Università degli Studi di Roma “Tor Vergata”
Facoltà di Ingegneria

Architetture dei Sistemi Distribuiti

Corso di Sistemi Distribuiti

Valeria Cardellini
Anno accademico 2009/10

Architettura sw di un SD

- Definisce l’organizzazione e l’interazione dei vari componenti software che costituiscono il SD
- Diverse scelte possibili nella realizzazione di un SD

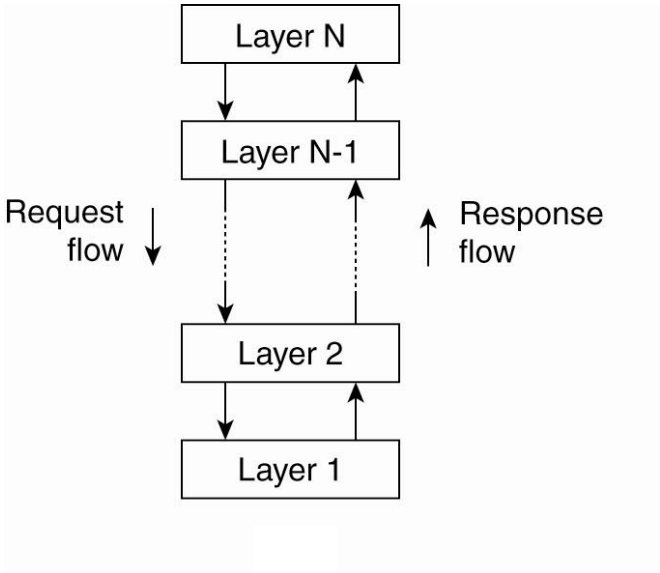
- Architettura di sistema: istanziazione finale di un’architettura software
  - I vari componenti del SD sono posizionati e istanziati sulle diverse macchine che costituiscono il sistema

Stili (*pattern*) architetturali

- Espressi in termini di definizione e uso di componenti e connettori
- Componente
  - Unità modulare dotata di interfacce ben definite
  - Completamente sostituibile nel suo ambiente
- Connettore
  - Meccanismo che consente comunicazione, coordinamento o cooperazione tra componenti
- Stili architetturali prevalenti per SD
  - Architetture a livelli (layer)
  - Architetture basate su oggetti
  - Architetture basate su eventi
  - Architetture orientate ai dati

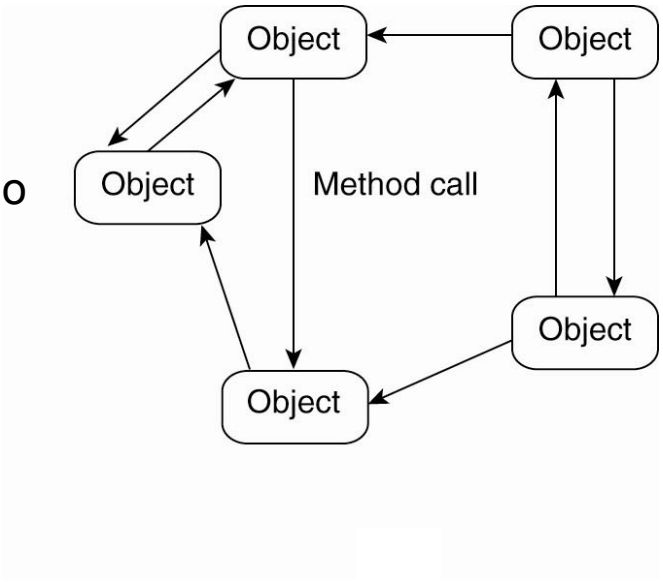
Architettura a livelli

- Componenti organizzati in livelli (*layer*)
- Un componente a livello *i* può invocare un componente del livello sottostante *i-1*
- Le richieste scendono lungo la gerarchia, mentre le risposte risalgono
- Largamente adottato dalla comunità della rete



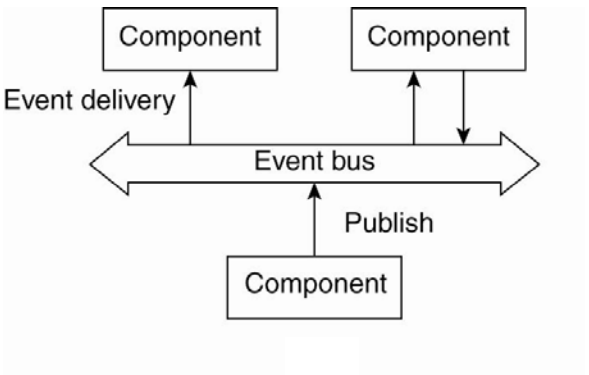
Architettura basata su oggetti

- Ogni componente è un oggetto
- Oggetti connessi attraverso un meccanismo di chiamata a procedura remota (o invocazione di metodo remota)



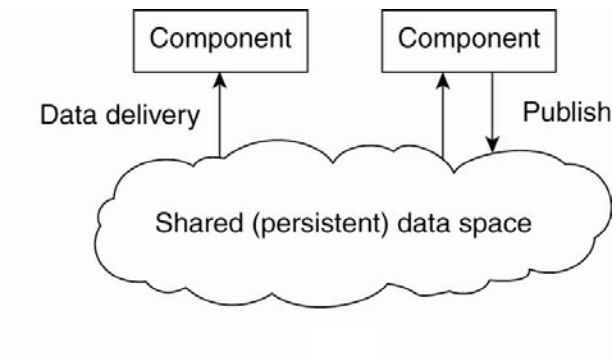
Evoluzione degli stili architetturali

- Il disaccoppiamento dei componenti nello spazio ("anonimi") ed anche nel tempo ("asincroni") ha determinato stili architetturali alternativi



