**SIMULAZIONE ESONERI SDCC**

6/12/2021

**DOMANDA 2)**

1. *Spiegare il meccanismo dell’anti-entropia in un protocollo di gossiping, discutendo anche quali vantaggi introduce rispetto al solo gossiping semplice.*

Il protocollo di gossiping si basa sull’idea del diffondere una informazione ad un numero ristretto di nodi, per poi propagarla. È PROBABILISTICO. Questa diffusione dell’informazione può avvenire tramite due meccanismi, i quali vengono spesso combinati.

Nel caso dell’anti-entropia, come suggerisce il nome, vogliamo ordine, che è rappresentato da nodi aventi stesse informazioni. Ogni nodo, regolarmente (quindi non per forza quando c’è un update), scambia il suo **stato** con un altro nodo scelto in modo casuale. Questo scambio può avvenire in tre modi:

Push (nodo P che ha scelto Q inoltra stato a Q), Pull (viceversa), Push-Pull, cioè uno scambio di stati bidirezionale.

I vantaggi rispetto al gossip semplice (usa solo push) risiedono nel fatto che:

* Lo scambio di stato avviene con maggior frequenza, mentre Pure Gossiping si basa su quando avviene l’update.
* Solo pure gossiping non mi dà certezza che raggiungo che raggiungerò tutti i nodi, e quindi solo introducendo l’anti entropia sono sicuro di toccarli tutti (perché avvenendo più volte, sceglierò più volte i nodi, e quindi è più probabile che li tocchi tutti).

1. *Spiegare se e come si può usare un algoritmo di gossiping per realizzare un meccanismo di failure detection per il rilevamento di guasti o arresti anomali dei nodi di un sistema distribuito a larga scala, discutendo anche quali vantaggi e svantaggi presenta rispetto ad una soluzione centralizzata in cui un singolo nodo invia messaggi di heartbeat a tutti gli altri nodi del sistema.*

Poiché gli algoritmi di gossiping veicolano il trasporto di informazioni, nulla vieta che esse possano interessare eventuali server down o arresti anomali. Un esempio di questo tipo di utilizzo è dato dal servizio di Amazon S3. PERIODICAMENTE POSSO USARE DEI PING PER COSTRUIRE IL PROPRIO STATO PER VEDERE I NODI CHE HANNO SUBITO FALLIMENTO. AD OGNI ROUND CONTATTO SOTTOINSIEME RANDOM DI NODI A CUI MANDO LA MIA INFORMAZIONE E MI FACCIO DIRE COSA LORO CONOSCONO, AUMENTANDO CONOSCENZA SULLO STATO DEI NODI. AD ALTRO ROUND FACCIO UGUALE, SCEGLIENDO SEMPRE CASUALMENTE ALTRO GRUPPO (O ANCHE SINGOLARMENTE) I NODI CON CUI COMUNICARE.

I vantaggi di una soluzione distribuita, rispetto heartbeat, consistono in:

* Velocità di propagazione dell’informazione, che può arrivare a O(log(N)) usando pure-gossiping ed anti-entropia.
* Maggior scalabilità, poiché in un sistema colmo di nodi, notificare a tutti di un’anomalia può essere un’operazione per il carico del sistema.
* Non abbiamo un SPOF, mentre nel caso centralizzato, se il nodo che comunica anomali avesse egli stesso anomalie, ciò comporterebbe una mancanza di informazioni.

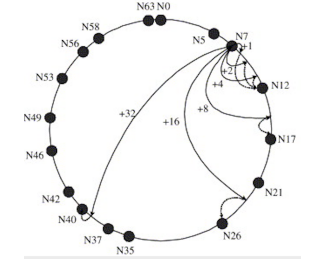
1. *Descrivere cosa è una hash table distribuita (DHT) e quali vantaggi e svantaggi presenta la ricerca di una risorsa basata su DHT rispetto alla ricerca di una risorsa basata su flooding.*

Una DHT è la versione distribuita di una Hash Table. La sua introduzione è dovuta ad una ricerca di scalabilità migliore riducendo i costi di lookup; infatti, mantenere una struttura può essere un’operazione costosa. SUPPORTA PUT E GET. Nella DHT si ha una Hash Table distribuita, cioè una tabella di tuple chiave-valore di dimensione M (cioè i bucket). È nei bucket, da 0 a M-1, che viene fatto il mapping. Tra i vantaggi troviamo sicuramente un mapping più veloce, ma che potrebbe incorrere in collisioni. Porta dietro a sé i vantaggi di un approccio distribuito (no SPOIF, no bottleneck, decentralizzato, ottimo per ricerche estese, configurazione iniziale minima). Tuttavia, la ricerca deve essere specifica sulla chiave (range più complessi), soffre guasti bizantini e comunque c’è sempre una piccola gestione tra nodi in entrata ed in uscita (O(log(N))).

Rispetto al flooding, che è non strutturata, ho sicuramente una ricerca più ordinata e veloce, meno soggetta a patch ciclici, che satura di meno la rete. Tuttavia, il flooding non richiede costi di gestione dei nodi (poiché chiedo a tutti, non mi interessa la collocazione). HA TEMPO O(N).

Chord interroga in O(log(N)); perché fa dei salti, non è né O(N) (esploro in senso orario anello) né O(1) cioè vado diretto

1. *Con riferimento al sistema Chord basato su DHT e la figura sottostante, spiegare cosa è la finger table ed indicare i valori della finger table del nodo N53. Cosa avviene quando il nodo N58 esce dalla rete?*



Un’idea di approccio per trovare facilmente una risorsa sarebbe che ogni nodo conoscesse tutto rispetto agli altri nodi. Questo però, in contesti distribuiti, si tradurrebbe in una tabella di grandi dimensioni.

L’approccio usato da Chord si basa sulla Finger Table: ogni nodo non conosce tutto di tutti, bensì solo i suoi vicini in modo preciso, **mentre più si allontana più l’informazione è “vaga”.**

Il costo di ricerca è logaritmico.

NB: poiché vado da N0 a N63, ho 64 indirizzi diversi, cioè 2^6, cioè ho m = 6 righe.

Esaminiamo il nodo N53:

Il suo successore è N56.

>> **FTp[i] = succ(p + 2^(i-1)) mod 2^(m)**

FTp = 53[1] = 53 + 20 = 53 -> 56 è il successore. Infatti, un nodo X gestisce risorse da un certo punto fino a X-1.

FTp = 53[2] = 53 + 21 = 55, è una risorsa, inoltre 55>53, devo mettere riferimento a nodo 56

FTp = 53[3] = 53 + 22 = 57 -> vado a nodo 58.

FTp = 53[4] = 53 + 23 = 61, idem (vado a 63)

FTp = 53[5] = 53 + 24 mod 64 = 5

FTp = 53[4] = 53 + 25 mod 64 = 21

Allora table è:

|  |  |
| --- | --- |
| 1 | 56 |
| 2 | 56 |
| 3 | 58 |
| 4 | 63 |
| 5 | 5 |
| 6 | 21 |

In pratica, quando devo cercare una chiave ‘X’ mi chiedo:

* Ce l’ho io? In questo caso manco guardo la tabella.
* Ce l’ha il mio successore? (Devo vedere se questa chiave è compresa tra me e nodo a riga 1. Se sì, chiedo a lui.
* Altrimenti chiedo al nodo in tabella più lontano ma tale che il valore del nodo sia <= della chiave della risorsa. (ad esempio, risorsa 59 la chiedo a 63, non a 58).

