effettivamente messi in pratica dall'organizzazione. Il processo di DPIA deve essere documentato per iscritto.

7.2 Blockchainzing e GDPR

Quando si adotta questa tecnologia si deve far molto attenzione perchè viola diversi punti del GDPR.

- **Violazione privacy by design**: questo principio è violato perchè un qualsiaci transazione su blockchain richiede l'invio di una richiesta a tutti i miner per la sua convalida e questa potrebbe contenere potenzialmente dei dati personali. Ad ogni transazione si aggiunge un nuovo blocco con consegente comunicazione a tutti i miner;
- **Violazione dello storage**: i dati sensibili della blockchain non è detto che siano memorizzati soltanto in Europa essendo quest'ultima una tecnologia decentralizzata;
- **Impossibilità diritto all'oblio**: poiché i dati sulla blockchain sono permanenti e immutabili, la cancellazione effettiva non è possibile (è possibile cancellare un informazione ma questo genera un nuovo blocco, quindi è sempre presente nei blocchi precedenti), questo è in contrasto il GDPR. Si sta pensando a come rendere possibile questo utizzando diversi approcci:
  - cancellare la chiave con cui si sono cifrati i dati rendendo così i dati indecifrabili;
  - inserire nella blochain solo l'hashing dei dati e memorizzare i dati effettivi fuori dalla blockchain, ma questo implica il proteggere anche quest'ultimi poi;
  - utilizzare tecniche di anonimizzazione.
- **Problema nel definire un DPO**: essendo una tecnologia decentralizzata è difficile definire chi è il DPO. Se tutte le organizzazioni si conosco (Fabric, blochchai-permissioned) ogni organizzazione potrebbe elleggere il proprio DPO e poi tutti i dippio si coordinerebbero tra di loro. Ma se parliamo di blockchain permissionless questo diventa problematico perchè le organizzazioni si collegano e scollegano a loro piacimento, a quel punto il DPO andrebbe ricercato tra gli sviluppatori degli Smart Contracts(in quanto trattano i dati personali per conto del titolare) e i minatori(poiché eseguono le istruzioni del titolare quando verificano che la transazione soddisfi i criteri tecnici).

7.3 Firma elettronica

È necessaria per conferire validità legale ai documenti informatici quando per esempio si sottoscrivono contratti o atti amministrativi. Questo implica che devono avere lo stesso valore di una **firma autografa** (firma che scrivi a mano). Esistono diverse tipologie di firme:

- **Firma elettronica semplice/debole (FES)** : sono dei dati elettronici associati ad altri dati che sono poi utilizzati come firma (es: immagine + età). È facilmente falsificabile;
- **Firma elettronica avanzata (FEA)**: è un tipo di firma elettronica che garantisce un'identificazione univoca del firmatario e fornisce una garanzia legale dell'autenticità del documento o del contratto firmato. QUesto è possibile perchè utilizza tecnologie come la crittografia a chiave pubblica per garantire l'integrità del documento e l'identità del firmatario;
- **Firma elettronica qualificata (FEQ)**: è una FEA ma creata da un dispositivo avente un certificato qualificato per le firme elettroniche;
- **Firma digitale (FD)**: è una FEQ basato su un sistema di chiave pubblica e privata. La privata consente al titolare di firmare, la pubblica consente di verificare l'integrità e la provenienza di uno o più documenti;

Per verificare l'autenticità di una firma elettronica si fa utilizzo di certificati per dimostrare la validità di quest'ultima. Il certificato deve essere emesso da una CA affidabile e non deve essere scaduto.

7.3.1 Busta crittografica

Quando apponiamo una firma digitale ad un documento, quello che andiamo a realizzare è una **busta crittografica**, overro un file che contiene il documento originale, la firma digitale e la chiave di verifica. Esistono diversi formati di busta crittografica. Tutti i formati devono avere le seguenti caratteristiche:

- **aperti**: devono essere disponibili a chiunque;
- **non proprietari**: devono seguire uno standard;
- **robusti**: in grado di recuperare tutto o una parte di un documento danneggiato;
- **stabile**: compatibili con versioni precedenti o future;
- **sicuri**: rispetto ai virus;
- non contenere **macroistruzioni**: cioè un insieme di istruzioni eseguibili con un comando.

Sigillo elettronico : è l'equivalente digitale del timbro giuridico.

7.4 Posta elettronica

La posta elettronica è un servizio a cui possono accedere gli utenti collegati a Internet per spedire e ricevere messaggi. Questo avviene tramite il protocollo **SMTP**. Esite anche lo standard **MIME**, quest'ultimo aggiunge la possibilità di inserire caratteri diversi da quelli ASCII e permette l'aggregazione di diversi messaggi tra di loro.

