Parte 1: Dalla Virtualizzazione alla Containerizzazione

Questa prima parte copre i concetti fondamentali della virtualizzazione, la distinzione tra macchine virtuali e container, e un'analisi approfondita di Docker.

1. Fondamenti di Virtualizzazione

La virtualizzazione è una tecnologia che permette di creare una versione virtuale (anziché fisica) di una risorsa informatica, come un server, un sistema di archiviazione, una rete o un sistema operativo. Il componente software chiave che rende possibile la virtualizzazione è l'**Hypervisor**.

1.1. L'Hypervisor (o Virtual Machine Monitor - VMM)

L'hypervisor è uno strato software che si interpone tra l'hardware fisico (host) e le macchine virtuali (guest). Il suo compito è astrarre e gestire le risorse hardware (CPU, RAM, storage, rete) e distribuirle tra le varie VM in esecuzione.

Esistono due tipi principali di hypervisor:

- **Hypervisor di Tipo 1 (Bare-metal):** Viene installato direttamente sull'hardware fisico dell'host, come se fosse un sistema operativo. Offre le massime prestazioni e sicurezza, poiché non c'è un SO host sottostante che consuma risorse o introduce vulnerabilità. È la scelta prediletta in ambienti data center e cloud.
  - *Esempi:* VMware vSphere/ESXi, Microsoft Hyper-V, Xen, KVM (Kernel-based Virtual Machine, integrato nel kernel Linux).
- **Hypervisor di Tipo 2 (Hosted):** Viene installato come un'applicazione su un sistema operativo host preesistente (es. Windows, macOS, Linux). È più semplice da installare e gestire, ideale per ambienti di sviluppo, testing e uso desktop, ma introduce un overhead di prestazioni a causa del doppio strato di sistemi operativi (host + guest).
  - *Esempi:* VMware Workstation, Oracle VirtualBox, Parallels Desktop.

1.2. Il Problema delle Istruzioni Privilegiate e le Soluzioni

Un sistema operativo è progettato per avere il controllo diretto dell'hardware, eseguendo istruzioni privilegiate (quelle che interagiscono con la CPU, la memoria, i dispositivi I/O). In un ambiente virtualizzato, il sistema operativo *guest* non può avere questo accesso diretto, altrimenti potrebbe interferire con l'host o con altre VM. Questo problema viene risolto principalmente in due modi:

1. **Full Virtualization con Binary Translation (Approccio Classico):**

- **Come funziona:** L'hypervisor intercetta *tutte* le istruzioni eseguite dal kernel del sistema operativo guest. Le istruzioni non privilegiate vengono eseguite direttamente sulla CPU host per massima velocità. Le istruzioni privilegiate, invece, vengono "intrappolate" e tradotte al volo dall'hypervisor in una sequenza di istruzioni sicure (non privilegiate) che producono lo stesso risultato, ma senza violare l'isolamento.
- **Vantaggi:** Non richiede alcuna modifica al sistema operativo guest. Qualsiasi SO può essere virtualizzato.
- **Svantaggi:** È un processo molto costoso in termini di performance. La continua intercettazione e traduzione introduce un notevole overhead.

2. **Virtualizzazione Hardware-Assisted (Approccio Moderno):**
    - **Come funziona:** I processori moderni (Intel VT-x, AMD-V) introducono nuove modalità di esecuzione. La CPU stessa è consapevole della virtualizzazione. Viene creato un nuovo livello di privilegio, spesso chiamato **"Ring -1"** o "root mode", ancora più privilegiato del "Ring 0" del kernel. L'hypervisor viene eseguito in questo Ring -1, mentre il sistema operativo guest può eseguire le sue istruzioni privilegiate direttamente nel Ring 0, come farebbe su una macchina fisica. La CPU, però, interviene automaticamente per impedire che queste istruzioni vadano a intaccare le risorse dell'host o di altre VM, passando il controllo all'hypervisor solo quando strettamente necessario.
    - **Vantaggi:** Prestazioni quasi native, poiché la traduzione software viene quasi del tutto eliminata. Maggiore sicurezza e stabilità. Permette la "nested virtualization" (VM dentro altre VM).
    - **Svantaggi:** Richiede supporto hardware specifico nel processore.

Un terzo approccio è la **Paravirtualizzazione**, in cui il kernel del sistema operativo guest viene modificato specificamente per essere "consapevole" di girare su un hypervisor. Invece di eseguire istruzioni privilegiate, effettua delle chiamate dirette all'hypervisor (hypercalls). Questo riduce l'overhead, ma richiede un SO guest modificato. Oggi, con la diffusione della virtualizzazione hardware-assisted, questo approccio è meno comune.

2. Dalle Macchine Virtuali ai Container: Un Cambio di Paradigma

Mentre le VM virtualizzano l'hardware, i container virtualizzano il sistema operativo. Questa è la distinzione fondamentale.

Caratteristica	Macchina Virtuale (VM)	Container
Livello di Astrazione	**Hardware**	**Sistema Operativo**
Componenti	Ogni VM ha un intero SO guest (kernel + librerie + binari)	I container condividono il **kernel del SO host**. Contengono solo l'applicazione e le sue dipendenze (librerie, binari).
Isolamento	Forte isolamento a livello hardware. Una VM non può "vedere" le altre.	Isolamento a livello di processo. Usano **namespaces** (per isolare processi, rete, utenti) e **cgroups** (per limitare le risorse CPU/RAM) del kernel Linux.
Dimensione	Grande (diversi Gigabyte).	Leggero (pochi Megabyte o decine di MB).
Tempo di Avvio	Lento (minuti), come avviare un computer.	Veloce (secondi o millisecondi), come avviare un processo.
Overhead	Alto (ogni VM consuma RAM e CPU per il suo SO).	Basso (le risorse sono condivise e gestite efficientemente).
Portabilità	Portabile, ma l'immagine della VM è molto pesante.	Estremamente portabile. L'immagine è piccola e autoconsistente.

In sintesi: Usa le VM quando hai bisogno di un forte isolamento o devi eseguire sistemi operativi completamente diversi (es. un server Linux su un host Windows). Usa i container per impacchettare e distribuire applicazioni in modo efficiente, leggero e scalabile, specialmente in architetture a microservizi.