Se N58 esce dalla rete ciò che accade è:

* Le chiavi di cui è responsabile (56 e 57, NON 58) vanno al nodo successivo (N63, per forza devo rispettare le regole seguite fino ad ora.)
* Contatto il nodo successivo (N63) e gli comunico che il suo nuovo predecessore è N56.
* Contatto il nodo precedente (N56) e gli comunico che il suo nuovo successore è N63.

Se arrivasse N30 tra N26 e N35?

* N30 deve fare richiesta di lookup per trovare il successore. (succ (30+2^0) mod 2^m =35).
* Compila la sua FT che avrà sei righe.
* N30 informa N35 che il suo nuovo predecessore è N30 e non N26. N30 chiede a N35 chi fosse il suo predecessore. Contatta quindi N26 dicendogli che il suo nuovo successore è N30 e non N35.
* N30 prende le risorse “26,27,28,29,30”.
* Potrebbe esserci inconsistenza su alcuni nodi con FT non aggiornata, ogni delta di tempo devono ricalcolare. (algoritmo di stabilizzazione).

**DOMANDA 3**

1. *Descrivere quali sono i servizi offerti ai livelli IaaS e PaaS da un provider Cloud, quali vantaggi e svantaggi presentano per gli utenti, presentando anche un esempio di servizio Cloud per ciascuno dei due livelli.*

Possiamo distinguere tre tipi di servizi nel cloud, tra i quali andiamo ad esaminarne due:

nb: Questi tre livelli si differenziano in cosa l’utente può usare e gestire e cosa no.

* Infrastructure as a Service:

È il “livello” più basso, nella quale l’utente/utilizzatore utilizza, ma non controlla, Servizi di memorizzazione, networking, server e virtualizzazione. L’utente deve fare lo scaling.

Ciò che l’utente può gestire sono i S.O, middleware, dati e applicazioni.

Rispetto a PaaS ho più responsabilità, quindi sono più flessibile ma nello stesso momento devo anche gestire più livelli. Inoltre, sono io che devo gestire lo scaling. Esempio: AWS, EC2 servizio per instanziare VM.

Condivisione di risorse virtualizzate, risorse, virtualizzazione per istanze di VM.

Pro: massima flessibilità, ma più complesso.

* Platform as a Service:

Livello sovrastante all’ IaaS, nella quale posso controllare solamente lo sviluppo delle applicazioni e gestione dati. Non devo preoccuparmi dello scaling, e non ho la complessità dei livelli inferiori.

Un esempio di Paas è AWS Elastic Beanstalk per il deployment automatico di applicazioni web, nel quale Beanstalk si occupa del provisioning delle risorse, security groups, Auto scaling e servizio DNS fornito da Amazon.

Ambiente per sviluppo di applicazioni software. Devo usare API e linguaggi messi a disposizione della piattaforma.

**VEDERE ANCHE SAAS**

1. *Spiegare cosa è il pattern architetturale MAPE per la realizzazione di un sistema software auto adattativo e descrivere un algoritmo per l’elasticità orizzontale di un insieme di macchine virtuali.*

Quando parliamo di middleware, sarebbe ideale che questi si adattassero a runtime, cioè rispetto a sé stessi e all’ambiente. Questi tipi di middleware sono detti Middleware Self\*, volti all’ottimizzazione delle risorse, autoconfigurazione e identificazione di fault.

Per raggiungere queste caratteristiche si introduce la soluzione **MAPE**. È un’architettura/PATTERN di riferimento composta da:

* **Monitoring**: collezione di dati tramite sensori, quindi si limita alla raccolta dei dati. Ciò che possiamo definire è cosa prende, quando lo prende, o come lo prende.
* **Analyze**: Si analizzano i dati precedentemente ottenuti, cercando di capire se c’è necessità di un adattamento. Anche qui possiamo vedere quando analizzare, e se adottare un’analisi basata sul passato o sul futuro (facendo predizione).
* **Plan**: Determina quale azione compiere. Questa fase è la più variegata, perché possiamo far riferimento a teorie delle code, ML, euristiche etc...
* **Execute**: Applicazione del cambiamento scelto dal Plan.

L’applicazione di questa architettura può essere **orizzontale FLAT** o verticale Hierarchical.

Nel caso FLAT, ogni nodo esegue il ciclo MAPE e si coordina con altri nodi.

Non c’è nessuno che dirige quindi no POF, ma le analisi sono meno precise perché i dati sono locali e non globali. (Non ho modo di capire la bontà del dato degli altri).

**ALGORITMO ELASTICITA’ ORIZZONTALE**

La parte più “interessante è il PLAN: posso usare euristiche, ML o threshold. Il monitor deve monitorare stato di carico delle vm come uso della CPU. L’algoritmo chiede all’utente di configurare soglia minima e massima. Se supero soglia alta istanzio nuove VM (Quante? Posso sceglierlo).

Se uso medio < soglia di calcolo, posso ridurre (scale in) il numero di VM. Per evitare sbalzi, posso fare in modo che una macchina appena allocata non venga subito de-allocata, per evitare le oscillazioni.

AMAZON EC2 NE È UN ESEMPIO.

NB: questa parte della domanda, sull’algoritmo, non mi sta dicendo “Spiega precisamente chi fa monitoring, chi plan etc..” dice solamente un algoritmo per gestire l’elasticità orizzontale (prelievo/rilascio di risorse), ma se parlo di algoritmo sto parlando dell’azione da compiere, cioè sto parlando della fase di PLAN, ma non è che mi sta dicendo “APPLICA ALGORITMO BASATO SU MAPE”.

1. *Descrivere l’architettura di Kafka e discutere quali soluzioni sono state adottate per offrire un throughput elevato nelle operazioni di publish e consume.*

Kafka è architettura distribuita di tipo publish subscribe composta da molteplici server chiamati broker. Questi broker gestiscono memorizzazione msg prodotti dai producer e diretti ai consumer all’interno di un log partizionato.

Il terzo attore è “kafka cluster” che gestisce i log partizionati.

I messaggi vengono identificati in categorie “topic”; ogni topic ha un solo LOG append-only, ogni topic è diviso in partizioni (unità di parallelismo del topic), le partizioni sono replicate sui diversi broker, per aumentare la tolleranza.

Un producer pubblica record su partizione del topic (round robin o altro), consumer lo legge). Ogni record ha associato un offset della partizione. Ogni partizione ha un solo leader e 0 o più follower. Solo leader gestisce le richieste di lettura e scrittura. Quando scrivo o leggo, lo faccio sulla partizione del LEADER, non dei follower.

Per aumentare la tolleranza si fa in modo che, quando c’è una write, il messaggio non è disponibile ai consumer finché i follower non lo hanno a loro trascritto e confermato con ACK.

Inoltre, quando consumer legge messaggio, salva il suo offset in un topic speciale chiamato \_\_consumer\_offset, un topic instanziato automaticamente da Kafka, in cui i consumer salvato i valori dell’offset. Utile per tolleranza ai guasti, poiché se perdo l’offset in memory, posso recuperarlo da questo topic.

Il retrieve da parte del consumer può essere fatto dall’inizio, oppure da un certo offset.

Aumento ancora di più il throughput perché ho **Consumer group:**

insieme di consumer che cooperano per consumare dati da topic, identificati da un **ID di gruppo**. I partizionamenti sono divisi tra questi consumer nel gruppo per bilanciare il carico. Ogni messaggio è consegnato solo *ad un elemento del gruppo*. Ogni gruppo mantiene gli offset delle partizioni.

L'idea fondamentale è vedere più consumer come unica entità. Se volessi due consumer che **leggono stesso messaggio**, i due consumer devono appartenere a gruppi diversi! (poiché messaggio consegnato ad un solo elemento del gruppo!). Non possono leggere da stessa partizione.