7.4.1 Posta elettronica sicura

- **SMTPS**: utilizza il protocollo SSL/TLS per autenticare le due parti scambiando dei certificati, instaura un canale di comunicazione sicura. La sicurezza è a livello di trasporto.
- **S/MIME**: è uno standard per la crittografia a chiave pubblica e la firma digitale di messaggi di posta elettronica in formato MIME. La sicurezza è a livello di messaggio.

Nessuno di questi standard certifica a livello legale l'invio di un email. A tal proposito è nata la **PEC** che ha la stessa valenza legale di una raccomandata.

Codici Bitcoin e Go lang

A.A. 2022/2023

Indice

Capitolo 1

Script Bitcoin

Il linguaggio Script di Bitcoin lavora sfruttando degli OP_CODE che lavorano su uno stack.

- **DUP o OP_DUP** : prende il primo elemento dello stack e lo duplica;
- **2DUP o OP_2DUP** : prende i primi due elementi sullo stack e li duplica;
- **EQUAL o OP_EQUAL** : Controlla se due elementi sullo stack e li sostituisce con un nuovo valore: se sono uguali ritorna 0x0 se è falso 0x1 se è vero, in caso che sia l'ultimo operazione true sblocca la transazione e false la blocca;
- **NOT o OP_NOT** : Inverte un valore booleano presente sul top dello stack, se il valore non è booleano da errore;
- **VERIFY o OP_VERIFY** : controlla sul top dello stack se si trova il valore true (0x1) se non è presente invalida la transazione;
- **SHA256 o OP_SHA256** : Prende il primo elemento dello stack e lo sostituisce con il suo valore HASH tramite l'algoritmo SHA256;
- **SHA1 o OP_SHA1** : Prende il primo elemento dello stack e lo sostituisce con il suo valore HASH tramite l'algoritmo SHA1;
- **SWAP o OP_SWAP** : Scambia i primi due elementi sullo stack; Esempio se abbiamo 1 e due sullo stack, l'operazione 1 2 SWAP ci darà 2 1;
- **EQUALVERIFY o OP_EQUALVERIFY** : Prende i primi due elementi sullo stack e ritorna un valore booleano: true (0x1) false(0x0);
- **CHECKSIG o OP_CHECKSIG**: controlla la firma della transazione se è valida conferma la transazione altrimenti la blocca;
- **HASH160 o OP_HASH160**: Calcola il valore hash160 del primo elemento presente sullo stack. Viene effettuato un doppio hash prima SHA-256 e poi RIPEMD-160;
- **HASH256 o OP_HASH256**: Calcola il valore hash256 del primo elemento presente sullo stack. Viene effettuato un doppio SHA256.
- **Operazione matematiche** : ADD, SUB, MUL, DIV effettuano la rispettiva operazione matematica fra i primi due elementi dello stack, li consuma e inserisce il risultato sullo stack;
- **Booleani**: sono indicati rispettivamente con OP_TRUE o OP_FALSE;
- **NEGATE o OP_NEGATE**: Nega il primo elemento dello stack, 2 diventa -2, -2 diventa 2.

Per provare gli script andare al seguente link Bitcoin IDE

Esercizio 1

```
1 2 OP_2DUP OP_EQUAL OP_NOT OP_VERIFY OP_SHA256 OP_SWAP OP_SHA256 OP_EQUAL OP_NOT
```

Lo script sopra riportato svolge le seguenti operazioni:

- **1**: effettua un operazione di push sullo stack per inserire 1;
- **2**: effettua un operazione di push sullo stack per inserire 2;
- **OP_2DUP** : prende i primi due elementi sullo stack e li duplica in questo caso duplicherà 1 e 2;
- **OP_EQUAL** : Verifica se i primi due elementi presenti sullo stack sono uguali in questo caso 1 e 2, e inserisce sullo stack un valore che rappresenta falso (0x0);
- **OP_NOT** :prende il primo valore sullo stack (0x1) essendo booleano lo inverte, quindi passa da falso a true (0x1);
- **OP_VERIFY** : controlla sul top dello stack se si trova il valore true (0x1) se non è presente invalida la transazione;
- **OP_SHA256** : Prende il primo elemento dello stack (2) e lo sostituisce con il suo valore HASH tramite l'algoritmo SHA256;
- **OP_SWAP** : Scambia i primi due elementi sullo stack (sha256 di 2 e il valore 1);
- **OP_SHA256** : Prende il primo elemento dello stack (1) e lo sostituisce con il suo valore HASH tramite l'algoritmo SHA256;
- **OP_EQUAL** : Verifica se i due valori hash presenti sullo stack sono uguali, in caso affermativo sblocca la transazione altriementi la blocca;

Quindi questo script in caso 1 e 2 avessero valore hash sha256 uguale sbloccherebbe la transazione. (1 e 2 si possono sostituire in modo generale con A e B). Con valori generali avremo:
Lo script ha lo scopo di sbloccare la transazione se fornisce due valori diversi che hanno stesso valore hash tramite l'algoritmo sha256.