3. Introduzione alla Containerizzazione con Docker

Docker è la piattaforma leader per lo sviluppo, la distribuzione e l'esecuzione di applicazioni all'interno di container.

3.1. Architettura di Docker

- **Docker Daemon (`dockerd`):** È il processo in background che gestisce tutto: costruisce, esegue e distribuisce i container. Ascolta le richieste dall'API di Docker.

- **Docker Client (`docker`):** È l'interfaccia a riga di comando (CLI) che l'utente utilizza per interagire con il Docker Daemon. Quando si digita `docker run ...` , il client invia il comando al daemon.
- **Docker Registry:** È un sistema di archiviazione per le immagini Docker.
  - **Registry Remoto:** Un servizio pubblico o privato per archiviare e condividere immagini. **Docker Hub** è il registry pubblico ufficiale e predefinito.
  - **Registry Locale:** Una cache locale sulla tua macchina dove vengono salvate le immagini scaricate (`docker pull`) o costruite (`docker build`).

3.2. I Componenti Fondamentali: Immagini e Container

- **Immagine (Image):** È un template *read-only* (sola lettura) che contiene un insieme di istruzioni per creare un container. Un'immagine include il codice dell'applicazione, un runtime, le librerie, le variabili d'ambiente e i file di configurazione. Le immagini sono costruite a partire da un **Dockerfile** e sono organizzate in *layer* (strati) sovrapposti. Questo rende la loro costruzione e distribuzione molto efficiente: se modifichi solo l'ultimo strato, devi scaricare/caricare solo quello.
- **Container:** È un'istanza eseguibile di un'immagine. Mentre l'immagine è statica, il container è un processo vivo e isolato. Quando si crea un container, Docker aggiunge uno strato *writable* (scrivibile) sopra gli strati dell'immagine. Tutte le modifiche fatte durante l'esecuzione del container (creazione/modifica di file) vengono scritte in questo strato.

3.3. Comandi Essenziali e Ciclo di Vita

- `docker build` : Costruisce un'immagine a partire da un Dockerfile.
  - `docker build -t my-app:1.0 .` -> Costruisce l'immagine, le assegna il tag (`-t`) `my-app:1.0` usando il `Dockerfile` nella directory corrente (`.`).
- `docker run` : Crea e avvia un container da un'immagine. Se l'immagine non è presente localmente, la scarica automaticamente da Docker Hub.
  - `docker run -d -p 8080:80 --name webserver nginx` -> Esegue un container in background (`-d`), mappa (`-p`) la porta 8080 dell'host sulla porta 80 del container, gli assegna un nome (`--name`) e usa l'immagine `nginx` .
- `docker stop` : Ferma un container in esecuzione (invia un segnale `SIGTERM` , poi `SIGKILL`).
- `docker start` : Riavvia un container fermato.
- `docker rm` : Rimuove un container fermato. Lo strato scrivibile viene eliminato.
- `docker rmi` : Rimuove un'immagine.
- `docker commit` : **Salva lo stato di un container come una nuova immagine.**
  - *Risposta alla domanda:* Sì, dopo aver stoppato un container, puoi salvare le sue modifiche in una nuova immagine. Se hai avviato un container, hai installato `vim` al suo interno e poi lo hai fermato, puoi usare `docker commit <container_id> nuova-immagine:con-vim` . Questo "congela" lo stato dello strato scrivibile e lo trasforma in un nuovo strato read-only per la nuova

immagine. **Tuttavia, questa pratica è considerata un anti-pattern.** Il modo corretto per creare immagini è usare un `Dockerfile`, che rende il processo di build riproducibile e documentato. Il `commit` è utile per il debug o per "salvare" rapidamente uno stato.

3.4. Il Dockerfile: La Ricetta per le Immagini

Un Dockerfile è un file di testo che contiene una serie di istruzioni per assemblare un'immagine Docker in modo automatizzato.

- `FROM`: Specifica l'immagine di base da cui partire (es. `FROM ubuntu:22.04`). È la prima istruzione obbligatoria.
- `WORKDIR`: Imposta la directory di lavoro per le istruzioni successive (`RUN`, `CMD`, `COPY`, etc.).
- `COPY` / `ADD`: Copia file e directory dalla macchina host al filesystem del container.
- `RUN`: Esegue un comando *durante la fase di build* dell'immagine (es. `RUN apt-get update && apt-get install -y vim`). Ogni `RUN` crea un nuovo layer.
- `ENV`: Imposta variabili d'ambiente permanenti.
- `EXPOSE`: Documenta su quale porta di rete il container ascolterà. **Non pubblica la porta**, serve solo come metadato. La pubblicazione avviene con `docker run -p`.
- `CMD`: Fornisce il comando e/o i parametri di default per un container in esecuzione. Può essere sovrascritto dalla riga di comando (`docker run <image> <new_command>`). Ci può essere solo un `CMD`.
- `ENTRYPOINT`: Configura il container per essere eseguito come un eseguibile. Il comando specificato in `ENTRYPOINT` viene sempre eseguito all'avvio, e `CMD` viene passato come argomento all'entrypoint. Non è facilmente sovrascrivibile.

Esempio di Dockerfile:

```
# Usa un'immagine ufficiale di Python come base
FROM python:3.9-slim

# Imposta la directory di lavoro nel container
WORKDIR /app

# Copia i file dei requisiti e installa le dipendenze
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Copia il resto del codice dell'applicazione
COPY . .

# Informa Docker che il container ascolta sulla porta 5000
EXPOSE 5000

# Definisci il comando per avviare l'applicazione
CMD ["python", "app.py"]
```

3.5. Networking in Docker (Approfondimento)

Quando si installa Docker, vengono creati di default alcuni driver di rete. I più importanti sono:

- `bridge` **(Ponte):** Questo è il driver di rete predefinito.
  - **Default Bridge Network:** Quando si avvia un container senza specificare una rete, viene collegato a una rete bridge di default chiamata `bridge` (visibile con `docker network ls`). Su un host Linux, questa rete è rappresentata da un'interfaccia virtuale `docker0`. Ogni container ottiene un proprio indirizzo IP in una sottorete privata (es. 172.17.0.0/16). Docker gestisce il **NAT (Network Address Translation)** per permettere ai container di comunicare con l'esterno.
  - *Limitazione:* I container sulla rete bridge di default possono comunicare tra loro solo tramite indirizzo IP. **Non c'è una risoluzione dei nomi automatica.**
  - **User-defined Bridge Network (Rete Bridge definita dall'utente):** È la pratica consigliata. Creando una rete personalizzata (`docker network create my-app-net`), si ottengono vantaggi significativi.
    - **Miglior Isolamento:** Solo i container collegati esplicitamente a questa rete possono comunicare.
    - **DNS Embedded: Questa è la funzionalità chiave richiesta nella tua domanda.** Docker fornisce un server DNS integrato per ogni rete bridge definita dall'utente. Questo server permette ai container sulla stessa rete di **risolvere i nomi degli altri container**. Ad esempio, un container chiamato `webapp` può raggiungere un container chiamato

`database` semplicemente usando l'hostname `database`. Docker si occuperà di tradurre `database` nel corretto indirizzo IP del container. Questo è fondamentale per le architetture a microservizi, in quanto disaccoppia i servizi dalla conoscenza degli indirizzi IP specifici, che possono cambiare.

- **`host`:** Il container non ha un proprio stack di rete isolato. Condivide direttamente lo stack di rete dell'host.
  - **Vantaggi:** Massime prestazioni di rete, poiché non c'è NAT.
  - **Svantaggi: Nessun isolamento di rete.** Un container può accedere a tutti i servizi di rete dell'host e viceversa. Si perde la capacità di mappare le porte (es. `docker run -p`), poiché il container usa direttamente le porte dell'host. Se il container ha bisogno della porta 80, questa deve essere libera sull'host, e non si possono eseguire due container che necessitano della stessa porta. È un rischio per la sicurezza.
- **`overlay`:** Questo driver è progettato per la **comunicazione multi-host**. Crea una rete distribuita che si estende su più host Docker, permettendo ai container in esecuzione su macchine fisiche diverse di comunicare tra loro come se fossero sulla stessa rete locale. È la tecnologia alla base di **Docker Swarm**.
- **`none`:** Il container viene creato con un proprio stack di rete ma senza alcuna interfaccia di rete esterna, a parte quella di loopback (`lo`). È completamente isolato dal punto di vista della rete.

Risposte dirette alle tue domande:

- **Port Mapping:** Sì, come hai correttamente indicato, con `docker run -p 60000:80` si può mappare una porta dell'host (60000) a una porta del container (80). Docker utilizza **iptables/netfilter** sull'host Linux per creare le regole di NAT (nello specifico, la chain `DOCKER` nella tabella `nat`) che intercettano il traffico sulla porta 60000 dell'host e lo reindirizzano all'IP privato del container sulla porta 80.
- **Attaccare un container a più reti:** Sì, è possibile e comune. Un container viene collegato a una rete al momento della creazione (`docker run --network=<rete> ...`). Per collegarlo a una seconda rete si usa il comando `docker network connect <seconda_rete> <nome_container>`.
  - *Caso d'uso:* Un container "frontend" che esegue un web server potrebbe essere collegato a due reti: una "public-net" per ricevere traffico dall'esterno (esposto tramite `-p`) e una "backend-net" privata per comunicare in modo sicuro con i container del database e di altri servizi, che non sono esposti all'esterno.

4. L'Orchestrazione dei Container: Docker Swarm e Kubernetes

Quando le applicazioni crescono e si basano su decine o centinaia di container distribuiti su più macchine, gestirle manualmente diventa impossibile. Nasce così la necessità di un **orchestratore di**

container. Un orchestratore automatizza il deployment, la scalabilità, la gestione, il networking e la disponibilità delle applicazioni containerizzate.

4.1. `docker-compose` : Orchestrazione su Singolo Host

- **Scopo:** `docker-compose` (ora integrato come `docker compose`) è un tool per definire e gestire applicazioni **multi-container su un singolo host Docker**. Utilizza un file di configurazione in formato **YAML** (`docker-compose.yml`) per descrivere tutti i servizi che compongono l'applicazione, le loro immagini, le reti, i volumi e le dipendenze.
- **Vantaggi:** Semplifica enormemente l'avvio di ambienti di sviluppo e testing complessi. Con un solo comando (`docker compose up`), si avvia l'intera applicazione.
- **Limitazione:** È pensato per un singolo host. Non gestisce cluster di macchine.

4.2. Docker Swarm: Orchestrazione Nativa e Semplificata

Docker Swarm è la soluzione di orchestrazione nativa di Docker, integrata direttamente nel motore Docker. È progettata per essere semplice da usare.

- **Architettura e Concetti Chiave:**
  - **Cluster:** Un insieme di macchine (nodi) che eseguono Docker e lavorano insieme.
  - **Nodo Manager vs. Nodo Worker:**
    - **Manager:** È il cervello del cluster. Le sue responsabilità sono:
      a. **Mantenere lo stato del cluster:** Sa quali servizi devono essere in esecuzione, con quante repliche, e su quali nodi sono attualmente. Per garantire l'alta disponibilità, si possono avere più manager; questi usano il protocollo di consenso **Raft** per eleggere un leader e mantenere lo stato sincronizzato.
      b. **Scheduling:** Decide su quale nodo worker avviare un container (task).
      c. **Gestione API:** Espone le API di Swarm per gestire il cluster.
    - **Worker:** È il "braccio" del cluster. La sua unica responsabilità è ricevere istruzioni (task) dal manager ed eseguirle (avviare/fermare container). Non ha potere decisionale.
  - **Service (Servizio):** È la definizione di un componente della tua applicazione nel cluster. Nel file di configurazione, si definisce un servizio specificando l'immagine del container, il numero di repliche desiderate, le porte da esporre, le reti a cui collegarsi, ecc. Swarm si assicurerà costantemente che il numero di repliche attive corrisponda a quello desiderato (self-healing).
  - **Stack:** Come hai giustamente notato, uno stack è un **gruppo di servizi interconnessi che definiscono un'intera applicazione**. È l'equivalente di un file `docker-compose.yml` applicato a un intero cluster Swarm. Si deploya uno stack con il comando `docker stack deploy -c docker-compose.yml my-app-stack`. Swarm leggerà il file e creerà/aggiornerà tutti i servizi definiti al suo interno su tutto il cluster.