Architettura basata su eventi

I componenti comunicano tramite la propagazione di eventi



Architettura orientata ai dati

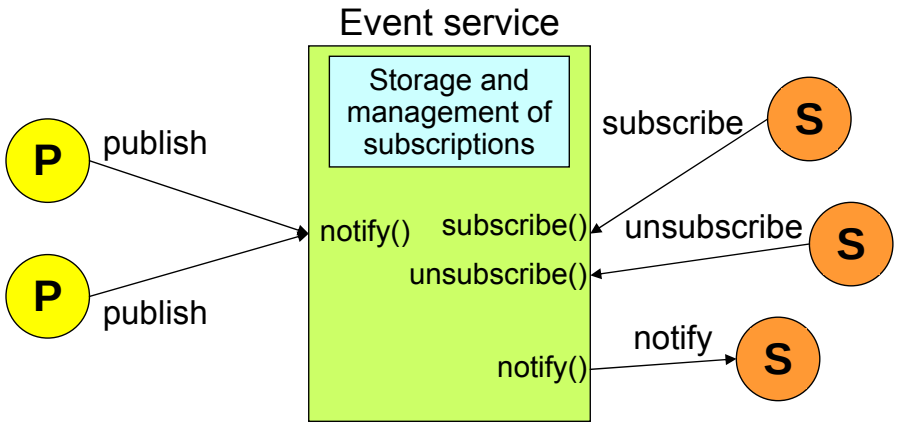
I componenti comunicano tramite un repository comune (passivo o attivo), ad es. un file system distribuito

Disaccoppiamento

- Porre vincoli sui componenti che possono interagire può introdurre necessità di conoscenze che a volte non sono necessarie
- Il disaccoppiamento diventa un fattore per l'abilitazione di maggiore flessibilità e per arrivare a definire stili architetturali che possono sfruttare al meglio la potenziale distribuzione e scalabilità
- Proprietà di **disaccoppiamento**
  - Spaziale: i componenti non devono conoscersi (reciprocamente o meno)
  - Temporale: i componenti interagenti non devono essere compresenti (presenti insieme nello stesso instante) quando la comunicazione ha luogo
  - Di sincronia: i componenti interagenti non devono aspettarsi a vicenda e non sono soggetti a blocchi reciproci

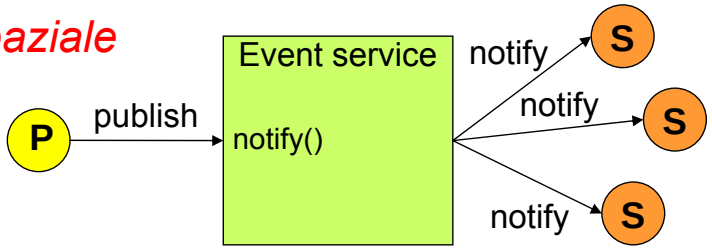
Publish/subscribe

- I produttori generano eventi (publish) e si disinteressano della loro consegna
- I consumatori si sono registrati come interessati ad un evento (subscribe) e sono avvisati (notify) della sua occorrenza
- **Disaccoppiamento** tra entità interagenti

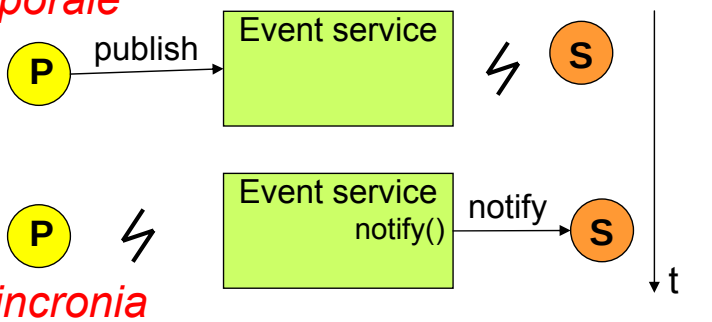


Publish/subscribe e disaccoppiamento

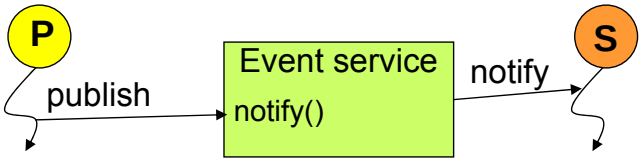
- Disaccoppiamento *spaziale*



- Disaccoppiamento *temporale*



- Disaccoppiamento *di sincronia*



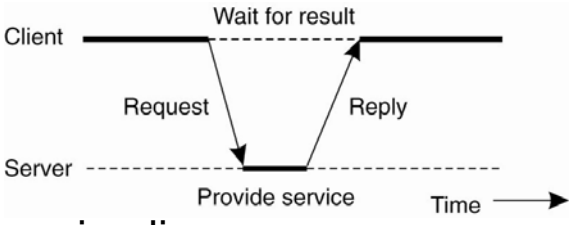
Riferimento: P.T. Eugster *et al*, "The many faces of publish/subscribe" *ACM Comput. Surv.* 35(2):114-131, June 2003.

Architetture di sistema

- Quali componenti software usare?
- Come interagiscono i componenti?
- Dove sono posizionati i componenti?