Esercizio 2

```
2 OP_DUP OP_HASH160 030353B92873B9A480DB2BB0E606009A5533FC6D OP_EQUALVERIFY OP_CHECKSIG
```

- **2**: effettua il push del valore 2 sullo stack;
- **OP_DUP** : prende il primo elemento dello stack e lo duplica, avremo quindi 2 2;
- **OP_HASH160** : Effettua il valore hash160 del primo elemento sullo stack; QUindi abbiamo 2 hash160(2);
- **030353B92873B9A480DB2BB0E606009A5533FC6D** : è un valore hash160 che viene pushato sullo stack;
- **OP_EQUALVERIFY** : Controlla se il valore hash del numero 2 e quello inserito da noi sono uguali, in caso affermativo procede ad effettuare la prossima istruzione; sullo stack abbiamo 2.
- **OP_CHECKSIG**: controlla la firma della transazione se è valida conferma la transazione altrimenti la blocca;

Lo script quindi controlla se il valore hash160(2) è uguale a 030353B92873B9A480DB2BB0E606009A5533FC6D, se si verifica la firma della transazione e se è valida la conferma.

Esercizio 3

```
2 3 OP_ADD 6 OP_EQUAL
```

- **2** : inserisce sullo stack 2 tramite una push; stack : 2;
- **3** : inserisce sullo stack 3 tramite una push; stack : 2 3
- **OP_ADD**: effettua la somma tra i primi due elementi dello stack e li consuma, inserisci il risultato sullo stack; stack : 5;
- **6** : inserisce sullo stack 6 tramite una push; stack : 5 6;
- **OP_EQUAL**: Controlla se i primi due elementi dello stack sono uguale e in caso affermativo restituisce true e sblocca il contratto. Nel nostro caso restituira false perchè 5 != 6.

Lo script effettua la somma fra 2 e 3 e controlla che il risultato sia uguale a 6. Essendo che che 2 + 3 fa 5 lo script non sarà mai sbloccato.

Esercizio 3

```
2DUP EQUAL NOT VERIFY SHA1 SWAP SHA1 EQUAL
```

- **2DUP**: Duplica i primi due elementi presenti sullo stack, per esempio se sullo stack abbiamo due stringhe A e B avremo dopo la chiamata a 2DUP: A B A B;
- **EQUAL**: Prende due elementi sullo stack consumandoli e verifica se sono uguali, inserisce sullo stack il risultato dell'uguaglianza (0 falso, 1 vero), supponendo che le due stringhe siano A e B lo stack avrà i seguenti valori: A B 0;
- **NOT**: Inverte un valore booleano: vero diventa falso e viceversa. Lo stack ora sarà: A B 1;
- **VERIFY**: Verifica che un il valore booleano (consumandolo) presente sul top dello stack sia vero, in caso contrario blocca la transazione. Lo stack ora sarà: A B;
- **SHA1**: Consuma il primo elemento presente sullo stack ed effettua il push del suo valore SHA1. Lo stack sarà: A SHA1(B);
- **SWAP**: Effettua lo scambio tra i primi due elementi delle stack. Lo stack ora ha come valore: SHA1(B) A;
- **SHA1**: Consuma il primo elemento presente sullo stack ed effettua il push del suo valore SHA1, in questo caso A. Lo stack sarà: SHA1(B) SHA1(A);
- **EQUAL**: Controlla se i due elementi sul top dello stack sono uguali.

Lo script controlla se due elementi presenti nello stack che chiameremo A e B sono diversi, e in caso affermativo procede con il verificare se che i valori SHA1 di A e B siano uguali e in caso affermativo blocca la transazione. Quindi verifica se due valori diversi hanno lo stesso valore SHA1

Capitolo 2

Go lang

2.1 Sintassi di GO LANG

- **package** <**nome**>: indica a quale package appartiene il programma;
- **import** <**nome**>: indica quale libreria importare;
- **libreria fmt**: è una libreria standard per input/output;
- **libreria time**: libreria per la gestione del tempo;
- **libreria sync**: libreria per le primitive della gestione della sincronia;
- **func**: serve per definire una nuova funzione. La sintassi di una funzione è la seguente:
  func nome_funzione(params) tipo_funzione { ... };
- **Costanti**: Le dichiarazioni di costanti possono utilizzare il contatore iota, che inizia da 0 in ogni blocco const e aumenta in automatico:

```
const (
    Monday  = iota // 0
    Tuesday = iota // 1
)
fmt.Println(Tuesday)
```