- **Routing Mesh:** È una delle funzionalità più potenti di Swarm. Quando si pubblica una porta per un servizio (es. `ports: "8080:80"`), questa porta viene aperta su **ogni nodo del cluster**, non solo su quelli che effettivamente eseguono un container di quel servizio. Qualsiasi richiesta inviata alla porta 8080 di *qualsiasi* nodo dello swarm verrà automaticamente bilanciata e reindirizzata a un container sano del servizio, indipendentemente dalla macchina su cui si trova. Questo semplifica enormemente il load balancing.

4.3. Kubernetes (K8s): Lo Standard de Facto per l'Orchestrazione

Kubernetes è un progetto open-source originariamente creato da Google, ora gestito dalla Cloud Native Computing Foundation (CNCF). È più complesso di Swarm ma offre una flessibilità e una potenza enormemente maggiori.

- **Architettura e Concetti Chiave:**
  - **Pod:** È l'**unità atomica di deployment** in Kubernetes. Un pod rappresenta un gruppo di uno o più container che vengono sempre deployati insieme sulla stessa macchina (nodo) e condividono lo stesso contesto di esecuzione:
    - **Stesso indirizzo IP e namespace di rete:** I container all'interno di un pod possono comunicare tra loro su `localhost`.
    - **Stesso spazio di porte:** Due container nello stesso pod non possono usare la stessa porta.
    - **Volumi di storage condivisi.**
    - Generalmente, un pod contiene un container principale e, opzionalmente, dei container "sidecar" che svolgono funzioni ausiliarie (es. logging, monitoring, proxy).
  - **Deployment:** È un oggetto di Kubernetes che gestisce i Pod. Il suo scopo è dichiarare uno **stato desiderato**: "Voglio 3 repliche del pod della mia applicazione basato su questa immagine". Il controller del Deployment si assicurerà costantemente che ci siano 3 pod sani in esecuzione. Gestisce gli aggiornamenti (rolling update) e i rollback in modo controllato.
  - **Service (Servizio):** In Kubernetes, un Service è un'**astrazione che definisce un punto di accesso stabile e logico a un insieme di Pod**. I Pod sono effimeri: possono essere distrutti e ricreati con nuovi IP. Un Service fornisce un **IP virtuale stabile e un nome DNS** che non cambiano. Le richieste inviate a questo Service vengono bilanciate e instradate ai Pod sottostanti (selezionati tramite `labels`). Questo disaccoppia i client dai Pod specifici.
  - **Differenza tra Deployment e Service:**
    - Il **Deployment** si occupa del *ciclo di vita* dei Pod (crearli, scalarli, aggiornarli, assicurarsi che esistano). È il "come" e il "quanti".
    - Il **Service** si occupa dell'**accesso di rete** ai Pod (fornire un indirizzo stabile e bilanciare il carico). È il "come li raggiungo".

    - Sono due concetti ortogonali ma che lavorano insieme: si crea un Deployment per eseguire i Pod dell'applicazione e poi si crea un Service per esporli in modo affidabile.

5. Architetture a Microservizi vs. Monoliti

Questa discussione si lega direttamente alla tua prima domanda.

- **Applicazione Monolitica:** L'intera applicazione è costruita come un'unica unità. Frontend, logica di business, accesso ai dati sono tutti strettamente accoppiati in un unico processo.
    - *Pro:* Semplice da sviluppare e deployare all'inizio.
    - *Contro:* Difficile da scalare (devi replicare l'intero monolite, anche se solo una piccola parte è sotto carico), difficile da aggiornare (un piccolo bug richiede il redeploy dell'intera applicazione), fragilità (un errore in un modulo può far crashare tutto), barriera tecnologica (tutta l'applicazione deve usare lo stesso stack tecnologico).
- **Architettura a Microservizi:** L'applicazione è scomposta in un insieme di piccoli servizi indipendenti. Ogni servizio è responsabile di una specifica capacità di business, viene sviluppato e deployato autonomamente e comunica con gli altri tramite API ben definite (solitamente via rete, es. REST o gRPC).
    - *Pro:* Scalabilità granulare (scali solo i servizi che ne hanno bisogno), resilienza (il fallimento di un servizio non blocca l'intera applicazione), flessibilità tecnologica (ogni servizio può usare la tecnologia più adatta), team indipendenti possono lavorare in parallelo.
    - *Contro:* Complessità operativa (gestire decine di servizi, il networking, la service discovery, il monitoraggio distribuito), latenza di rete tra servizi.

Risposta alla domanda "Un container è un microservizio?":

La tua risposta è eccellente e corretta. Riassumendo e ampliando:

- **No, non intrinsecamente.** Un container è una **tecnologia di packaging e deployment**. Puoi tranquillamente "containerizzare" un'applicazione monolitica. In questo caso, avrai un monolite che gira in un container. Ottieni i benefici della portabilità e dell'isolamento di Docker, ma l'architettura interna dell'applicazione rimane monolitica.
- **Sì, se fa parte di un'architettura più grande.** Un container diventa la *realizzazione pratica* di un microservizio quando:
    i. **Implementa una singola responsabilità di business** (es. servizio utenti, servizio pagamenti).
    ii. **Comunica con altri container/servizi tramite API di rete** (invece di chiamate a funzioni interne).
    iii. **È deployabile e scalabile in modo indipendente.** L'uso di orchestratori come Swarm o Kubernetes è la naturale conseguenza di questa necessità: si definisce il "servizio utenti" come

un `Service` (in K8s) o uno `stack service` (in Swarm) e l'orchestratore si occupa di eseguirlo in N repliche containerizzate e di gestirne il ciclo di vita.

La containerizzazione non *crea* microservizi, ma è la tecnologia che li ha resi *praticabili ed efficaci* su larga scala.