- Tipologie di architetture di sistema
  - Architetture centralizzate
  - Architetture decentralizzate
  - Architetture ibride

Architetture centralizzate



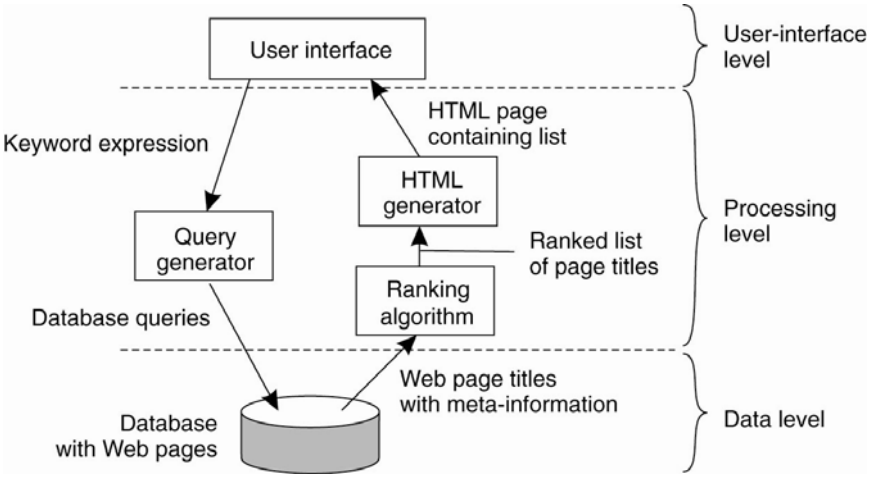
- E' il modello client/server
  - Già noto
  - Intrinsecamente distribuito
- L'interazione tra client e server implica un comportamento "request-reply"
  - Una sorgente del problema di prestazioni nel Web
  - Alcune richieste (ma non tutte) sono *idempotenti*
    - Possono essere ripetute più volte senza causare danni o problemi
    - Proprietà importante in caso di comunicazioni inaffidabili
    - Rendere logicamente affidabile una connessione fisicamente inaffidabile ha un costo molto elevato
- Asimmetria con il client che conosce il server e interagisce in modo *sincrono* e *bloccante* (di default)
- Forte accoppiamento
  - Compresenza di chi interagisce

Stratificazione

- Il tradizionale stile architetturale a livelli applicato alle architetture centralizzate
  - Livello dell'interfaccia utente
    - Spesso suddiviso in due livelli (livello client e livello di presentazione lato server)
  - Livello applicativo
  - Livello dei dati
- Stratificazione presente in molti sistemi informativi distribuiti, che implementano il livello dei dati mediante basi di dati relazionali, ad oggetti o object-relational

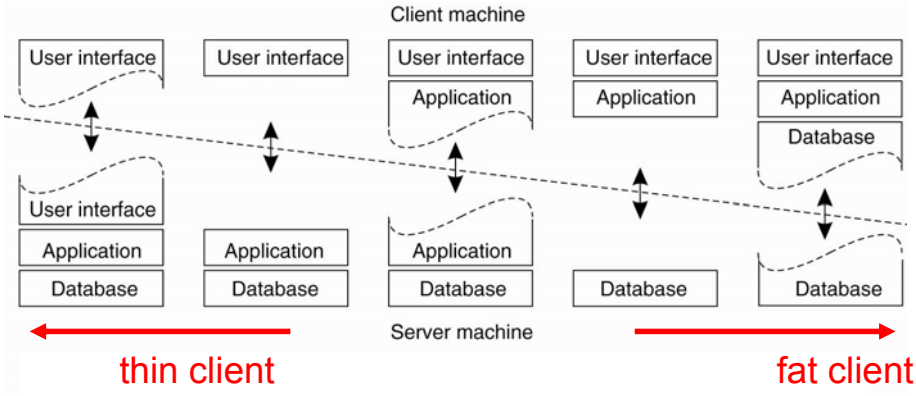
Stratificazione delle applicazioni (2)

- Esempio: motore di ricerca Web



Architetture multilivello

- Mapping tra livelli logici (layer) e livelli fisici (tier)
- Architettura ad un livello (single-tier): configurazione monolitica mainframe e terminale "stupido" (non è C/S!)
- Architettura a due livelli (two-tier): due livelli fisici (macchina client/singolo server)
- Architettura a tre livelli (three-tier): ciascun livello su una macchina separata
- Diverse configurazioni two-tier



Architetture multilivello (2)

- Da un livello ad N livelli
- Con l'introduzione di ciascun livello
  - L'architettura guadagna in flessibilità, funzionalità e possibilità di distribuzione
- Ma l'architettura multilivello potrebbe introdurre un problema di prestazioni
  - Aumenta il costo della comunicazione
  - Viene introdotta più complessità, in termini di gestione ed ottimizzazione

Architetture multilivello (3)

- Sono possibili varie architetture client-server
  - In relazione all'organizzazione del servizio e dei suoi dati
- Distribuzione verticale
  - Componenti logicamente diversi dello stesso servizio possono essere assegnati a macchine distinte
    - Sia sul lato server che sul lato client (delega parziale)
  - Servizio reso tramite la cooperazione di componenti distribuiti
    - Ripartizione gerarchica (anche di autorità)
- Distribuzione orizzontale
  - Server e client possono essere partizionati ma ogni loro componente può operare da solo
  - Condivisione del carico (con ripartizione del lavoro gestita da un dispatcher)
  - Ogni componente sa fornire "il" servizio richiesto
  - Esempio che esamineremo: Web cluster

Architetture decentralizzate

- … ovvero i **sistemi peer-to-peer** (P2P)
- Il termine peer-to-peer si riferisce ad una classe di sistemi ed applicazioni che utilizzano *risorse distribuite* per eseguire una funzionalità (anche critica) in modo *decentralizzato*
- Peer: entità con capacità simili alle altre entità nel sistema
- *Condivisione* delle risorse (cicli di CPU, storage, dati)
  - Offrire ed ottenere risorse dalla comunità di peer