- **assegnazione tipo**: I tipi principali di Go sono: int, float, string, bool, ecc. Per dire che una variabile a un determinato tipo si inserisce quest'ultimo dopo il suo nome, es: var c **int**, var l **string**.
- **Dichiarazione variabili**: in go ci sono due modi per dichiarare le variabili:
  - var c int : dichiariamo una variabile c di tipo int;
  - m **:=** 4 : dichiariamo una variabile m a cui assegnamo 4, in questo caso il tipo di m viene inferito dal valore assegnato.
- **Array** : Abbiamo due modi per dichiarare un array:
  - var array_name = [length]tipovalues
  - var array_name = [...]tipovalues : la lunghezza è inferita

  Esistono anche le slice, simili agli array ma hanno dimensione variabile:
  - array_name = []tipovalues : si può usare il metodo len() per verificare quanti elementi ci sono al suo interno.
- **defer**: esegue una funzione (o metodo) quando il contesto o funzione iche la contiene effettua un return. Gli argomenti sono valutati al momento dell'esecuzione di defer, non della sua definizione. La defer ad ogni esecuzione mette la chiamata a funzione o metodo in una coda, cha saranno successivamente eseguite in ordine LIFO;
- **WaitGroups**: viene utilizzato per attendere il completamento di più goroutine, attende finchè il suo contatore non arriva a 0. Per capire quante goroutine si devono attendere si utilizza wg.add(#digoroutine). Per decrementarlo al termine di ogni goroutine si utilizza wg.Done();

- **Map**: var a = map[KeyType]ValueTypekey1:value1, key2:value2,...
- **Canali**: un canale è un meccanismo di comunicazione tra goroutine che consente di scambiare tra di loro dei dati in modo sincrono (Simili alle pipeline). Una goroutine che invia un valore su un canale bloccherà l'esecuzione fino a quando un'altra goroutine non lo riceve, creando così una sincronizzazione tra le due goroutine.
  - ch := make(chan int) : dichiariamo un canale;
  - ch := make(chan int, buf_dim) : dichiariamo un canale, il parametro opzionale buf_dim indica la dimensione del buffer, se è 1 possiamo scrivere un solo valore alla volta, 2 possiamo inserire due valori alla volta. Quando leggiamo un valore dal buffer questi vengono letti in modo sequenziale, se abbiamo ch <- 1; ch <- 2; alla lettura leggeremo sempre prima 1 e poi 2. Se la dimensione del buffer è 1 e non abbiamo letto il valore non possiamo inserirne un'altro.
  - ch <- v : invia v al canale ch;
  - v := <-ch : ricevo dal canale h ed assegno a v;
  - in una funzione un canale può essere di tre tipi:
    - func n(ch chan<- string) : ch è in solo scrittura;
    - func n(ch <-chan string) : ch è in solo lettura;
    - func n(ch chan string) : ch è in lettura e scrittura;
- In Go, una goroutine è una funzione che viene eseguita in parallelo con altre funzioni all'interno del programma. Una goroutine è molto più leggera di un thread vero e proprio, poiché viene eseguita all'interno del medesimo indirizzo spaziale del programma che la ha creata, il che significa che condivide tutte le risorse del programma padre, come la memoria. Questo rende molto più semplice e conveniente creare molte goroutine all'interno del programma, rispetto a creare molte thread che devono gestire risorse separate. Una gorutine si indica come segue: **go** funzione; Una funzione passata ad una goroutine può anche essere anonima:

```
..
go func() {
    ...
}()
...
```

- **select**: Select è una struttura di controllo in Go analoga a un'istruzione switch di comunicazione. Ogni case deve essere una comunicazione, inviata o ricevuta. Select esegue un case eseguibile a caso. Se nessun case è eseguibile, si blocca finché uno non lo è. Una clausola default è sempre eseguibile.

```
c := make(chan int)
go func() {
    for {
        fmt.Println(<-c)
    }
}()
for {
    select {
        case c <- 0: // no statement, no fall through
        case c <- 1:
    }
}
```