Perfetto, ecco la terza parte degli appunti. Questa sezione si concentra su sicurezza, autenticazione, gestione delle identità e i protocolli di rete di supporto, argomenti cruciali in qualsiasi sistema distribuito.

Parte 3: Sicurezza, Autenticazione e Protocolli di Rete

6. Concetti Fondamentali di Sicurezza e Identity Management

6.1. Authentication vs. Authorization (Autenticazione vs. Autorizzazione)

Questi due concetti sono il fondamento della gestione degli accessi, ma sono distinti e sequenziali.

- **Authentication (AuthN): Chi sei?**
  - **Definizione:** È il processo di **verificare l'identità** dichiarata da un'entità (utente, servizio, dispositivo). Il sistema risponde alla domanda: "Sei davvero chi dici di essere?".
  - **Meccanismi comuni:**
    - Qualcosa che conosci (es. password, PIN).
    - Qualcosa che hai (es. token fisico, smart card, chiave OTP generata da un'app).
    - Qualcosa che sei (es. impronta digitale, scansione del volto - biometria).
  - **Analogia:** Mostrare la carta d'identità a un buttafuori per dimostrare di essere maggiorenne.
- **Authorization (AuthZ): Cosa puoi fare?**
  - **Definizione:** È il processo che avviene **dopo** un'autenticazione riuscita. Determina a quali risorse o azioni un'identità autenticata ha il permesso di accedere. Il sistema risponde alla domanda: "Ora che so chi sei, cosa ti è concesso fare?".
  - **Meccanismi comuni:** Liste di Controllo degli Accessi (ACL), Role-Based Access Control (RBAC), attributi.
  - **Analogia:** Una volta entrato nel locale (autenticazione), il pass che hai (normale, VIP, backstage) determina a quali aree puoi accedere (autorizzazione).

L'autenticazione precede sempre l'autorizzazione. Non ha senso decidere cosa un utente può fare se non si è certi della sua identità.

6.2. IAM (Identity and Access Management)

Un sistema IAM è l'infrastruttura centralizzata che gestisce le identità digitali e le regole di autorizzazione.

- **Vantaggi di un IAM Centralizzato:**
  i. **Single Source of Truth (Fonte Unica di Verità):** Tutte le identità e i loro permessi sono gestiti in un unico posto. Questo previene la duplicazione e la desincronizzazione dei dati.
  ii. **Sicurezza Migliorata (Reduced Attack Surface):** Invece di dover proteggere decine di sistemi di autenticazione diversi (uno per ogni applicazione), si concentra la protezione su un unico sistema IAM robusto. L'attacco è più difficile.
  iii. **Gestione e Manutenibilità Semplificate:** Creare, modificare o revocare l'accesso di un utente viene fatto una sola volta nel sistema centrale, e le modifiche si propagano a tutti i servizi collegati.
  iv. **Auditing e Compliance Centralizzati:** È molto più facile tracciare chi ha fatto cosa e quando, poiché tutti i log di accesso convergono in un unico punto.
  v. **Esperienza Utente Migliorata (Single Sign-On - SSO):** L'utente si autentica una sola volta presso l'IAM e ottiene l'accesso a tutte le applicazioni federate senza dover inserire nuovamente le credenziali.
- **Identity Provider (IdP) e Service Provider (SP):**
  - **Identity Provider (IdP):** È il servizio che **gestisce e autentica le identità**. È il cuore dell'IAM centralizzato. Conserva il database degli utenti (es. username/password) e, quando richiesto, afferma ("assertion") l'identità di un utente a un Service Provider. Esempio: il sistema di login della tua università, l'account Google.
  - **Service Provider (SP):** È il servizio a cui l'utente vuole accedere (es. una piattaforma di e-learning, un'app web, Google Drive). L'SP **non gestisce le password**, ma si fida ("trusts") dell'IdP per l'autenticazione.
  - **Flusso Tipico (Federated Identity):**
    a. L'utente tenta di accedere all'SP (es. `servizio.com`).
    b. L'SP non conosce l'utente e lo reindirizza all'IdP (`login.universita.it`), includendo una richiesta di autenticazione.
    c. L'utente inserisce le proprie credenziali sull'IdP.
    d. L'IdP, se l'autenticazione ha successo, genera un **token/assertion** (un pezzo di dati firmato digitalmente che attesta l'identità e gli attributi dell'utente) e lo invia al browser dell'utente.
    e. Il browser dell'utente presenta questo token all'SP.
    f. L'SP verifica la firma digitale del token usando la chiave pubblica dell'IdP (con cui ha una relazione di fiducia pre-stabilita) e, se valida, concede l'accesso all'utente.

6.3. SAML 2.0 e Shibboleth

- **SAML (Security Assertion Markup Language) 2.0:** È uno **standard aperto basato su XML** per scambiare dati di autenticazione e autorizzazione tra un IdP e un SP. SAML definisce il formato dei

messaggi (le "asserzioni") e i protocolli per questo scambio. È uno dei pilastri della federazione di identità sul web.

- **Shibboleth:** È una delle più popolari **implementazioni open-source** dello standard SAML 2.0. È ampiamente utilizzato nel mondo accademico e della ricerca per fornire accesso Single Sign-On (SSO) a risorse web distribuite tra diverse istituzioni. In pratica, Shibboleth è il software che fa da IdP o da SP in un'architettura federata basata su SAML.