Caratteristiche dei sistemi P2P

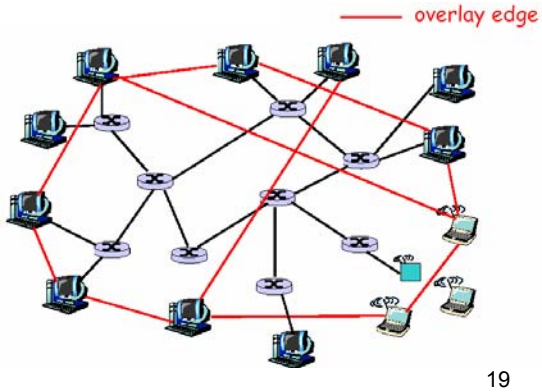
- Tutti i nodi (peer) del sistema hanno identiche capacità e responsabilità (almeno in linea di principio)
  - Nodi *indipendenti* (autonomi) e localizzati ai bordi (*edge*) di Internet
    - Nessun controllo centralizzato
  - Ogni peer ha funzioni di client e server e condivide risorse e servizi
    - **servent** = server + client
  - In realtà, possono essere presenti server centralizzati o nodi con funzionalità diverse rispetto agli altri (*supernodi*)
- Sistemi altamente distribuiti
  - Il numero di nodi può essere dell'ordine delle centinaia di migliaia
- Nodi altamente dinamici ed autonomi
  - Un nodo può entrare o uscire dalla rete P2P in ogni momento
    - Operazioni di ingresso/uscita (join/leave) dalla rete
  - Ridondanza delle informazioni

Applicazioni P2P

- Distribuzione e memorizzazione di contenuti
  - Contenuti: file, video streaming (incluso streaming media), …
  - Reti, protocolli e client per *file sharing* (la "killer application" del P2P): Gnutella, FastTrack, eDonkey, BitTorrent, LimeWire, eMule, iMesh, uTorrent, …
  - *File storage*: Freenet, …
- Condivisione di risorse di calcolo (elaborazione distribuita)
  - Es.: SETI@home – Search for Extraterrestrial Intelligence
- Collaborazione e comunicazione
  - Es.: Chat/Irc, Instant Messaging, Jabber
- Telefonia
  - Es.: Skipe
- Content Delivery Network
  - Es.: CoralCDN
- Piattaforme
  - Es:. JXTA (http://www.sun.com/software/jxta/)

Overlay network

- **Overlay network**: è la rete virtuale che interconnette i peer
  - I nodi sono costituiti dai processi
  - I collegamenti rappresentano i possibili canali di comunicazione
  - Fornisce un servizio di localizzazione delle risorse
  - Basata su una rete fisica sottostante
  - Vantaggi: semplicità e velocità di sviluppo di nuove applicazioni
- Instradamento a livello applicativo
  - Routing *content-aware* ⬅
  - Routing *application semantic-aware*
    - La localizzazione si basa sulla vicinanza semantica delle risorse contenute in un nodo



Overlay routing

- Idea base:
  - Il sistema fa trovare la strada per raggiungere una risorsa
- Rispetto al routing tradizionale
  - La risorsa non è esattamente un indirizzo di un nodo della rete, ma può essere un file, una CPU disponibile, dello spazio libero su disco, …
- Concentriamo l'attenzione sul routing, non sull'interazione tra peer per recuperare una risorsa
  - Il recupero avviene con un'interazione diretta tra peer, usando protocolli come HTTP
- Tipi di reti overlay:
  - Reti overlay **non strutturate**
  - Reti overlay **strutturate**

Reti non strutturate

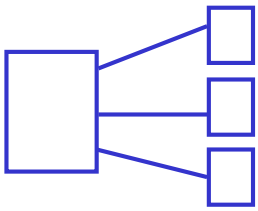
- Reti **non strutturate**
  - La costruzione della rete overlay si basa su algoritmi causali
  - Nodi organizzati in modo disordinato (grafo random)
    - L'organizzazione della rete segue principi molto semplici
  - Non ci sono vincoli sul posizionamento delle risorse sui nodi rispetto alla topologia del grafo
    - La localizzazione delle risorse è resa difficoltosa dalla mancanza di organizzazione della rete
  - L'aggiunta o la rimozione di nodi è un'operazione semplice e poco costosa
  - Obiettivo: gestire nodi con comportamento fortemente dinamico (tassi di join/leave elevati)
  - Esempi: Gnutella, FastTrack, eDonkey, …

Reti strutturate

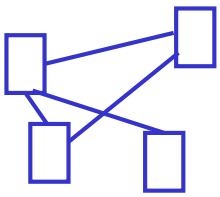
- Reti **strutturate**
  - La costruzione della rete overlay si basa su algoritmi deterministici
  - Vincoli sulla topologia del grafo e sul posizionamento delle risorse sui nodi del grafo
    - L'organizzazione della rete segue principi rigidi
  - Sistemi basati su Distributed Hash Table (DHT)
  - L'aggiunta o la rimozione di nodi è un'operazione costosa
  - Obiettivo: migliorare la localizzazione delle risorse
  - Esempi: Chord, Pastry, CAN, Kademlia, …

Routing in reti non strutturate

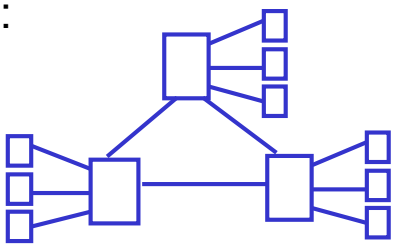
- Sistemi P2P decentralizzati ibridi: directory centralizzata



- Sistemi P2P decentralizzati puri: ricerca flood-based
  - Per ora, ci focalizziamo su questo tipo di reti non strutturate



- Sistemi P2P parzialmente centralizzati: directory semi-centralizzate e ricerca flood-based limitata