Semantiche di comunicazione supportate da Kafka:  
-> **At-least-once** (default).  
-> **Exactly-once** (non è però un’Exactly-once completa e in più utilizza un protocollo oneroso – il 2 phase commit - per la gestione delle transazioni).  
-> L’utente può anche implementare la sematica **At-most-once** disabilitando le ritrasmissioni sul producer e facendo effettuare il commit al consumer prima di processare un messaggio.

Perché throughput elevato? Ciò ci dà il vantaggio in fase di publish: più producer possono pubblicare su partizioni diverse di stesso topic. In termini di consumer ho il vantaggio di letture contemporanee. Inoltre, uso di consumer group aumentano il throughput, accedono a partizioni differenti. Tra gli svantaggi abbiamo la mancanza di tolleranza ai guasti, risolvibile con la replicazione guidata da “leader-follower”. C’è una copia primaria sulle quali avvengono W/R, e poi le copie che vengono mantenute aggiornate.

1. *Considerando un sistema distribuito che offre un’applicazione per l’adattamento di video, spiegare perché l’uso di una coda di messaggi nell’architettura del sistema può migliorare la scalabilità e la tolleranza ai guasti dell’applicazione.*

Slide 18 del pacco 6. Servizio SQS di Amazon, esempio.

Disaccoppiamento front/back end grazie alla coda dei msg. Uso due code bidirezionali, una per inviare foto dalla parte frontend a backend, e una che esegue il percorso opposto. (sarebbero richieste e risposta). Poiché foto/video pesanti, nei msg metto URL delle foto memorizzate su S3. Posso replicare su EC2 le istanze del processamento, per replicare scalabilità e aumentare tolleranza ai guasti senza influenzare il funzionamento. Se visibilità è bene settata (ad esempio uso SQS) quando il msg torna visibile viene preso da un’altra copia di processamento, l’importante è prendere il video.

**Domanda 4**

1. *Si spieghi il ruolo di client stub, server stub e service registry nelle chiamate di procedura remota, spiegando le possibili differenze in questi ruoli nel caso di Sun RPC, Java RMI e Go.*

Nelle RPC abbiamo un client che effettua la chiamata sito su una macchina diversa rispetto al server che deve rispondere a tale chiamata. Quindi abbiamo aspetti non locali da gestire, e questo porta all’introduzione degli stub o proxies. Il client stub presenta l’interfaccia del servizio, il client vero e proprio lo vede localmente ed effettua la chiamata locale come se il server fosse sulla stessa macchina. Lo stub client effettua una operazione di marshaling (impacchetta i dati in formato comune) spedendolo al server stub (su altra macchina!), ed eseguirà unmarshaling dei dati, in modo che siano comprensibili al server. Il server, quindi, eseguirà la chiamata localmente rispetto alla chiamata del server stub, e il risultato seguirà il procedimento inverso fino a tornare al client.

In SunRPC, il client trova i servizi forniti dal server mediante Portmapper, il cui hostname è noto.

In JavaRMI, il client sfrutta l’RMI-registry, entità intermedia, cioè un binder, per registrare i servizi.

Inoltre, il server stub prende il nome di skeleton. Si eseguono operazioni di serializzazione e deserializzazione.

In Go, non ho un service registry dedicato, bensì mi appoggio a funzioni Register e Listen.

client stub:

rpc fa marshaling, rmi fa deserializzazione, grpc usa altro.

go ha supporto al livello di linguaggio, rpc e rmi no.

service registry:

in sun rpc c'è portmapper: numero di porta da contattare, client stub conosce numero di porta su cui esposto il portmapper, devo sapere dove si trova.

in java rmi broker.

in go: non c'è.

1. Si spieghino i ruoli di un linguaggio di definizione dell'interfaccia (IDL) e del marshalling nelle chiamate di procedure remote. Quali sono le soluzioni adottate in Sun RPC, Java RMI e Go a questo riguardo?

Un obiettivo che ci si pone nello sviluppo delle RPC è quello che esse risultino il più trasparenti possibili rispetto a delle chiamate a procedura locali. Ovvero vorrei limare le differenze tra una chiamata remota ed una locale, senza preoccuparmi di astrazioni o composizione di altri servizi.  
Per realizzare ciò si introducono le interface Definition Language, le quali definiscono dei “contratti” per le procedure, ovvero descrivono le chiamate remote. Tramite le IDL, infatti, è possibile generare automaticamente stub ed operazioni di marshal. Il marshal permette la conversione di dati in un formato interpretabile da client e server stub, evitando quindi al programmatore l’onere di realizzare anche queste componenti.

* In SunRPC l’IDL usato è XDR, external data representation, insieme a IDL compiler per realizzare client e server stub.
* In Java RMI, Java stesso è un IDL, quindi possiamo dire che “Non serve un IDL, ci pensa java”.
* In gRPC, si usano i “protocol buffers” sia come IDL, per definire interfaccia, sia per il formato di mesaggi. I protocol buffers serializzano le strutture dati.

idl consente di creare automaticamente stub (perchè non ho supporto nativo).

sun usa XDR, RMI usa direttamente Java, gRPC usa protocol buffer, che crea stub server per grpc.

1. *Descrivere i meccanismi usati per garantire la semantica di comunicazione at-most-once e quali meccanismi aggiuntivi sono richiesti per realizzare la semantica exactly-once*

Introduco tre meccanismi alla base delle semantinche:

* request retry: client reinvia msg finchè non riceve risposta, oppure è sicuro che server sia crashato.
* Duplicate filtering: il server riconosce se la richiesta è duplicata. (uso log)
* Result retransmit: il server, riconosciuto un messaggio duplicato, non lo ricalcola.

At-most-once: operazione eseguita al max una volta. Sicuramente uso RR1 perché server potrebbe non rispondere, e mi servono DF e RR2 perché devo riconoscere messaggi doppi e non rieseguire le operazioni (soprattutto per operazioni non idem-potenti).

Exactly-once: operazione eseguita esattamente una volta. Richiedi i tre meccanismi, ma si basa sul concetto di all-or-nothing, cioè o tutto va bene o al minimo errore non completo.

Questa richiede meccanismi in più:

* Replicazione trasparente server: non so distinguere server reale dalla copia.
* WAL: log in cui ho tutte le operazioni eseguite e da cui ripartire.
* Recovery: mantengo snapshot distribuito per ritornare ad uno stato sicuro.

NB: At least once usa solo RR1.

1. *Descrivere quali sono le caratteristiche di gRPC che lo differenziano rispetto alle altre soluzioni per RPC esaminate a lezione.*

La caratteristica principale di gRPC è il fatto di poter permettere la creazione di applicazioni distribuite composte da servizi indipendenti ed autonomi, anche in linguaggi diversi. Si basa su http/2: lavoro con oggetti http.

gRPC usa Protocol buffers sia come IDL per definire le interfacce sia che come formato di messaggi usato (fa da tramite per comunicazione stub-server). Non c’è il concetto di RMI-Registry o PortMapper, bensì funzioni “Register” e “Listen”.

gRPC pensa alle procedure remote come oggetti http, e trasmette “stream di dati” bidirezionali, trasportanti 1 o più messaggi, cioè sequenze complete di frame, che è la più piccola unità di comunicazione.

gRPc permette client server in linguaggi diversi, uso stream (solo client, solo server, o entrambi). usa http/2, usa i protocol buffer. maggiore compattezza.