6.4. Integrità e Segretezza dei Dati

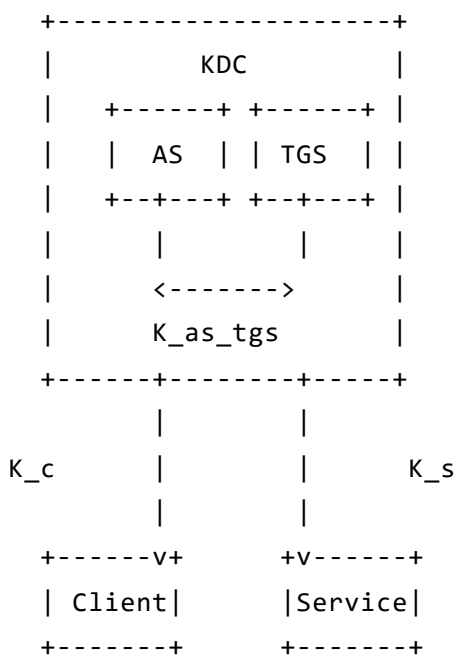
- **Concetto di Integrità:** In sicurezza informatica, l'integrità garantisce che i dati non siano stati **alterati o manomessi** in modo non autorizzato durante la loro archiviazione (at rest) o trasmissione (in transit).
  - *Forme di Integrità:*
    a. **Integrità dei Dati:** Protezione da modifiche accidentali o malevole (es. un checksum come SHA-256 su un file scaricato verifica che non sia corrotto).
    b. **Integrità del Messaggio:** Garanzia che un messaggio ricevuto sia identico a quello inviato.
  - *Non-ripudiabilità:* È un concetto correlato ma più forte. Garantisce che il mittente non possa negare di aver inviato il messaggio. Richiede sia l'integrità sia un'autenticazione forte del mittente (tipicamente tramite firma digitale).
- **Perché MAC e Firma Digitale non offrono Segretezza:**
  - **Scopo:** Sia il **MAC (Message Authentication Code)** sia la **Firma Digitale** hanno lo scopo di garantire **integrità e autenticità**, non segretezza (confidenzialità).
  - **Funzionamento:**
    a. Si calcola un **hash** (un digest a lunghezza fissa, es. SHA-256) del messaggio in chiaro.
    b. Questo hash (e non l'intero messaggio) viene "protetto":
       - **MAC:** L'hash viene combinato con una **chiave segreta condivisa** tra mittente e destinatario. Il risultato è il MAC.
       - **Firma Digitale:** L'hash viene cifrato con la **chiave privata** del mittente. Il risultato è la firma.
    c. Il messaggio originale **in chiaro** e il MAC/firma vengono inviati insieme.
  - **Conclusione:** Poiché il messaggio viaggia in chiaro, chiunque lo intercetti può leggerlo. La segretezza si ottiene solo **cifrando l'intero corpo del messaggio**, tipicamente con una chiave simmetrica. Spesso, si usano entrambe le tecniche: si cifra il messaggio per la segretezza e si firma il messaggio cifrato per l'integrità e l'autenticità.

7. Protocolli di Autenticazione e Directory Services

7.1. Kerberos

Kerberos è un protocollo di autenticazione di rete basato su un sistema di "ticket" per permettere a nodi che comunicano su una rete insicura di provare la loro identità in modo sicuro. È il protocollo di autenticazione di default in Active Directory di Microsoft.

- **Schema delle Chiavi Segrete Statiche:** Kerberos si basa su un'architettura di fiducia centralizzata basata su chiavi simmetriche.
  - **Entità:** Client (C), Service (S), Authentication Server (AS), Ticket-Granting Server (TGS). L'AS e il TGS insieme formano il **Key Distribution Center (KDC)**.
  - **Chiavi Statiche (a lungo termine):**
    a. **K_c:** Una chiave segreta condivisa solo tra il **Client (C)** e l'**AS**. Questa chiave è tipicamente derivata dalla password dell'utente.
    b. **K_s:** Una chiave segreta condivisa solo tra il **Service (S)** e il **TGS**.
    c. **K_as_tgs:** Una chiave segreta condivisa solo tra l'**AS** e il **TGS** per la loro comunicazione interna sicura.
  - **Schema Grafico Semplificato:**



- **Uso di Kerberos:** Non tutte le applicazioni possono usarlo nativamente. Un'applicazione deve essere "kerberizzata", cioè modificata per includere le librerie client di Kerberos e per saper:
  i. Richiedere un Ticket-Granting Ticket (TGT) all'AS all'avvio.
  ii. Usare il TGT per richiedere un Service Ticket al TGS per un servizio specifico.
  iii. Presentare il Service Ticket al servizio per autenticarsi.

iv. Verificare il ticket ricevuto dal client (se l'applicazione è un servizio).

7.2. Directory Services e LDAP

- **Schema di un Directory Service:** È l'insieme di regole che definisce la struttura di un servizio di directory. Specifica:
  - **Object Classes:** I tipi di oggetti che possono essere memorizzati (es. `user`, `group`, `organizationalUnit`).
  - **Attributes:** Le informazioni che ogni classe di oggetti può o deve contenere (es. un oggetto `user` deve avere un `cn` - common name - e può avere un `telephoneNumber`).
  - **Sintassi degli Attributi:** Il tipo di dato per ogni attributo (es. stringa, intero, binario). L'analogia con lo schema E-R di un database relazionale è molto calzante. Lo schema garantisce consistenza e validità dei dati nella directory.
- **Active Directory: Foresta vs. Albero (Forest vs. Tree)**
  - **Dominio (Domain):** Unità di base di amministrazione e sicurezza (es. `sales.mycorp.com`).
  - **Albero (Tree):** Una collezione di uno o più domini che condividono uno **spazio dei nomi DNS contiguo**. Il primo dominio creato in un albero è la radice dell'albero (es. `mycorp.com`). Altri domini come `sales.mycorp.com` sono figli. Tra tutti i domini di un albero esiste una relazione di trust (fiducia) automatica, bidirezionale e transitiva.
  - **Foresta (Forest):** È il confine di sicurezza più alto in Active Directory. È una collezione di **uno o più alberi che non condividono necessariamente uno spazio dei nomi contiguo**. Tutti gli alberi in una foresta condividono uno schema comune e un Global Catalog (un indice parziale di tutti gli oggetti della foresta). Esempio: La foresta `MyOrg` potrebbe contenere l'albero `mycorp.com` e l'albero `altra-azienda-acquisita.net`. Le relazioni di trust tra alberi diversi nella stessa foresta sono anch'esse automatiche e transitive.