Esercizio 1

```
package main
import "fmt"

func ping(pings chan<- string, msg string) {
        fmt.Println("--- Inserisco msg in pings ---")
        pings <- msg
        fmt.Println("--- msg inserito ---")
}

func pong(pings <-chan string, pongs chan<- string) {
        //msg := <-pings
        // pongs <- msg
        fmt.Println("--- Inserisco il contenuto di pings in pongs ---")
        pongs <- <-pings
}

func main() {
        pings := make(chan string, 1)
        pongs := make(chan string, 1)
        ping(pings, "Messaggio passato")
        pong(pings, pongs)
        fmt.Println("---", <-pongs, "---")
}
```

Il seguente codice go sopra riportato appartiene al package "main", ed importa la libreria fmt che si occupa di gestire metodi per l'input e l'output. Vengono definite due funzioni:

- ping(): non ha tipo di ritorno e prende due parametri come input, il primo è un canale di tipo stringa in solo scrittura e il secondo una variabile di tipo stringa. La funzione non fa altro che copiare il contenuto di msg nel canale pings.
- pong(): non ha tipo di ritorno e prende due parametri come input: il primo è un canale in solo lettura di tipo stringa, il secondo è un'altro canale di tipo stringa aperto in modalità solo scrittura. La funzione si occupa di copiare il contenuto del canale pings in pongs.

All'interno del main vengono inizializzati due canali di tipo stringa tramite la funzione make: pings e pongs, hanno dimensione del buffer 1 quindi possiamo scrivere soltanto un valore alla volta. Vengono poi successivamente invocate le funzioni ping e pong. E infine viene stampato il contenuto del canale pongs. Il programma ha lo scopo di passare la stringa "Messaggio passato" dal canale ping al canale pongs.

Esercizio 2

```
package main

import (
        "fmt"
        "time"
)

func worker(done chan bool) {
        fmt.Println("--- Working... ---")
        time.Sleep(time.Second)
        fmt.Println("--- done ---")
        done <- true
}

func main() {
        done := make(chan bool, 1)
        go worker(done)
        <-done
}
```

Il codice go sopra riportato appartiene al package main ed importa due librerie, fmt per la gestione dell'input/output e time utilizzata per la gestione del tempo. Oltre al main viene definita un'altra funzione chiama worker. La funzione worker prende come parametro un canale booleano aperto sia in scrittura che lettura. La funzione inizialmente stampa a video "Working", poi si ferma per un secondo, stampa a video "done" ed assegna al canale done il valore true.
Il main inzializza un canale booleano tramite la funzione make, questo canale ha dimensione del buffer 1, vuol dire che può essere scritto un solo valore alla volta. Richiama poi tramite la parola chiave **go** una goroutine e chiama la funzione worker passando il canale "done" come parametro. L'istruzione " <- done" fa si che il padre attenda che il figlio creato tramite la goroutine termini la sua esecuzione. Fatto questo termina anche lui.

Esercizio 3

```
package main

import (
        "fmt"
        "time"
)

func main() {
        c1 := make(chan string)
        c2 := make(chan string)
        go func() {
                time.Sleep(1 * time.Second)
                c1 <- "one"
        }()
        go func() {
                time.Sleep(2 * time.Second)
                c2 <- "two"
        }()

        fmt.Println("received", <-c2)
        fmt.Println("received", <-c1)
}
```

Il codice go sopra riportato appartiene al package "main", ed importa due librerie "fmt" per la gestione di input/output, "time" per la gestione del tempo.
All'interno del main vengono inizializzati due canali di tipo string. Sono poi create due goroutine tramite l'uso di funzioni anonime. La prima goroutine attende per un secondo e scrive nel canale c1 la strina "one". La seconda goroutine anch'essa ha una funzione anonima, aspetta per due secondi ed inserisce nel canale c2 la strigna "two". Viene stampato al video dopo due secondi "two" e subito dopo "one". Anche se il canale 1 era pronto dopo 1 secondo si è dovuti attendere la stampa della seconda goroutine che aspettava per 2 secondi.

Esercizio 4

```
package main

import (
        "fmt"
        "sync"
)

func main() {
        slice := []string{"a", "b", "c", "d", "e"}
        sliceLength := len(slice)

        var wg sync.WaitGroup
        wg.Add(sliceLength)

        fmt.Println("Running for loop...")

        for i := 0; i < sliceLength; i++ {
                go func(i int) {
                        defer wg.Done()
                        val := slice[i]
                        fmt.Println("i: %v, val :%v\n", i, val)
                }(i)
        }
        fmt.Println("Doing other stuff")
        wg.Wait()

        fmt.Println("Finishing for loop")

}
```