4/12/2019

**Domanda 2 (punti 6)**

1. *Si descriva un algoritmo di gossiping a scelta tra quelli esaminati a lezione.*

Un algoritmo di gossiping può essere rappresentato come segue:

* Un nodo P sceglie casualmente un altro nodo vicino Q, cioè ho selezionato un Peer.
* Il nodo P seleziona alcune informazioni che ritiene importanti da condividere, esse possono dipendere dall’implementazione/contesto in cui si usa l’algoritmo di gossiping.
* Il nodo P invia queste informazioni al nodo Q, che le riceve.
* Il nodo Q sceglie a sua volta alcune entries dalla sua cache locale da inviare a P, e le invia. (ipotizzando strategia push-pull, la più efficace).
* Entrambi, dopo lo scambio di informazioni, selezionano le entries da tenere memorizzate, e rimuovendo eventuali duplicati.
* Entrambi processano i nuovi dati acquisiti.

Queste operazioni vengono eseguite in ogni intervallo delta T.

GOSSIPING PURO NON E' ALGORITMO A SCELTA, ma chiede multicast o blind conter rumor mongering.

Immagine che contiene testo

Descrizione generata automaticamenteImmagine che contiene testo

Descrizione generata automaticamente

1. *Quali vantaggi presenta il gossiping rispetto al flooding considerando la diffusione di informazioni, ad es. per il monitoraggio di un sistema distribuito a larga scala? E quali svantaggi?*

In ottica di diffusione delle informazioni, un approccio con flooding presenta, a parità di “round/cicli”, come vantaggio una saturazione minore della rete, cioè invio meno messaggi e ho meno ridondanza, ma risulta essere più lento, cioè si raggiungono meno nodi.

1. *Si spieghi se e come si può usare una distributed hash table per realizzare il routing di eventi ai subscriber in un sistema publish/subscribe di tipo topic-based, discutendone anche i possibili vantaggi e svantaggi. Un sistema così realizzato risulta scalabile rispetto al numero dei publisher e dei subscriber e perché?*

È possibile usare una DHT per applicazioni di tipo topic based. L’assegnazione del topic è infatti statica, e nulla vieta di usare una funzione Hash per mappare i vari topic nei bucket (qui rappresentati dai worker). Quindi il routing viene eseguito su eventi e la ricerca risulta più veloce, oltre che maggiormente scalabile: essendo la table distribuita, i publisher e i subscriber risolvono la funzione hash del topic e possono scrivere o leggere il messaggio.

Uno svantaggio dato da questa implementazione risulta nel carico dei bucket mantenenti i topic, essendo infatti l’hash “selettiva”, possiamo avere bucket più comuni e quindi saturi di altri.

ogni nodo gestisce dei topic, con topic = chiave.

qui è publish/subcribe, vorrei 1-to-many, 1 msg arriva a molteplici subscriber. Posso usare multicast, con dht come faccio? uso albero di multicast, insieme a DHT, in modo che i subscriber si iscrivano ai topic.

Scalabile perchè uso realiazzazione distribuita basata su dht, che per definizione è scalabile.

CONTRO:

se aggiungo nuovi publisher o subscriber devo riadattare la tecnologia, cioè riadattare l'albero di multicast. Pro e contro sono quelli delle reti strutturate.

**Domanda 3 (punti 7)**

1. *Si descriva lo stack dei servizi Cloud, presentando un esempio di servizio Cloud per ciascun livello. (pagina 13 del pacco 2)*

Lo stack dei servizi cloud presenta tre livelli:

* Infrastructure as a Service: è il livello più basso. Qui la computazione, memorizzazione e risorse di rete sono fornite come servizi (li uso ma non li controllo), mentre posso gestire applicazioni, dati, runtine, middleware e OS. L’utente deve gestire lo scaling. Un esempio è rappresentato dagli AWS EC2 e S3.
* Platform as a Service: Livello più alto rispetto Iaas, l’utente può sviluppare e gestire applicazioni e dati, senza preoccuparsi dei livelli sottostanti. Il provider effettua lo scaling. Un esempio è AWS Beanstalk, che configura le risorse quando eseguo il run del codice.
* Software as a Service: è livello più alto di tutti, l’utente non gestisce nulla, può solo usare i servizi offerti. Esempi come Drive o Dropbox sono utilizzi comuni degli utenti in internet.

1. *Perché il modello di deployment del cloud ibrido è il più diffuso in ambito aziendale?*

Il cloud ibrido risulta ampiamente diffuso poiché mette insieme i vantaggi del cloud pubblico e privato: permette di differenziare la privacy (da cloud privato), posso passare da private cloud a public cloud in caso di disaster recovery, o comunque gestire il carico quando il cloud privato è insufficiente, e avendo provider diversi evito di “dipendere” da un solo provider.

1. *Si definisca il concetto di elasticità per le risorse di un sistema distribuito, discutendo anche i vantaggi che introduce per l’utente e per il provider di un servizio elastico. Con riferimento al ciclo MAPE, si presenti una possibile architettura per gestire l’elasticità di un insieme di macchine virtuali.*

Quando ho a che fare con carichi variabili, vorrei essere in grado di gestire picchi di carico, ma anche non “sprecare” risorse quando i picchi sono minimi. Lo studio di questi aspetti è alla base dell’elasticità, ovvero la capacità di “scalare” le risorse in modo da sfruttarle al meglio, senza sprechi, in modo automatico. Scelte implementative che permettono di modellare bene l’elasticità consentono al provider di usare sempre un numero di risorse adeguato, senza sprechi, e senza evitare abbandono degli utenti (poiché quando un applicativo risulta non raggiungibile si perdono i guadagni derivanti dall’accesso) e consentono all’utente una fruizione migliore dei contenuti (se resisto ai picchi, avrò pochi rallentamenti, e l’utente userà l’applicazione).

In ambito MAPE, posso utilizzare una architettura gerarchica “regional pattern”, ovvero ogni macchina virtuale scambia le informazioni di Monitoring, Analyze e Execute con un master che compie il “Plan”, ovvero sceglie la strategia da adottare (aumentare istanze, mantenerle etc).

SCALABILITA’: SUPPORTA AGGIUNTA DI NODI AL SISTEMA, MA NODO DEVE ESSERE AGGIUNTO.

CON ELASTICITA’ LO AGGIUNGE DA SOLO. ELASTICITA’ SE SISTEMA AUTONOMICO, SE SCALABILE (CIOE POTENZIALMENTE LO FA, MA NON LO FA AUTOMATICAMENTE) ALLORA NON È ELASTICO.

KUBERNETES automaticamente istanza container di quel tipo.

VANTAGGI X L’UTENTE DELL’ELASTICITA’: non se ne preoccupa lui, manco deve gestirle.

1. *Si presenti un algoritmo per l’elasticità di un insieme di macchine virtuali*

Un algoritmo per l’elasticità potrebbe essere il seguente:

Possiamo basarci su delle euristiche basate su soglie, in cui definiamo una size minima (non posso usare meno di questa size), una size massima (non posso usare più di questa), una capacità desiderata (vorrei mantenermi su questa size mediamente). Successivamente possiamo definire le nostre soglie, ad esempio basate sul carico della CPU. Imponiamo quindi che, dato il carico della CPU medio in un certo deltaT, se questo eccede una certa % allora si aumenta di istanza, altrimenti si mantiene. Se soglia è minore di una certa % allora si decrementa. Posso fissare tempi diversi per le due operazioni per fare in modo che non ci sia un’entrata/uscita di istanze troppo variabile.

**Domanda 4 (punti 6)**

1. *Si definisca il disaccoppiamento spaziale e temporale, si spieghi se una coda di messaggi supporta entrambi, motivando la risposta, e si motivi la scelta frequente di adottare una coda di messaggi per la comunicazione in un’applicazione a microservizi.*Disaccoppiamento spaziale: la capacità dei componenti di non necessitare la conoscenza degli altri componenti per comunicare e cooperare, cioè sono anonimi.