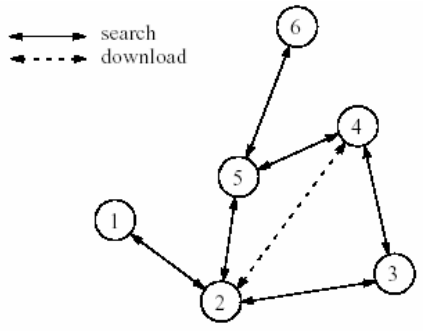


Ricerca flood-based

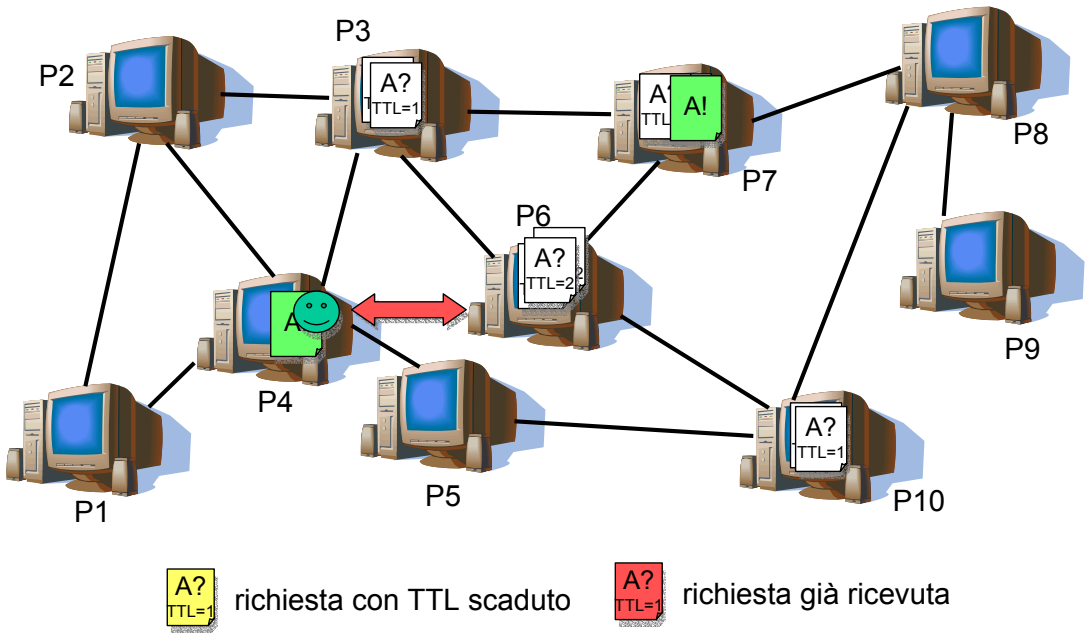
- Approccio completamente distribuito per localizzare le risorse
- L'invio delle query utilizza un meccanismo di **flooding** (inondazione)
- Ogni peer propaga (flood) la richiesta ai peer "vicini", che a loro volta inviano la richiesta ai loro "vicini" (escludendo il vicino da cui hanno ricevuto la richiesta)
  - Fino a che la richiesta è risolta, oppure viene raggiunto un massimo numero di passi (definito da Time To Live o TTL)
    - Ad ogni inoltro il TTL viene decrementato
    - Il TTL limita il "raggio della ricerca", evitando che il messaggio sia propagato all'infinito
  - ID univoco (detto GUID) assegnato al messaggio per evitare che venga nuovamente elaborato dai nodi da cui è stato già ricevuto (presenza di cammini cicli nei grafi)

Ricerca flood-based (2)

- L'invio delle risposte utilizza il backward routing
- La risposta positiva ad una query viene inoltrata a ritroso lungo lo stesso cammino seguito dalla query
- Possibile uso del GUID per individuare il backward path



Esempio di ricerca con flooding

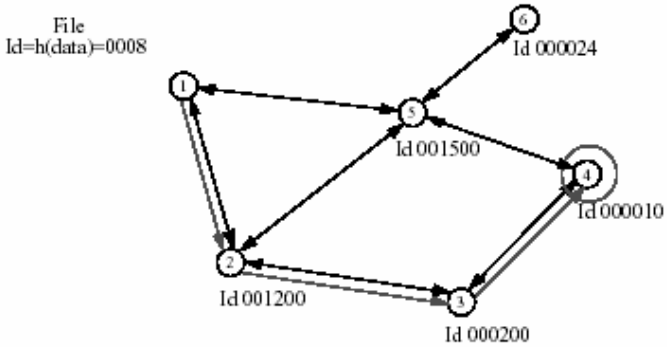


Problemi del flooding

- Crescita esponenziale del numero di messaggi
  - Possibilità di attacchi di tipo denial-of-service
  - Nodi black-hole in caso di congestione
  - Non è realistico esplorare tutta la rete
- Costo della ricerca
  - Le risposte dovrebbero avere tempi ragionevoli
  - Come determinare il raggio di ampiezza del flooding?
- Non è garantito che vengano interrogati (tutti) i nodi che posseggono la risorsa
- Traffico di query
  - Messaggi senza risultato occupano comunque banda
- Mancanza di una relazione tra topologia di rete virtuale e reale
  - A che distanza sono i peer “vicini”?