8. Protocolli di Supporto alla Rete

8.1. Sincronizzazione Temporale (NTP)

- **Utilità della Sincronizzazione:** Anche in una rete isolata, avere un tempo comune è fondamentale per:
  - **Ordinamento degli eventi:** I log di sistema provenienti da macchine diverse possono essere correlati correttamente solo se gli timestamp sono sincronizzati. Essenziale per il troubleshooting e la forensics.
  - **Protocolli di sicurezza:** Kerberos, come menzionato, è estremamente sensibile alla sincronizzazione temporale. I ticket hanno una validità temporale limitata per prevenire attacchi

di tipo "replay". Se l'orologio di un client e di un server differiscono di più di qualche minuto (skew), l'autenticazione fallisce.
    - **Sistemi distribuiti:** Algoritmi di consenso, database distribuiti e file system di rete si basano spesso su una nozione condivisa di tempo.
- **NTP (Network Time Protocol) e Stratum:**
    - **Stratum:** Indica la "distanza" in termini di hop da un orologio di riferimento.
        - **Stratum 0:** Sono i dispositivi di riferimento di alta precisione (orologi atomici, ricevitori GPS). **Non sono server di rete**, sono la fonte del tempo.
        - **Stratum 1:** Sono i server di rete direttamente collegati a un dispositivo di Stratum 0. Sono i server più precisi disponibili in rete.
        - **Stratum 2:** Sono server che si sincronizzano con un server di Stratum 1.
        - **Stratum n:** Server che si sincronizzano con un server di Stratum n-1.
    - La raccomandazione di usare server di Stratum 2 è per distribuire il carico e non sovraccaricare i pochi e critici server di Stratum 1.

8.2. NAT Traversal (STUN / TURN)

Questi protocolli sono cruciali per stabilire comunicazioni Peer-to-Peer (P2P), come chiamate VoIP o video, quando i client si trovano dietro a un NAT.

- **STUN (Session Traversal Utilities for NAT):**
    - **Scopo:** Un client contatta un server STUN pubblico su Internet. Il server STUN risponde dicendo al client qual è il suo indirizzo IP pubblico e la porta vista dall'esterno (l'indirizzo "mappato" dal NAT). STUN aiuta anche a **determinare il tipo di NAT**.
    - **Funzionamento:** Se due client conoscono i rispettivi indirizzi IP pubblici e il se due client conoscono i rispettivi indirizzi IP pubblici e le porte mappate dal loro NAT, e se almeno uno dei due si trova dietro a un **NAT non-simmetrico (o "a cono")**, possono tentare una tecnica chiamata **"NAT hole punching"**. Entrambi i client inviano simultaneamente pacchetti l'uno verso l'altro. Questo "buca" il NAT, creando una sessione che permette una comunicazione P2P diretta.
- **Limite di STUN:** STUN fallisce quando entrambi i client si trovano dietro a **NAT simmetrici**. Un NAT simmetrico mappa la stessa porta interna a porte esterne diverse a seconda dell'IP di destinazione. Quindi, la porta che il client A usa per parlare con il server STUN non è la stessa che verrebbe usata per parlare con il client B. La connessione diretta è impossibile.

TURN (Traversal Using Relays around NAT):

- **Scopo:** TURN è la soluzione di ripiego quando la comunicazione P2P diretta fallisce (tipicamente a causa di NAT simmetrici o firewall restrittivi). Non serve a *stabilire* una connessione diretta, ma a **inoltrare (relay) il traffico** tra i due peer.

- **Funzionamento:**
  i. Ciascun client stabilisce una connessione con il server TURN.
  ii. Il server TURN alloca delle risorse e un indirizzo di inoltro per ciascun client.
  iii. Quando il client A vuole inviare dati al client B, li invia al server TURN, incapsulati in un pacchetto TURN.
  iv. Il server TURN riceve il pacchetto, lo "scarta" e inoltra i dati originali al client B.
- **Svantaggi:** È costoso. Tutto il traffico media passa attraverso il server TURN, consumando la sua banda e CPU. Per questo motivo, è sempre l'ultima risorsa.

Differenza principale tra STUN e TURN:

Caratteristica	Server STUN	Server TURN
Scopo	**Scoprire** l'indirizzo IP pubblico e il tipo di NAT. Aiutare a stabilire una connessione diretta (P2P).	**Inoltrare** il traffico quando la connessione diretta non è possibile. Agire da intermediario.
Flusso Dati	Il traffico dati **non** passa attraverso il server STUN, se non per un breve scambio iniziale.	**Tutto** il traffico dati tra i peer passa attraverso il server TURN.
Risorse	Leggero, basso consumo di banda.	Pesante, alto consumo di banda e CPU, richiede una buona infrastruttura.
Scenario d'uso	Prima opzione da tentare per la comunicazione P2P.	Soluzione di ripiego quando STUN e il "hole punching" falliscono.

ICE (Interactive Connectivity Establishment): È un framework che utilizza sia STUN sia TURN per trovare il percorso di comunicazione più efficiente tra due peer. ICE raccoglie tutti i possibili "candidati" di indirizzi (l'IP locale, l'IP pubblico scoperto da STUN, l'indirizzo di inoltro di TURN) e li testa in ordine di preferenza (diretto > via STUN > via TURN) per stabilire la migliore connessione possibile.

9. Argomenti di Approfondimento

9.1. GDPR (General Data Protection Regulation)

Il GDPR è un regolamento dell'Unione Europea sulla protezione dei dati e della privacy per tutti i cittadini dell'UE. È diventato legge nel 2018.

- **Scopo:** Dare ai cittadini il controllo sui propri dati personali e unificare le normative sulla privacy in tutta Europa.
- **Dati Personali:** Qualsiasi informazione relativa a una persona fisica identificata o identificabile. Questo include nomi, indirizzi, e-mail, dati di localizzazione, indirizzi IP, cookie, ma anche dati più sensibili come dati sanitari, biometrici, genetici e opinioni politiche.
- **Risposta alla domanda: Sì, l'ammontare dello stipendio di un dipendente è assolutamente un dato personale protetto dal GDPR.** È un'informazione direttamente collegata a una persona identificabile. Le aziende (in qualità di "titolari del trattamento") devono trattare questo dato con la massima cura, garantendone la liceità del trattamento (es. per l'esecuzione di un contratto di lavoro), la riservatezza, e limitandone l'accesso solo al personale autorizzato (es. ufficio paghe, HR).