Disaccoppiamento temporale: i componenti che interagiscono tra loro non necessitano di essere presenti allo stesso momento quando comunicano.

In una coda di messaggi (cioè il pattern) ho solamente il disaccoppiamento temporale, perché la coda di messaggi è un buffer persistente in cui memorizzare la richiesta.

Non ho necessariamente disaccoppiamento spaziale, perché in una coda ho svariati messaggi destinati a vari consumer, e devo indentificarli.

Se invece parlo di MOM (es Kakfa basato a topic) allora SI.

**In una comunicazione orientata ai messaggi (MOM), come Kafka, vengono supportati entrambi:**

ho disaccoppiamento spaziale poiché, avendo a disposizione queue, il compito del producer è di inviare li messaggio, che poi nella coda verrà processato per arrivare al consumer. Sempre perché c’è la coda, che realizza la persistenza, non è richiesto essere presenti.

CODA DI MESSAGGI PERSISTENTE, DISACCOPPIAMENTO TEMPORALE E SPAZIALE.

1. *Con riferimento al servizio Cloud Amazon SQS, quale è la semantica di delivery supportata e quali vantaggi introduce rispetto alla semantica di delivery at-least-once?*

Amazon SQS è Amazon Simple Queue Service, un esempio di Message Oriented Middleware basato su cloud. I messaggi all’interno delle code hanno durata limitata (configurabile), vengono cancellati solo se sono stati ricevuti correttamente. Semantica di comunicazione può essere timeout based, se messaggio non viene ricevuto entro lo scadere del time-out di visibilità, il messaggio torna ad essere visibile. Rispetto alla at-least-once, i msg vengono cancellati se ricevuti correttamente, altrimenti tornano ad essere visibile. Non rinvio quindi il messaggio, perché è sempre lo stesso che torna ad essere visibile.

**Domanda 5 (punti 7)**

1. *Si spieghi il ruolo di client stub, server stub e service registry nelle chiamate di procedura remota, spiegando le possibili differenze in questi ruoli nel caso di Sun RPC, Java RMI e Go.*

Nelle RPC abbiamo un client che effettua la chiamata sito su una macchina diversa rispetto al server che deve rispondere a tale chiamata. Quindi abbiamo aspetti non locali da gestire, e questo porta all’introduzione degli stub o proxies. Il client stub presenta l’interfaccia del servizio, il client vero e proprio lo vede localmente ed effettua la chiamata locale come se il server fosse sulla stessa macchina. Lo stub client effettua una operazione di marshaling (impacchetta i dati in formato comune) spedendolo al server stub (su altra macchina!), ed eseguirà unmarshaling dei dati, in modo che siano comprensibili al server. Il server, quindi, eseguirà la chiamata localmente rispetto alla chiamata del server stub, e il risultato seguirà il procedimento inverso fino a tornare al client.

In SunRPC, il client trova i servizi forniti dal server mediante Portmapper, il cui hostname è noto.

In JavaRMI, il client sfrutta l’RMI-registry, entità intermedia, cioè un binder, per registrare i servizi.

Inoltre, il server stub prende il nome di skeleton. Si eseguono operazioni di serializzazione e deserializzazione.

In Go, non ho un service registry dedicato, bensì mi appoggio a funzioni Register e Listen.

1. *Si spieghino i ruoli di un linguaggio di definizione dell'interfaccia (IDL) e del marshalling nelle chiamate di procedure remote. Quale è la soluzione adottata in Java RMI e perché differisce da Sun RPC?*

Un obiettivo che ci si pone nello sviluppo delle RPC è quello che esse risultino il più trasparenti possibili rispetto a delle chiamate a procedura locali. Ovvero vorrei limare le differenza tra una chiamata remota ed una locale, senza preoccuparmi di astrazioni o composizione di altri servizi.  
Per realizzare ciò si introducono le interface Defenition Language, le quali definiscono dei “contratti” per le procedure, ovvero descrivono le chiamate remote. Tramite le IDL, infatti, è possibile generare automaticamente stub ed operazioni di marshal. Il marshal permette la conversione di dati in un formato interpretabile da client e server stub, evitando quindi al programmatore l’onere di realizzare anche queste componenti.

* In SunRPC l’IDL usato è XDR, external data representation, insieme a IDL compiler per realizzare client e server stub.
* In Java RMI, Java stesso è un IDL, quindi possiamo dire che “Non serve un IDL, ci pensa java”.
* In gRPC, si usano i “protocol buffers” sia come IDL, per definire interfaccia, sia per il formato di messaggi. I protocol buffers serializzano le strutture dati.

1. **Perché il meccanismo di garbage collection distribuito usato in Java RMI è basato su lease?**

Si usano dei contatori di riferimenti, solo quando scendo a 0 dealloco. Questo meccanismo è detto “leaed-based garbage collection”.

Il client stub manda periodicamente dei msg “dirty” (sto usando tale risorsa), e manda msg “clean” quando non lo usa più. Se stub client non invia nulla, assumo che l’oggetto non venga più usato.

Uso questo meccanismo poiché non posso semplicemente togliere una risorsa, ma devo vedere chi la possiede e chi la sta usando. Richiede coordinazione fra client e server, ho quindi limiti nella scalabilità. Pagina 50 slide 4. Opera il client perchè se lo facesse il server dovrebbe fare troppe cose.

1. **Quali meccanismi devono essere usati dalle chiamate a procedura a remota per garantire una semantica di comunicazione at-most-once? Quale semantica di comunicazione supportano le diverse implementazioni di RPC esaminate a lezione?**

Introduciamo i tre meccanismi:

* Request Retry RR1: il client, dopo un deltaT in cui non riceve risposte, reinvia il msg.
* Duplicate filtering: il server è in grado di identificare le richieste già pervenute, usando un Id appartenente alla richiesta.
* Result retransmit RR2: il server, identificata una richiesta di operazione già eseguita, non la ri-esegue, ma fornisce quella già calcolata.

At most once le usa tutte e 3. RR1 serve per avere almeno una risposta (se client non reinvia msg non avrò mai risposta), le altre 2 evitano di rieseguire operazioni e quindi ri eseguire più di una volta. Non garantiscono exactly once.

SUNRPC supporta at least once DEFAULT (sincrona bloccante) e at most once(asincrona, disponibile solo su alcune estensioni)

Java RMI è sincrona -> at most once.

RPC in GO usa semantica at most once.

25/11/2018

**Domanda 2)**

1. **Si descrivano le caratteristiche dei protocolli di gossiping (o epidemici).**

Pacco di slide 6.

Il gossiping viene utilizzato, a livello di Multicast a livello applicativo, per realizzare un overlay network non struttato.

Il gossiping adotta un approccio probabilistico, basato sull’idea che, una informazione, condivisa con un numero limitato di nodi vicini scelti casualmente, possa propagarsi rapidamente.

Gli algoritmi di gossiping sono semplici, non centralizzati, scalabili (sia con pochi o tanti nodi io comunico la mia info a un numero limitato di nodi) e robusto (poiché si ha sempre ridondanza del messaggio). Il gossiping presenta due operazioni fondamentale, combinate per ottenere migliori prestazioni:

* Pure gossiping: quando P ha nuova informazione, seleziona randomicamente un insieme di vicini comunicando tale informazione. È tuttavia dimostrabile che non basta solo questa operazione per raggiungere tutti i nodi, perché, essendo probabilistico, c’è una probabilità 1/k che un nodo non propaghi l’update (magari lo ha già visto). Per questo si introduce la seconda operazione:
* Anti-entropia: periodicamente (quindi non coincide per forza con un update), un nodo P seleziona seleziona nodo Q (sempre random) e scambia i propri updates.