Il seguente codice go appartiene al package main e importa come librerie fmt e sync. La libreria fmt è utilizzata per gestire le operazioni di input/output, la libreria sync è utilizzata per usare le primetive della gestione della sincronia.
Il main definisce un array di stringhe contenente le prime 5 lettere dell'alfabeto, successivamente crea un WaitGroup settando il contatore uguale alla lunghezza dell'array definito precedentemente. Successivamente viene eseguito un ciclo for fincé la variabile i non è minore di 5 (dimensione dell'array). Ad ogni iterazione viene lanciata una nuova goroutine che richiama una funzione anonima passando come parametro il valore della variabile i. Ogni goroutine esegue le seguenti istruzioni:

- prima di tutto con una defer si fa in modo che il decremento del contatore avvenga sempre alla fine della funzione anonima;
- poi si assegna alla variabile val l'i-esima posizione dell'array;
- infine stampa a video l'iterazione corrente e il valore preso dall'array es:
  i: 1, val :b;

Nel frattempo viene eseguita una prima stampa da parte del main "Doing other stuff". Poi Aspetta finché non vengono eseguite tutte le goroutine ed infine stampa "Finishing for loop".

2.2 Chaincode

Per implementare un chaincode in go in java abbiamo bisogno di diverse cose:

- **libreria github.com/hyperledger/fabric/core/chaincode/shim** fornisce un insieme di funzioni e strutture per interagire con il ledger e con il sistema di consenso. Ad esempio, fornisce una struct ChaincodeStubInterface che rappresenta un'interfaccia tra il Chaincode e il peer di Hyperledger Fabric, e fornisce funzioni per la lettura e la scrittura sul ledger;
- **libreria github.com/hyperledger/fabric/protos/peer** contiene i protocolli di rete e i messaggi utilizzati per comunicare tra i componenti di Hyperledger Fabric. Ad esempio, contiene il pacchetto pb che contiene la definizione dei messaggi di peer protobuf utilizzati per l'invocazione del Chaincode e la risposta all'invocazione;
- **struct**: type SimpleChaincode struct {}: definiamo una struttura su cui andremo poi a implementare i metodi dell'interfaccia chaincode;
- **pb.Response** è un tipo definito nel pacchetto peer di Hyperledger Fabric. È un alias per peer.Response, che rappresenta una risposta protobuf inviata dal Chaincode al peer di Hyperledger Fabric. Questa risposta contiene informazioni sullo stato dell'invocazione del Chaincode, ad esempio se è stato eseguito con successo o se è stato riscontrato un errore, e può anche contenere dati restituiti dal Chaincode.
- I metodi dell'interffaccia chaincode sono i seguenti:
  - **Init**: Questo metodo viene chiamato solo una volta durante il processo di deploy del Chaincode e viene utilizzato per inizializzare le variabili e i valori iniziali del ledger.

    ```
    func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
        // Inizializzazione delle variabili e dei valori iniziali del ledger
        return shim.Success(nil)
    }
    ```

  - **Invoke**: Questo metodo viene chiamato dalle applicazioni per eseguire transazioni sul ledger. Può leggere e scrivere sul ledger e viene utilizzato per effettuare operazioni come l'aggiornamento di una registrazione o la creazione di una nuova.

    ```
    func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
      // Implementazione delle funzioni di transazione
      return shim.Success(nil)
    }
    ```

    La funzione implementa le funzioni di transazione, ovvero le operazioni effettuate sul ledger, come la creazione, la lettura, l'aggiornamento e la cancellazione di oggetti sul ledger.
  - **Query**: Questo metodo viene chiamato dalle applicazioni per leggere i dati dal ledger. Non può modificare i valori sul ledger e viene utilizzato per recuperare informazioni specifiche.

    ```
    func (t *SimpleChaincode) Query(stub shim.ChaincodeStubInterface) pb.Response {
        // Implementazione delle funzioni di query
        return shim.Success(nil)
    }
    ```

    La funzione implementa le funzioni di query, ovvero le operazioni effettuate sul ledger per recuperare informazioni, come la lettura di un oggetto sul ledger.

  Tutte e tre le funzioni prendono come input un oggetto stub di tipo shim.ChaincodeStubInterface, che rappresenta un'interfaccia tra il Chaincode e il peer di Hyperledger Fabric. La funzione utilizza questo oggetto per accedere a funzionalità come la lettura e la scrittura sul ledger.
  Tutte e tre le funzioni restituiscono shim.Success con un argomento nil, che indica che l'inizializzazione, la transazione o la query è stata eseguita con successo. In caso di errore, si può utilizzare la funzione shim.Error con un messaggio di errore come argomento.