9.2. NIS (Network Information Service)

- **Cos'è:** Originariamente chiamato "Yellow Pages" (YP), NIS è un protocollo di directory service client-server sviluppato da Sun Microsystems per la **gestione centralizzata di dati amministrativi in reti di sistemi Unix/Linux**.
- **Scopo:** Semplificare l'amministrazione di una rete distribuita consentendo di mantenere dati come account utente (`/etc/passwd`), gruppi (`/etc/group`), nomi host (`/etc/hosts`), ecc., in un unico posto (il server NIS) invece che su ogni singola macchina.
- **Stato attuale:** NIS è considerato **obsoleto e insicuro**. Le sue debolezze principali sono la mancanza di meccanismi di sicurezza robusti (i dati viaggiano spesso in chiaro) e una struttura non gerarchica e poco flessibile. Oggi è stato quasi completamente sostituito da soluzioni più moderne, sicure e potenti come **LDAP** (spesso implementato con OpenLDAP) o sistemi integrati come Active Directory con i suoi servizi per Unix/Linux.

9.3. VoIP e SDP

- **VoIP (Voice over IP):** È la tecnologia che permette di effettuare conversazioni telefoniche utilizzando una rete basata sul protocollo IP (come Internet) invece della tradizionale rete telefonica pubblica (PSTN).
- **Protocolli Coinvolti:** Una chiamata VoIP richiede due tipi di protocolli:
    i. **Protocolli di Segnalazione:** Gestiscono la creazione, la modifica e la terminazione della chiamata (es. "chiamare", "squillare", "rispondere", "riagganciare"). L'esempio più comune è **SIP (Session Initiation Protocol)**.
    ii. **Protocolli di Trasporto Media:** Trasportano i dati audio/video effettivi una volta che la chiamata è stata stabilita. L'esempio più comune è **RTP (Real-time Transport Protocol)**.
- **Scopo di SDP (Session Description Protocol):**
    - SDP **non è un protocollo di trasporto**, ma un **formato per descrivere** una sessione multimediale. È un semplice testo con coppie `tipo=valore` .

- Il suo scopo è comunicare ai partecipanti alla chiamata quali sono i **parametri della sessione media**:
  - Indirizzi IP e porte su cui inviare/ricevere i flussi RTP (audio, video).
  - Codec supportati (es. G.711, Opus per l'audio; H.264 per il video) e il loro ordine di preferenza.
  - Informazioni sul timing e sulla larghezza di banda.
- In pratica, un messaggio di segnalazione SIP (es. un invito a una chiamata) contiene al suo interno un "body" formattato secondo lo standard SDP per negoziare le caratteristiche tecniche della chiamata prima che i flussi multimediali RTP inizino a scorrere.

9.4. SASL (Simple Authentication and Security Layer)

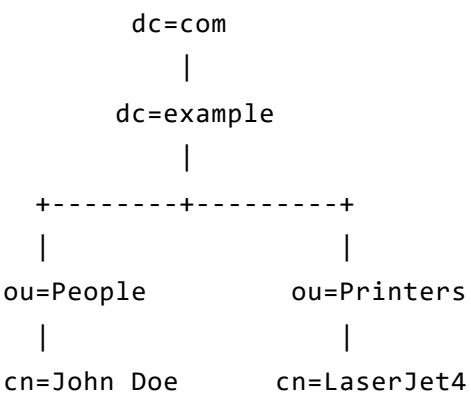
- **Cos'è:** SASL è un **framework** (RFC 4422) che **disaccoppia i meccanismi di autenticazione dai protocolli applicativi**.
- **Scopo:** Permettere a un protocollo applicativo (es. SMTP, LDAP, AMQP) di supportare vari meccanismi di autenticazione senza doverli implementare direttamente nel protocollo stesso.
- **Funzionamento:**
  i. Un client si connette a un server che supporta SASL.
  ii. Il server annuncia al client una lista di meccanismi di autenticazione che supporta (es. `PLAIN`, `LOGIN`, `CRAM-MD5`, `GSSAPI` per Kerberos, `EXTERNAL`).
  iii. Il client sceglie un meccanismo dalla lista (che anche lui supporta) e inizia una sequenza di "sfida-risposta" (challenge-response) definita da quel meccanismo.
  iv. Lo scambio avviene all'interno del protocollo applicativo, ma la sua logica è opaca al protocollo stesso.
- **Vantaggio:** Un'applicazione può aggiungere il supporto per nuovi e più sicuri metodi di autenticazione semplicemente aggiungendo un nuovo modulo SASL, senza dover modificare il codice del protocollo di base.

9.5. Modello dei Nomi LDAP (LDAP Naming e DIT)

- **DIT (Directory Information Tree):** I dati in una directory LDAP sono organizzati in una **struttura ad albero gerarchica**, chiamata DIT. La radice dell'albero è detta "suffix" (es. `dc=example,dc=com`).
- **Entry (Voce):** Ogni nodo dell'albero è una "entry" che rappresenta un oggetto (es. un utente, un gruppo, una stampante).
- **Distinguished Name (DN):** Ogni entry nell'albero ha un **nome univoco globale**, il DN. Il DN è formato concatenando il nome dell'entry stessa con i nomi dei suoi antenati fino alla radice. È l'equivalente di un percorso assoluto in un file system.
- **Relative Distinguished Name (RDN):** È il nome di un'entry **relativo al suo genitore**. È l'equivalente di un nome di file all'interno di una directory. L'RDN è composto da una o più coppie

```
    attributo=valore .
```

Esempio pratico:



- Il **DN** dell'utente "John Doe" è: `cn=John Doe,ou=People,dc=example,dc=com`
- L'**RDN** di "John Doe" (relativo a `ou=People`) è: `cn=John Doe`
- Le componenti comuni sono:
  - `cn` (Common Name): Il nome comune dell'oggetto.
  - `ou` (Organizational Unit): Un'unità organizzativa (es. un dipartimento).
  - `dc` (Domain Component): Una parte del nome di dominio DNS (usato per mappare la struttura DNS alla struttura LDAP).

Spero che questi appunti approfonditi ti siano di grande aiuto per la preparazione del tuo esame. In bocca al lupo