(P può mandarli a Q; riceverli da Q, o scambiarli bidirezionalmente). Quest’ultima è la più veloce, ci mette O(log(N)) per disseminare l’update.

1. **Si spieghi se e come si può usare un protocollo di gossiping per realizzare il routing in un sistema middleware di tipo publish/subscribe topic-based e quali vantaggi e svantaggi presenta rispetto all’uso di una rete overlay strutturata.**

P.49 slide 6.

È possibile applicare il gossiping per realizzare routing in un contesto topic based.

Quando un nodo ha un nuovo update, appartenente ad un topic “T”, possiamo “condizionare” la scelta dei nodi vicini a quelli che presentano una iscrizione al topic “T”, scegliendone comunque un sottoinsieme randomico. Il resto dell’applicazione non prevede altre modifiche agli algoritmi già visti. L’unica differenza sta nel restringere il sottoinsieme di nodi considerabili per l’update.

Rispetto ad overlay strutturata abbiamo:

+ : ho meno “controlli” sulla coerenza della struttura dell’overlay, cioè non ho una forma da mantenere. (nella strutturata potrei usare albero).

* : Ho più probabilità di inondare la rete con un elevato numero di messaggi rispetto ad approccio strutturato.

1. **Con riferimento ad un sistema a code di messaggi ed all’operazione di delivery di un messaggio, si spieghino le proprietà della semantica di comunicazione di tipo at-least-once e di quella timeout-based e quali sono i meccanismi utilizzati per realizzarle.**

Pacco 6 di slide.

In MOM abbiamo le seguenti semantiche di delivery:

* At-least-once:

Voglio che un messaggio sia ricevuto ALMENO una volta.

Per fare ciò, sender usa ACK, inviato dopo l’inoltro del msg al receiver.  
Se receiver non risponde, allora dopo un deltaT rimando msg.

Così quello che succede è: io reinvio finchè non ricevo ack di conferma, e receiver potrebbe riceverlo piu di una volta (magari manda ack di conferma ma ci sono problemi di rete o crash).

* Timeout based delivery:

Cancello il messaggio se è già stato ricevuto almeno una volta.

Producer invia msg VISIBILE in coda, receiver lo consuma, il msg diventa INVISIBILE per un tot di tempo. Se passa deltaT senza risposta ack, msg ritorna visibile per i consumer (anche gli altri).

transaction based replica il fatto della transazione, ma non garantisce in tutte le condizioni le exactly one, gli manca la replicazione trasparente del server.

Domanda 5 (punti 7)

1. **Si descrivano le caratteristiche dei sistemi P2P basati su hash table distribuite.**

Slide 3.

Nei sistemi decentralizzati il lookup delle informazioni ha un ruolo fondamentale.

Nei sistemi decentralizzati strutturati si cerca un compromesso tra i tempi di lookup e la mole di dati memorizzati. (infatti: se io sapessi tutto di tutti lookup semplice ma troppi dati da gestire, se non so nulla di nessuno lookup lento). Nelle reti p2p l’idea è che ogni peer è responsabile di alcune risorse, e conosce alcuni nodi dell’overlay. Risorse e nodi sono identificati tramite ID, quindi tramite DHT stiamo creando una ricerca di coppie chiave-valore.

Ogni riga della tabella è un bucket (peer), e con HT mappo la chiave per cercare il peer responsabile di una risorsa. In DHT questi bucket sono distribuiti.

Abbiamo quindi <key k, value v> distribuite tra peer. K identifica valore V, K corrisponde a GUID , che è funzione hash fatta sui metadati di v.

Una applicazione distribuita fa Hash(metadati) = Guid = K e trova la coppia chiave valore PRENDENDOLA DAL NODO CHE LA POSSIEDE.

Caratteristiche:

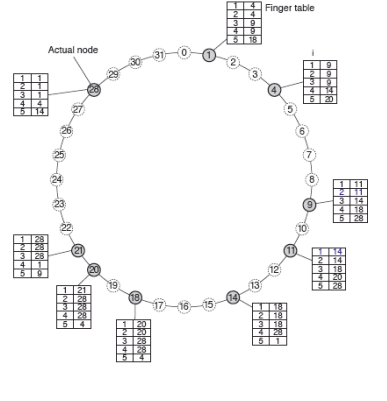
* Ricerca efficiente O(1)
* La funzione Hash deve ridurre al massimo le collisioni.
* Nel redesign della DHT, dove risorse e nodi hanno stesso GUID, dato K si mappa nel GUID piu vicino a k.
* Ricerca precisa, no query complesse.
* Altamente scalabili.

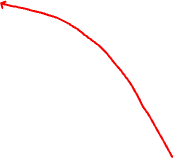
1. **Quali vantaggi e svantaggi presentano rispetto ai sistemi P2P basati su reti non strutturate?**

Rispetto a P2P non strutturate hanno come vantaggio una ricerca più veloce (O(1)), e la rete non viene quindi inondata di messaggi, e sicuramente non ho cicli infiniti.

Come svantaggi ho comunque una “struttura” da mantenere, quindi join/leave sono più complessi, il flooding di messaggi nelle reti non strutturare può portare ad una fault tolerance più efficiente.

1. **Con riferimento al sistema Chord basato su DHT raffigurato nella pagina seguente, si spieghi il significato dei valori della finger table del nodo 11 e si spieghi come viene effettuata da parte del nodo 11 la ricerca per la chiave 9.**

****



La finger table ha 5 righe, quind m = 5 e le varie operazioni sono in modulo 2^5 = 32

Il nodo 11 gestisce risorsa 10 ed 11. **FTp[i] = succ(p + 2^(i-1)) mod 2^(m)**

FTp = 11[1] = succ(11 + 20) = 14, cioè mi dice che 14 è il nodo successore.

FTp = 11[2] = succ(11 + 21) = 14, cioè mi dice che 14 è il nodo successore.

FTp = 11[3] = succ(11 + 22) = 18,

FTp = 11[4] = succ(11 + 23) = 20

FTp = 11[5] = succ(11 + 24) = 28

Ovvero ho una lista di successori del nodo 11, via via “sempre meno precisa”.

Se cerco una key k, devo trovare “il nodo più lontano (leggasi la riga più in fondo) tale che k non sia superiore alla riga di questo nodo”.

Esempio: voglio cercare chiave 9.

1. K=9 la gestisce 11? No, lo saprebbe.
2. K=9 è gestita dal successore? Cioè è vero che k=9 è minore della riga 1 = 14, e maggiore del guid del nodo in questione 11? NO.
3. Allora esploro le righe, cercando la più lontana che non eccede la risorsa. Poiché questa condizione non si verifica mai (9 infatti è posseduta dal predecessore di 11) vado al nodo piu lontano, cioè 28.
4. PICCOLA OTTIMIZZAZIONE: so precedente, quindi potrei chiedere a lui.
5. Nodo 28 si chiede se la possiede lui (NO), e quindi cerca la riga piu lontana che non eccede 9, ovvero la riga 4. Quindi vado al nodo 4.
6. 4 si chiede se la possiede lui (NO), e vede se la possiede il successore, Si. Salto al nodo 9 e finalmente ho la risorsa.
7. **Con riferimento al sistema Chord basato su DHT raffigurato nella pagina seguente, supponendo che entri nel sistema il nodo 24, quali sono le operazioni che vengono eseguite nella rete e con quali valori viene inizializzata la sua finger table.**

Entra il nodo 24, quello che deve fare è trovare il suo successore.