Routing in reti strutturate

- Sistemi basati su **Distributed Hash Table (DHT)**
- Ad ogni peer è assegnato un ID univoco in uno spazio di identificatori grande; ogni peer conosce un certo numero di peer
- Ad ogni risorsa condivisa viene assegnato un ID univoco (dallo stesso spazio di identificatori usato per i peer), definito applicando alla risorsa una funzione hash
- Per instradare la richiesta per una risorsa verso il nodo associato, occorre mappare univocamente l'ID della risorsa nell'ID di un nodo, basandosi su qualche metrica di distanza
  - Richiesta instradata verso il peer che ha l'ID più "vicino" a quello della risorsa
  - Possibile copia locale della risorsa ad ogni peer attraversato



Distributed Hash Table

- Astrazione distribuita della struttura dati *hash table*
- Una risorsa è rappresentata dalla coppia chiave-valore (*key* K, *value* V)
  - K identifica la risorsa (contenuta in V)
- API per l'accesso alla DHT (comune a molti sistemi basati su DHT)
  - put(K, V): inserimento di informazione (memorizza V in tutti i nodi responsabili per la risorsa identificata da K)
  - remove(K, V): cancella tutti i riferimenti a K e all'associato V
  - V = get(K): recupera V associato a K da uno dei nodi responsabili
- Ogni risorsa è identificata solo mediante il valore della chiave
  - Per cercare una risorsa occorre conoscere il valore esatto della chiave
  - Non sono possibili query complesse (ad es. range query, ricerche contenenti wildcard)

Distributed Hash Table (2)

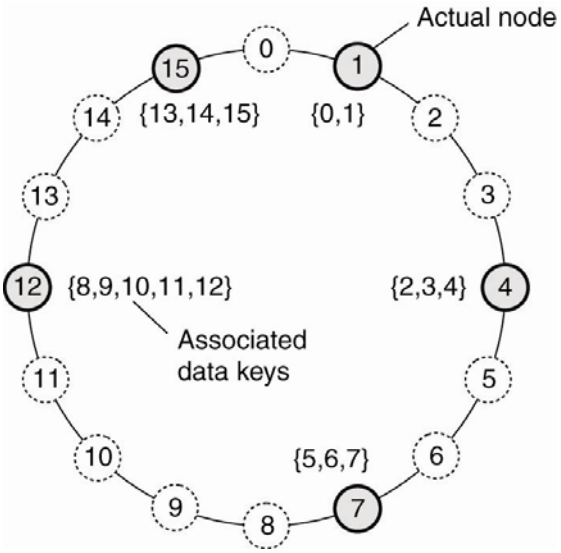
- Occorre mappare univocamente la chiave di una risorsa nell'identificatore di un nodo
  - Identificatore a *m* bit (*m*=128 o 160)
- Le risorse vengono mappate nello stesso spazio di indirizzamento dei nodi mediante una funzione hash (ad es. SHA-1)
- Ogni nodo è responsabile di una parte di risorse memorizzate nella DHT
- Ad ogni nodo viene assegnata una porzione contigua dello spazio di indirizzamento
  - Ogni nodo memorizza informazioni relative alle risorse mappate sulla propria porzione di indirizzamento
- Ogni chiave viene mappata su almeno un nodo della rete
  - Spesso si introduce una certa ridondanza

Distributed Hash Table (3)

- Le DHT operano in maniera distribuita con molti nodi
  - Elevata scalabilità
- Diverse soluzioni che specificano anche la modalità di routing delle ricerche e della memorizzazione
  - Più di 20 protocolli e prototipi per reti P2P strutturate, tra cui:
    - Chord (MIT)
    - Tapestry (Berkeley)
    - CAN (Berkeley)
    - Pastry (Rice Univ., Microsoft)
    - Kademlia (NY Univ.)
    - SkipNet/SkipGraph
    - Viceroy
    - Z-Ring
    - Chimera (UCSB)

Chord

- I nodi e le chiavi delle risorse sono mappati in un anello mediante una funzione hash
- Ogni nodo è responsabile delle chiavi poste tra sé e il nodo precedente nell'anello
  - La risorsa con chiave *k* è mappata nel nodo con il più piccolo identificatore *id* ≥ *k*
  - Questo nodo è detto *successore* della chiave k *succ(k)*
- La metrica di distanza si basa sulla differenza tra gli identificatori