Esercizio 1

```
package main

import (
        "fmt"
        "github.com/hyperledger/fabric/core/chaincode/shim"
        "github.com/hyperledger/fabric/protos/peer"
)

type HeroesServiceChaincode struct{}

func (t *HeroesServiceChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
        function,_ := stub.GetFunctionAndParameters()
        if function != "init" {
                return shim.Error("Unknown function call")
        }
        err := stub.PutState("hello", []byte("world"))
        if err != nil {
                return shim.Error(err.Error())
        }
        return shim.Success(nil)
}

func (t *HeroesServiceChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
        fuction, args := stub.GetFunctionAndParameters()
        var err error
        if function != "Invoke"{
                reutn shim.Error("Unknown function call")
        }
        if len(args) < 1 {
                retunt shim.Error("The number of arguments is insufficient")
        }
        if args[0] == "query" {
                return t.query(stub, args)
        }
        if args[0] == "invoke" {
                return t.invoke(stub, args)
        }
        return shim.Error("Unknown action, check the first argument")
}

func (t *HeroesServiceChaincode) query(stub shim.ChaincodeStubInterface, args []string) pb.Response {
        fmtPrintln("### HeroesServiceChaincode query ###")
        if len(args) < 2 {
                return shim.Error("The numer of arguments is insufficient")
        }
        if args[1] = "hello" {
                state, err := stub.GetState("hello")
                if err != nil {
                        return shim.Error("Failed to get state of hello")
                }
        }
        return shim.Error("Unknown query action, check the second argument")
}

func (t *HeroesServiceChaincode) invoke(stub shim.ChaincodeStubInterface, args []string) pb.Response {
        fmtPrintln("### HeroesServiceChaincode query ###")
        if len(args) < 2 {
                return shim.Error("The numer of arguments is insufficient")
        }
        if args[1] = "hello" && len(args) )) 3 {
                state, err := stub.PutState("hello", []byte(args[2]))
                if err != nil {
                        return shim.Error("Failed to update state of hello")
                }
                err = stub.SetEvent("eventInvoke",[]byte{})
                if err != nil{
                        return shim.Error(err.Error())
                }
        }
        return shim.Error("Unknown invoke action, check the second argument")
}
```

```
func main() {
        err := shim.Start(new(HeroesServiceChaincode))
        if err != nil {
                fmt.Println("Error starting Heroes Service chaincode: %s",err)
        }
}
```

La keyword package indica dove si trova il chaincode. Con la keyword import indichiamo le librerie da importare. Viene poi definita la struttura del CHaincode, chiamato in questo caso HeroesServiceChaincode.
Vengono poi implementate le funzioni essenziali di un chaincode ovvero:

- **Init**: permette di deployare il chaincode. Tramite lo stub preleviamo il nome della funzionalità richiesta. Se la funzione richiesta è diversa da "Init" allora viene restituito un messaggio di errore, altrimenti inseriamo l'asset "world" identificato dalla chiave "hello". Viene controllato l'avvenuto inserimento dell'asset.
- **Invoke**: viene invocata ad ogni esecuzione del chaincode una volta deployato. Tramite lo stub otteniamo i parametri e la funzionalità richiesta. Se la funzione richiesta non è "Invoke" allora restituiamo un messaggio di errore. Successivamente controlliamo il numero dei parametri. In base al parametro passato verrà invocata una di queste funzioni:
  - **query**: controlliamo il numero di parametri. Se il secondo parametro (args[1]) è la stringa "hello" allora viene prelevato l'asset identificato da quest'ultima. Vengono inoltre effettuati controlli sul prelievo e la presenza dell'asset.
  - **invoke**: controlliamo il numero di parametri. Se i parametri sono esattamente tre ed il secondo parametro è la stringa "hello" allora viene inserito l'asset args[2] identificato dalla stringa "hello". Viene controllato l'effettivo inserimento. Viene inoltre settato un evento chiamato "eventInvoke".

  Se la funzione richiesta non corrisponde a nessuna di queste due la funzione Invoke restituisce un messaggio di errore.

La funzione main crea un nuovo HeroesServiceChaincode e restituisce un errore se non lo si riesce ad avviare.