FTp = 24[1] = succ(24 + 20 ) mod 32= 28, 28 è il suo successore.

Avvisa 28 che il suo nuovo predecessore è 24, e si fa dire chi fosse il vecchio predecessore (21).

Il nodo 24 contatta il nodo 21 e lo informa che 24 è il suo nuovo successore.

Il nodo 24 preleva le chiave comprese tra 21 escluso e 24 incluso: 22, 23, 24.

29/11/2017

Domanda 3)

1. **Si descriva lo stack dei servizi Cloud, approfondendo in particolare il livello IaaS e presentando le caratteristiche di un servizio Cloud pubblico a tale livello.**

Nello stack dei servizi cloud troviamo il livello Saas, cioè il livello più alto e vicino all’utente, in cui esso usa il servizio (come, ad esempio, GDRIVE o dropbox). Al livello intermedio troviamo PaaS, dove l’utente può sviluppare applicazioni e memorizzare dati, senza preoccuparsi di scaling o dei livelli sottostanti.

Il livello Iaas è il livello più basso, in cui l’utilizzatore deve gestire lo scaling, aspetti di applicazioni, memorizzazioni, middleware, sistemi operativi e runtime. L’unica cosa che non gestisce sono le risorse fisiche. Ne è un esempio AWS E2, nella quale non abbiamo controllo sulle risorse fisiche, ma possiamo scegliere Regioni, capacità delle Vm e istanze, SO da usare...

1. **Si spieghino quali sono le caratteristiche dei sistemi self-\* e qual è l’architettura di riferimento per la progettazione di un sistema autonomico.**

I sistemi self-\* hanno come caratteristica principale quella di auto-adattarsi ai contesti a cui sono posti. Ciò che varia è la scelta dei criteri da valutare, e la modalità degli approcci da seguire. L’auto adattamento viene realizzato in ottica di ottimizzazione delle risorse, individuazione di fault, gestione della sicurezza e integrazione di nuovi elementi senza interruzione del servizio.

Ciò viene realizzato tramite architettura MAPE che ha per componenti:

* Monitoring: campionamento dei dati tramite sensori o simili. Posso scegliere cosa monitorare, quando, e se simulare richieste o usare quelle reali.
* Analyze: analisi dei dati, si cercano delle tendenze. Se serve adattamento lo decide lui, ma non decide cosa adottare. Quando analizzo? Mi baso sul passato o sul futuro? Sono scelte.
* Plan: quale azione compiere. Posso usare euristiche, ML, threshold...
* Execute: applico il cambiamento scelto dal plan.

Esiste anche Knowledge, cioè memorizzo ciò che succede, magari è utile per situazioni che si ripropongono nel futuro.

1. **Si descriva un esempio di servizio autonomico a livello IaaS che soddisfa la proprietà di elasticità.**Un esempio è il servizio di autoscaling EC2. Fisso un numero di istanze minime (non posso averne meno di x), una soglia massima (non posso averne più di y) e poi posso mettere delle “regole” del tipo: se in DeltaT : %Cpu > 77% -> increase istances + 1

Se in DeltaM : %Cpu < 40% -> decrease instances – 1

La difficoltà poi risiede nella “scelta” di questi parametri.

Domanda 5)

1. **Si descrivano le caratteristiche dei pattern architetturali message queue e publish/subscribe, evidenziando anche quali tipologie di disaccoppiamento supportano e quali semantiche di comunicazione per la consegna dei messaggi.**

Nel Message queue, un producer invia, in una coda, messaggi destinati a uno o più consumer. Il messaggio viene consegnato una sola volta ad un solo consumer. Inoltre, la coda è persistente.

Questo comporta il disaccoppiamento temporale (c’è la coda), ma non garantisce necessariamente ed esplicitamente disaccoppiamento spaziale o di sincronia (il consumer, a meno che non implementi callback di notify, deve rimanere in attesa di msg).

Nel sistema publish/subscribe, c’è un terzo ente che fornisce il servizio di notifica. Il tutto si basa sui “topic”, quando producer invia un msg di un certo topic, lo manda all’ente terzo che mantiene tabella con i consumer iscritti a quel topic, indirizzando a loro il messaggio.

Grazie al dispatcher abbiamo disaccoppiamento spaziale (io invio al dispatcher, poi è lui che deve reindirizzare), ho anche disaccoppiamento di sincronia (perché c’è la notify). Se il dispatcher ha memoria ho anche disaccoppiamento temporale.

Supportano:

* At least once: con ack
* Exactly once: se c’è filtro duplicate
* **Basata su transazione: deve essere processato, poi cancello dalla coda.**
* Timeout: cancello dalla coda solo dopo che è stato corretamente ricevuto almeno una volta. (visibilità)

**Gli ultimi 2 parlano espressamente di code, quindi non possono essere di tipo P/S.**

**I primi due?**

**Exactly once non ha filtro: il filtro è il topic, se mando due msg uguali di topic uguale non viene filtrato.**

**At least once: mi devo “fidare” del servizio in mezzo, però non mi dice nulla.**

**DA RIVEDERE.**

1. **Si spieghi quali tipologie di disaccoppiamento sono supportate dalle implementazioni RPC esaminate a lezione (Sun RPC, Java RMI e Go)**

GO: disaccoppiamento rispetto alla sincronia, NO SPAZIALE NE TEMPORALE.

Semantica at most once.

SUN RPC: In alcune implementazioni disaccoppiamento rispetto alla sincronia.

SUN RMI: nessuna.

1. **Si spieghi se e come tali implementazioni RPC offrono trasparenza all’ubicazione, all’accesso e alla replicazione.**

SUNRPC:

* Trasparenza all’accesso: incompleta, devo specificare versioni etc.
* Trasparenza di ubicazione: NO, devo conoscere hostname del registry che risponde su porta default.
* Trasparenza di replicazione: NO, il portmapper è locale al server che offre il servizio, per cui dovrebbe essere replicato nel caso di più server., ma non è direttamente supportato.

JAVARMI:

* Trasparenza all’accesso: migliore, invocazioni locali e remoti sono molto simili.
* Trasparenza di ubicazione: NO, devo conoscere hostname.
* Trasparenza alla replicazione: NO, perché ho riferimenti UNICAST.

GO:

* devo sempre dire hostname e porta, quindi nessun meccanismo ha replicazione trasparente, devo per forza aggiungere un proxy che mi fa da load balancer, cioè intermediario, il client vedrebbe solo quello, non gli frega degli altri, forse è poco poco collo di bottiglia.
* in go rispetto ubicazione serve ip + porta (quindi è peggio rispetto gli altri due)

1. **Considerando un’implementazione decentralizzata di un sistema publish/subscribe di tipo topic based, si confrontino vantaggi e svantaggi nell’adottare il flooding oppure il gossiping come soluzione per la diffusione delle informazioni.**

* Gossiping: ... sempre le stesse cose...

7/12/2016

**Domanda 3 (punti 5)**

**Si descrivano quali sono le caratteristiche dei livelli IaaS e PaaS nel Cloud computing, discutendo anche le differenze tra i due livelli e presentando opportuni esempi di servizi offerti ai due livelli. Si descrivano inoltre le proprietà di un sistema autonomico e si presenti un esempio di servizio Cloud di livello IaaS o PaaS che può essere realizzato secondo i principi dei sistemi autonomici, evidenziandone gli obiettivi e spiegando come possono essere raggiunti.**

EC2 esempio Iaas

Amazon Beanstalk Paas, usa il loadbalancer.