http://www.pdos.lcs.mit.edu/chord/

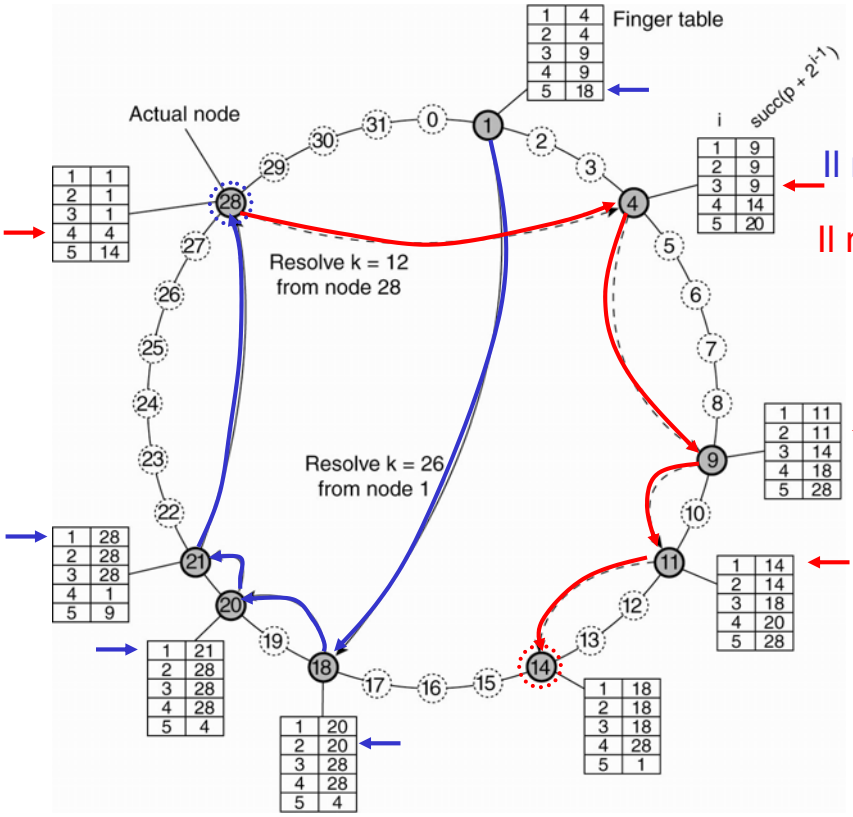
Finger table in Chord

- Finger table (FT)
  - E' la tabella di routing posseduta da ogni nodo
  - Se FT_p è la FT del nodo *p*, allora $FT_p[i] = succ(p + 2^{i-1})$
    - $succ(p+1)$, $succ(p+2)$, $succ(p+4)$, $succ(p+8)$, $succ(p+16)$, …
  - Dimensione della FT pari a log(*N*), con *N*=numero dei nodi
- Idea della finger table
  - Ogni nodo conosce "bene" le posizioni vicine ed ha un'idea approssimata delle posizioni più lontane
- Come risolvere la chiave *k* nell'identificatore di *succ(k)* a partire dal nodo *p* (algoritmo di routing):
  - Se *k* è nella zona di competenza di *p*, la ricerca termina
  - Altrimenti, *p* inoltra la richiesta a *q* con indice *j* in FT_p

$$q = FT_p[j] \le k \le FT_p[j+1]$$

- Caratteristiche
  - L'algoritmo raggiunge velocemente le vicinanze del punto cercato, per procedere poi con salti via via più piccoli
  - Costo di lookup: O(log(*N*))

Esempio di routing in Chord



Il nodo 1 risolve la chiave 26

Il nodo 28 risolve la chiave 12

Ingresso e uscita di nodi in Chord

- Al join nell'overlay network, il nodo *p*:
  - Contatta un nodo arbitrario e richiede la ricerca di *succ*(*p*+1)
  - Si inserisce nell'anello
  - Inizializza la sua FT
  - Aggiorna la FT del nodo che lo precede sull'anello
  - Trasferisce su se stesso le chiavi di cui è responsabile
- Alla leave dall'overlay network, il nodo *p*:
  - Trasferisce al suo successore le chiavi di cui è responsabile
  - Aggiorna il puntatore al nodo che lo precede del nodo che lo segue sull'anello
  - Aggiorna il puntatore al nodo che lo segue del nodo che lo precede sull'anello
- Costo di join/leave: O(log(N^2))
- Problemi Chord
  - Manca una nozione di prossimità fisica
  - Supporto costoso per ricerche senza matching esatto

Architetture ibride

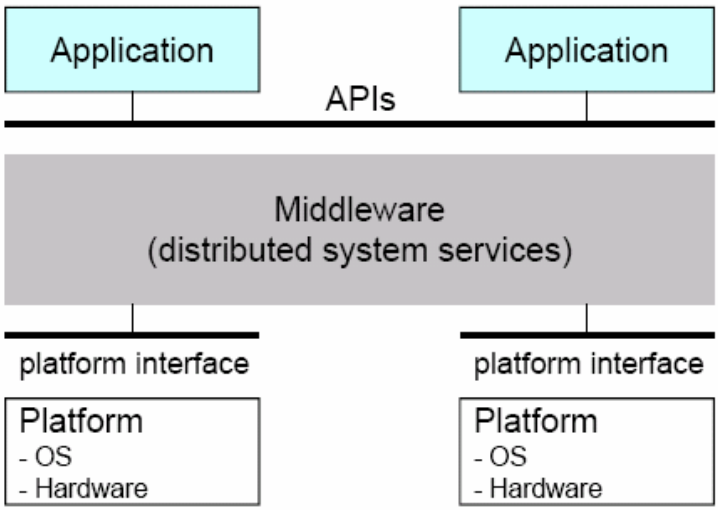
- Esamineremo in seguito diversi esempi di architetture ibride
  - In cui la classica soluzione client-server è combinata con un'architettura decentralizzata
- Ad esempio:
  - Sistemi P2P con superpeer
  - Sistemi edge-server nelle Content Delivery Network
  - Sistemi distribuiti collaborativi, ad esempio basati sul protocollo Bit Torrent

Middleware

- Per **middleware** si intende
  - una classe di tecnologie software che aiuta gli sviluppatori nel gestire la complessità e l'eterogeneità presenti nei SD
  - uno strato software "in mezzo"
    - sopra al sistema operativo, ma sotto le applicazioni
    - fornisce un'astrazione di programmazione distribuita (un modello computazionale uniforme)
    - per mascherare alcune eterogeneità degli elementi sottostanti (reti, hw, sistemi operativi, linguaggi di programmazione, ...)
  - un software di connessione che consiste in un insieme di servizi e/o di ambienti di sviluppo di applicazioni distribuite che permettono a più entità (processi, oggetti, ...), residenti su uno o più elaboratori, di interagire attraverso una rete di interconnessione *nonostante* differenze nei protocolli di comunicazione, architetture dei sistemi locali, sistemi operativi, …
- Alcuni esempi di middleware: RPC, Java RMI, CORBA, Java EE, .NET, Web services

Middleware (2)

- Un **servizio di middleware** è un servizio general-purpose che si colloca tra piattaforme e applicazioni (P. Bernstein, 1996)



Tipi di middleware

- Varie famiglie di tecnologie di middleware
- Middleware per oggetti distribuiti
  - Evoluzione di RPC: i componenti distribuiti sono considerati oggetti (con identità, interfaccia, incapsulamento)
  - Difficile la gestione della configurazione degli oggetti distribuiti
- Middleware message-oriented (MOM)
  - Basato sullo scambio asincrono di messaggi (e non su protocolli sincroni di richiesta/risposta)
  - Può offrire elevata flessibilità ed affidabilità