Esercizio 2

```
package main

import (
        "fmt"
        "github.com/hyperledger/fabric/core/chaincode/shim"
        "github.com/hyperledger/fabric/protos/peer"
)

type SampleChaincode struct{}

func (t *SampleChaincode) Init(stub shim.ChaincodeStubInterface) peer.Response {
        args := stub.GetStringArgs()
        if len(args) != 2 {
                return shim.Error("Incorrect arguments. Expecting a key and a value")
        }
        err := stub.PutState(args[0], []byte(args[1]))
        if err != nil {
                return shim.Error(fmt.Sprintf("Failed to create asset: %s", args[0]))
        }
        return shim.Success(nil)
}

func (t *SampleChaincode) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
        fn, args := stub.GetFunctionAndParameters()
        var result string
        var err error
        if fn == "set" {
                result, err = set(stub, args)
        } else {
                result, err = get(stub, args)
        }
        if err != nil {
                return shim.Error(err.Error())
        }
        return shim.Success([]byte(result))
}

func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
        if len(args) != 2 {
                return "", fmt.Errorf("Incorrect arguments. Expecting a key and a value")
        }
        err := stub.PutState(args[0], []byte(args[1]))
        if err != nil {
                return "", fmt.Errorf("Failed to set asset: %s", args[0])
        }
        return args[1], nil
}

func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {
        if len(args) != 1 {
                return "", fmt.Errorf("Incorrect arguments. Expecting a key")
        }
        value, err := stub.GetState(args[0])
        if err != nil {
                return "", fmt.Errorf("Failed to get asset: %s with error: %s", args[0], err)
        }
        if value == nil {
                return "", fmt.Errorf("Asset not found: %s", args[0])
        }
        return string(value), nil
}

func main() {
        err := shim.Start(new(SampleChaincode))
        if err != nil {
                fmt.Println("Could not start SampleChaincode")
        } else {
                fmt.Println("SampleChaincode successfully started")
        }
}
```

La parola chiave package indica dove si trova il chaincode. Invece con la parola chiave import indichiamo le librerie da importare. Viene poi definita la struttura del Chaincode, chiamato in questo caso SampleChaincode. Sono quindi implementate le funzioni essenziali per il chaincode:

- **Iinit**: è la funzione che permette di deployare il chaincode. Tramite lo stub otteniamo gli argomenti forniti al momento dell'invocazione del Chaincode. Se il numero di argomenti è diverso da due viene lanciato un errore. Altrimenti va avanti e procede ad inserire un nuovo asset con chiave args[0] e valore args[1], viene anche controllato se l'asset viene effettivamente creato.

- **Invoke**: viene invocata ad ogni esecuzione del chaincode una volta deployato. Tramite lo stab otteniamo sia il nome della funzionalità richiesta che gli argomenti passati al momento dell'invocazione. Se la funzionalità richiesta non è né "set" viene sempre chiamata "get". Le due funzioni vengono invocate passando come valori lo stub e gli argomenti. Le due funzioni svolgono il seguente compito:

  – **set**: controlla se il numero di argomenti è diverso da due, se è diverso da 2 si ritorna una stringa vuota e un errore. Se è uguale a due procede ad inserire l'asset avente come chiave args[0] e valore args[1], viene anche controllato se l'asset viene effettivamente creato. Vengono ritornati due valori:

    * in caso di successo i valori sono: args[1] per segnale che quale valore è stato inserito e nil, per dire segnale che non c'è stata presenza di errore;
    * in caso di errore i valori sono: "" per segnale che nulla è stato aggiunto e viene ritornato un errore;

  – **get**: controlla se il nuemro di argomenti è diverso da 1, se è diverso da 1 ritorna una stringa vuota e un errore. Successivamente procede a prelevare l'asset avente chiave args[0]. Vengono effettuati due controlli:

    * si verifica un qualsiasi tipo di errore: viene ritornata una stringa vuota, e un messaggio contenente l'informazione che si è verificato un errore nell'ottenere l'asset desiderato;
    * il valore dell'asset è nil: viene ritornata una stringa vuota e l'errore con messaggio asset not found;

    In caso di successo viene ritornato il valore dell'asset, e nil per segnalare che non si è verificato nessun errore.

  Dopo le invocazioni di uno dei due metodi viene controllato se la variabile err è diverso da nil, se lo è si procede a restituire un errore. Altrimenti si restituisce con successo la stringa restituita da uno dei due metodi.

La funzione main invece crea un nuovo SampleChaincode e restituisce un messaggio di errore se non lo si è riusciti ad avviare, o in caso contrario un messaggio di successo.

2.3 Vulnerabilità Chaincode

Non supporto della logica Read-Your-Write

```
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.Succes(nil)
}
```

Il problema è 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 riflessi.

Variabile globale in una struttura

```
type BadChaincode struct {
    globalValue string;
}

func  (t *BadChaincode) Invoke(stub shim.ChaincodeStubInterface) peer.response {
    t.globalValue = args[0]
    return shim.Success([]byte("success"))
}
```

Non si dovrebbe inserire una variabile globale all'interno di una struttura questo perché non è detto che ogni peer esegue ogni transizione, e quindi il valore della variabile sarà sicuramente diverso per ogni peer.