Domanda 5 (punti 6)

**Si descrivano le caratteristiche del disaccoppiamento spaziale e del disaccoppiamento temporale e si spieghi se Java RMI ed i sistemi a code di messaggi supportano o meno tali tipologie di disaccoppiamento, motivando opportunamente la risposta. Si descriva inoltre una possibile implementazione distribuita di un sistema a code di messaggi, discutendo anche quali sono le principali funzionalità che tale sistema supporta.**

Disaccoppiamento spaziale: non devo conosce l’ubicazione dell’altra entità con cui sto comunicando (es receiver).

Disaccoppiamento temporale: le due entità non devono essere presenti allo stesso momento per lo scambio di messaggi.

Java RMI non supporta nessuno dei due: serve hostname -> no disacoppiamento spaziale

No disaccoppiamento temporale -> sincrona.

Implementazione sistema a code: elaborazione foto/video.

2/12/2015

**Domanda 2 (punti 6) Si descrivano le caratteristiche, le proprietà e la struttura dei protocolli di gossiping (o epidemici) e si discutano i loro vantaggi e svantaggi, considerando in particolare la diffusione dell’informazione in un sistema peer-to-peer. Si discuta inoltre se e come un protocollo di gossiping può essere usato per rilevare guasti dei server di un’infrastruttura Cloud distribuita su larga scala e quali vantaggi potrebbero derivare dall’uso di un protocollo di gossiping in tale contesto rispetto all’uso di un sistema a code di messaggi**

Gia fatto

**Domanda 3 (punti 5)**

**Si descrivano le caratteristiche e le proprietà della semantica di comunicazione di tipo at-least-once e di quella at-most-once. Si consideri come esempio una procedura o metodo Deliver(T, m) che permette di consegnare il messaggio m al nodo di destinazione T e si discutano le differenze tra le due tipologie di semantica nell’esempio considerato. Si spieghi inoltre quali meccanismi devono essere forniti da un middleware di comunicazione orientato ai messaggi (MOM) che offra una o l’altra semantica. Se il MOM supporta anche le transazioni, è in grado di garantire la semantica exactly-once e perché?**

At-least-once:

Richiede meccanismo RR1, ovvero Request Reply se, dopo un DeltaT di tempo, non si riceve risposta.

Invio il messaggio almeno una volta, perché il receiver potrebbe ricevere il msg ma crashare poco prima dell’ack, e quindi questo porta il consumer a rinviare il msg piu di una volta.

At most once:

Richiede RR1 appena visto, duplicate filtering, ovvero la possibilità di riconoscere un msg doppione analizzando l’id del msg, e Result retransmit: se ho già ricevuto un msg più volte, invece di rieseguire l’operazione, invio quella già eseguita. (non ricalcolo).

Nel caso di Deliver (T,m) nel caso at-least once, dovremmo quindi una iterazione che reinvia il msg se, dopo un certo clock di tempo, non si riceve alcuna risposta.

Nel caso at-most-once è uguale, perché solo RR1 è un aspetto client, gli altri due sono del server.

In MOM:

* Per at least once serve meccanismo di ack con timer.
* Per at most once in realtà dobbiamo “indebolire” exactly once? Transaction based è quell ache piu gli si avvicina. Non garantisce exacty once perché sono cose diverse.

3/12/2014

**Domanda 5 (punti 6)**

**Si descrivano le caratteristiche dell’architettura publish-subscribe, specificando anche le proprietà di disaccoppiamento supportate. Si discutano inoltre almeno due approcci per realizzare in modo distribuito la notifica degli eventi ai sottoscrittori in un sistema publish-subscribe di tipo topic-based, discutendone anche benefici e svantaggi.**

Publish subscribe c’è entità terza, si basa sui topic.

Topic un po' troppo generici, quindi no grana fine.

Disaccoppiamento spaziale (non mi serve ubicazione dei consumer perché mando tutto al service)

Disaccoppiamento temporale e di sincronia (io pubblico sul service, che ha una memoria, quindi i consumer ricevono una notify e non c’è bisogno che io sia presente).

Nel caso topic based, la notifica distribuita può essere tramite flooding o gossiping con tutte le conseguenze del caso.

transaction based ha meccanismi at least once + visibilità.

9/12/2013

*Domanda 5 (punti 6)*

*Si spieghi il funzionamento di una chiamata a procedura remota (RPC). Con riferimento all’implementazione di RPC esaminata a lezione, si discuta come SUN RPC supporta la trasparenza all’ubicazione e all’accesso e se tale supporto presenta delle limitazioni. Inoltre, si confronti il supporto alla trasparenza all’ubicazione e all’accesso di SUN RPC con quello offerto da Java RMI. Infine, si consideri una RPC nulla (RPC senza parametri che esegue una procedura nulla e non ritorna nessun valore) e si spieghi perché il suo tempo di esecuzione con client e server interconnessi in LAN è dell’ordine della decina di msec*

L’idea alla base di una RPC è quella di mappare 1-1 una chiamata a procedura locale. Questo vuol dire che l’obiettivo è quello di ridurre al massimo le astrazioni dovute alla parte “remota” della funzione rispetto alla sua controparte reale.

Per fare questo, si introducono gli stub client e server:

* Il client stub simula un’interfaccia server per il client, il quale vi si interfaccia come se fosse il server vero.
* Il server stub, che si trova lato server, simula una chiamata locale al server, come se fosse fatta dallo stub stesso, quando invece parte dal client vero e proprio.
* In mezzo, a seconda dell’implementazione, ci sono più tecniche per inviare i messaggi in un formato comprensibile.

SunRPC supporta in modo limitato la trasparenza all’ubicazione, questo perché richiede l’hostname del servizio portmapper su una porta di default.

SunRPC ha una trasparenza all’accesso limitata, poiché ci sono differenza tra chiamate e controparti stub (\_svc), e bisogna specificare dei numeri di versione.

In javaRMI ho una trasparenza all’ubicazione limitata, simile a SunRPC ma con RMI Registry.

La trasparenza all’accesso è migliore (ho meno elementi da specificare), ma comunque non è completa.

Una RPC void è comunque più lenta della controparte locale (altro motivo per la quale non ho trasparenza locale) poiché nel caso remoto devo considerare interprocessi di comunicazione.

10/12/2012

*Domanda 1 (punti 5)*

*Si spieghi il funzionamento di un’invocazione di metodo remoto (RMI). RMI supporta una forma di comunicazione transiente e sincrona: si spieghi cosa significa e quali sono i principali svantaggi. Inoltre, si presenti una possibile soluzione per rendere trasparente rispetto all’accesso, alla replicazione ed ai guasti l’invocazione di un metodo remoto replicato su molteplici server*

RMI = Remote Method Invocation.

Questo vuol dire che localmente creo oggetto che però è attivo su un host remoto.

Quindi client invoca metodi remoti sfruttando questa referenza locale.

La forma di comunicazione di java RMI è:

* Transiente: questo implica che non c’è persistenza dei dati, e quindi non ho disaccoppiamento temporale.
* Sincrona: Il client che chiama metodo remoto deve aspettare una risposta, non può procedere nel suo flusso di esecuzione. Questo comporta mancato disaccoppiamento rispetto alla sincronia.

Una possibile miglioria consente in un’entità di mezzo ti tipo “Load Balancer” che prende la richiesta del client e, senza che esso se ne accorga, la re-distribuisce in un server ritenuto idoneo. Il client non nota questa astrazione, e questo proxy non soffre eccessivamente il carico di lavoro, perché re-indirizza solamente,

“devo per forza aggiungere un proxy che mi fa da load balancer, cioè intermediario, il client vedrebbe solo quello, non gli frega degli altri, forse è poco poco collo di bottiglia.”