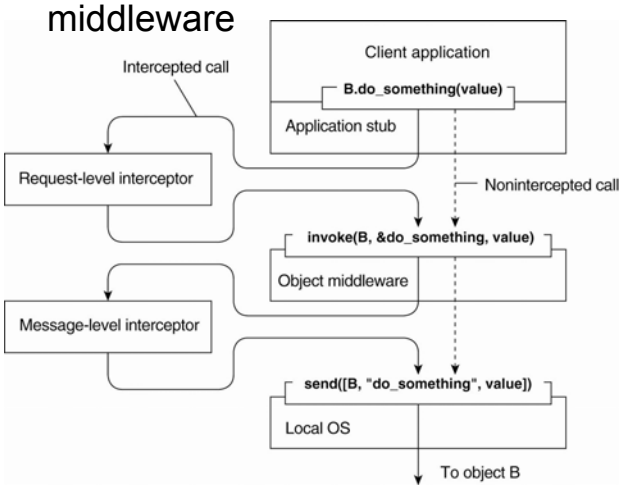
Tipi di middleware (2)

- Middleware per componenti
  - Evoluzione del middleware per oggetti distributi
  - Possibile sia la comunicazione sincrona che quella asincrona
  - I componenti vivono in contenitori (application server) in grado di gestire la configurazione e la distribuzione dei componenti, e fornire ad essi funzionalità di supporto
- Middleware orientato ai servizi
  - Enfasi sull'interoperabilità tra componenti eterogenei, sulla base di protocolli standard aperti ed universalmente accettati
  - Generalità dei meccanismi di comunicazione (sia sincroni che asincroni)
  - Flessibilità nell'organizzazione dei suoi elementi (servizi)

Architetture e middleware

- In molti casi lo strato di middleware segue uno specifico stile architetturale
  - Ad es. Message Oriented Middleware (MOM), Distributed Object Computing (DOC) Middleware
- Un approccio architetturale al middleware offre
  - Semplicità progettuale
  - Scarsa adattabilità e flessibilità
- E' preferibile un middleware che si possa adattare dinamicamente alla applicazione specifica ed all'ambiente

- Interceptor: costrutto software che interrompe il normale flusso di esecuzione e consente ad altro codice di essere eseguito
- Modifica la richiesta prima che sia inviata al componente
- E' uno strumento per adattare il middleware



Software adattivo

- Tre tecniche di base per ottenere un software adattivo
  - Separazione degli ambiti
  - Riflessione computazionale
  - Progettazione per componenti
- Separazione degli ambiti
  - Cercare di separare le funzionalità dalle extra funzionalità (affidabilità, prestazioni, sicurezza, …) e successivamente integrarle in una singola implementazione
  - Anche aspect-oriented software
  - Fino ad oggi esempi “giocattolo”
- Riflessione computazionale
  - Capacità di un programma di controllare se stesso e, se necessario, di adattare il proprio comportamento durante l’esecuzione
    - Relativamente al middleware: le politiche di azione sono espresse e visibili nel middleware stesso e si possono cambiare come componenti del sistema
  - Integrato in alcuni linguaggio di programmazione (ad es. Java), non in sistemi distribuiti a larga scala

Software adattivo (2)

- Progettazione per componenti
  - Organizzare un’applicazione distribuita in componenti che possono essere cambiati dinamicamente quando occorre
  - Supporto per il binding tardivo (o late composition): il sistema può essere configurato a tempo di esecuzione
    - Non solo staticamente in fase di progettazione
  - Complessa, anche per le interdipendenze tra componenti
- Occorre un software adattivo o un sistema adattivo?

Riferimento: P.K. McKinley et al., “Composing adaptive software”, *IEEE Computer*, 37(7):56-64, July 2004.

Sistemi distribuiti autonomici

- *Autonomic Computing*: introdotto come paradigma architetturale e tecnologico in grado di rispondere all'esigenza di gestire la crescente complessità ed eterogeneità dei sistemi IT mediante adattamenti automatici
- Metafora del sistema nervoso centrale, in grado di controllare alcune funzioni vitali (battito cardiaco, temperatura, …) mascherando la complessità all'uomo
- Un sistema autonomico dovrebbe essere in grado di gestire le proprie funzionalità autonomamente (ovvero senza o con limitato intervento umano), esprimendo capacità di adattamento (comportamenti *reattivi*) a dinamiche interne ed esterne, più in generale ai cambiamenti dell'ambiente

Sistemi distribuiti autonomici (2)

- Un sistema autonomico, a fronte di determinate politiche impostate dall'amministratore, dovrebbe possedere le seguenti caratteristiche:
- Autogestione
- Auto-configurazione
- Auto-guarizione
  - Garantire sopravvivenza ai guasti (ad es. malfunzionamenti hardware)
- Auto-ottimizzazione
  - Ottimizzare le proprie prestazioni
- Auto-protezione
  - Proteggersi da attacchi esterni
- Complessivamente: essere un sistema auto-* (o *self-**)

Riferimento: J.O. Kephart, D.M. Chess, "The vision of Autonomic Computing", *IEEE Computer*, 36(1), Jan. 2003.

Sistemi a controllo dei feedback

- In molti casi, i sistemi auto-* sono organizzati come sistemi a controllo dei feedback
  - Componente per la stima della metrica
    - Misura il comportamento del sistema
  - Componente di analisi dei feedback
    - Analizzare le misure e le confronta con dei valori di riferimento
    - E' il cuore del ciclo di controllo, poiché contiene gli algoritmi che decidono gli eventuali adattamenti
  - Meccanismi per influenzare il comportamento del sistema

Organizzazione *logica* di un sistema a controllo dei feedback (l'organizzazione *fisica* può essere molto diversa: i vari componenti possono essere distribuiti)