1 Lezione del 06-10-25

Continuiamo la discussione del DNS.

1.0.1 Caching DNS

Vediamo che anche nel DNS il caching può dare dei vantaggi in prestazioni non indifferentiVediamo che anche nel DNS il caching può dare dei vantaggi in prestazioni non indifferenti.

Quando un qualsiasi server DNS trova una data traduzioni fra dominio DNS e indirizzo IP, la memorizza (per ridurre il traffico DNS) con un certo **timeout**, sostanzialmente una data di scadenza della traduzione (solitamente impostata a 2 giorni). Molto spesso i server TLD (che raramente cambiano) sono memorizzati per primi, in questo modo i server Root vengono visitati di rado.

Infine, possiamo assumere che sia il S/O usato dall'utente che utilizza il DNS, che l'applicazione (metti il *browser*) che usa per collegarsi in rete, mantengono una loro cache di traduzioni. Questo significa che l'applicazione, per risolvere un DNS, effettuerà i passaggi:

- Consulterà la sua cache interna;
- Se non trovato, consulterà la cache del S/O;
- Se non trovato, effettuerà un richiesta al server DNS locale;
- Questo cercherà nella sua cache e se non trovato continuerà con i passaggi visti da 6.2.5 a 6.2.7.

1.0.2 Record DNS

Vediamo la struttura dei *Resource Record* (**RR**) memorizzati dai server DNS.

Questi hanno struttura generale:

```
(name, value, type, ttl)
```

e diversi tipi, fra cui:

- `type=A` (*Address*): `name` è il nome dell'host, e `value` l'indirizzo IP;
- `type=CNAME` (*Canonical NAME*): `name` è l'*alias* per qualche nome *"canonico"*, e `value` il nome canonico;
- `type=NS` (*Name Server*): `name` è il dominio e `value` è il nome di dominio del server autoritativo per questo dominio;
- `MX` (*Mail eXchange*): `value` è il nome del mail server associato a `name`.

1.0.3 Protocollo DNS

Vediamo alcune specifiche del protocollo DNS usato per inviare richieste (*query*) e risposte (*reply*). Anticipiamo subito che il protocollo DNS si basa su UDP e non TCP.

Abbiamo quindi che la struttura delle richieste e delle risposte è la stessa:

```
2 byte                    2 byte
<id>                      <flags>
# domande (query)         # risposte RR (reply)
# autorita' RR (reply)    # RR aggiuntivi (reply)
...
```

L'identificatore (`<id>`) è su 16 bit e viene usato per riconoscere i destinatari delle risposte. I flag (`<flags>`) sono vari e specificano:

- Se è un messaggio *query* o *reply*;
- Se si richiede ricorsione (per messaggi query);
- Se la ricorsione è disponibile (per messaggi reply);
- Se la risposta è authoritative, cioè viene da un server DNS authoritative.

1.0.4 Inserire record in DNS

Vediamo quindi il processo usato per registrare nomi di dominio nel DNS. Mettiamo che la startup *"Alpha S.r.l."* voglia registrare un dominio per il suo sito:

- Dovrà registrare il nome rivolgendosi ad un'azienda specializzata, e fornendo il nome di dominio (che non deve essere già stato registrato), l'indirizzo IP del server authoritative primario e secondario;
- L'azienda specializzata inserirà record di tipo NS e A in un server TLD (mettiamo `.com`):

```
(alpha.com, dns1.alpha.com, NS) # RR NS dal nome di dominio al DNS
    autoritativo
(dns1.alpha.com, 212.212.212.1, A) # RR A dal nome del DNS
    autoritativo al suo indirizzo
```

- A questo punto Alpha S.r.l. dovrà configurare i suoi server DNS authoritative inserendo record di tipo A e volendo MX (per la posta elettronica):

```
(www.alpha.com, 212.212.212.2, A) # RR A dalla pagina web all'IP del
    server che la fornisce
(alpha.com, mail.alpha.com, MX) # RR MX al nome del mail server
(mail.alpha.com, 212.212.212.3, A) # RR A dal nome all'IP del mail
    server
```

1.1 Applicazioni P2P

Abbiamo visto una certa gamma di applicazioni in rete che adottano il paradigma client-server: ad esempio il Web, l'e-mail, e proprio adesso il DNS.

Vediamo adesso l'altro grande paradigma, quello del **Peer-to-Peer**. Ricordiamo che la caratteristica delle applicazioni P2P è che non c'è un server centrale sempre attivo, ma sistemi host finali comunicano fra di loro in maniera arbtiraria: i *peer* richiedono servizi ad altri peer, fornendo in cambio altri servizi.

Questa architetturà è *auto-scalabile* (più peer, più grande la rete), complesso da gestire e sostanzialmente decentralizzato.

Ci sono diverse applicazioni realizzabili come P2P. Possiamo classificarle largamente:

- **Condivisione contenuti**: i peer rendono disponibili contenuti multimediali o in generale file ad altri peer, in maniera decentralizzata. Dal punto di vista legale questo fornisce una piattaforma abbastanza sicura per lo scambio di contenuti illegali (film e musica pirata, ecc...). Celebre è l'esempio di *Napster*;
- **Messaggistica istantanea**: gli *username* degli utenti devono essere mappati a certi indirizzi IP. Quando un utente va online un *indice* è notificato col suo indirizzo IP: a questo punto altri utenti possono contattarlo in P2P a tale indirizzo.

1.1.1 Indici P2P

Vediamo che in entrambi casi presentati nella scorsa sezione (nel primo non si è detto ma è chiaro riflettendo sulle soluzioni tecniche) abbiamo bisogno di **indici**: per stabilire quali peer contattare per ottenere una data risorsa, bisogna avere un modo di effettuare ricerche sui loro indirizzi IP.

Un indice P2P è organizzato come un database di coppie chiave-valore, ad esempio:

```
Led Zeppelin IV, 203.17.123.38
```

I peer effettuano *query* a questo database per trovare i peer che possiedono date risorse; i peer possono anche *inserire* nuove coppie chiave-valore (ad esempio per segnalare che possiedono una nuova risorsa).

Il database può essere realizzato in più modi:

1. Si può dislocare un singolo **indice centralizzato** per tutti i peer. I problemi di questo sistema sono ovvi: che può essere facilmente usato per risalire ai peer (dubbia sicurezza) o sovraccaricato per rendere l'intero sistema P2P inutilizzabile.

   Diciamo infatti che applicazioni P2P con indici centralizzati sono P2P *"ibridi"*, con funzionalità client-server per l'ottenimento degli IP dei peer.

   Un sistema di questo tipo veniva usato da *Napster*, con celebri risultati (il server centrale viene chiuso e l'intera applicazione cade);

2. Si può sfruttare il **query flooding** per realizzare un indice *decentralizzato*: secondo questo paradigma ogni peer forma una parte dell'indice.

   In questo caso la fase di distribuzione di contenuti è banale: non bisogna collegarsi ad un indice centralizzato, ma ogni peer diventa parte dell'indice nel momento stesso in cui inizia a condividere un contenuto.

   Il problema giunge in fase di ricerca di contenuti: abbiamo che il client che deve ricevere contenuti deve limitarsi a *chiedere* agli altri peer "vicini" se hanno il contenuto desiderato. Definiamo i peer come vicini se hanno fra di loro una connessione TCP attiva. La richiesta può chiaramente diffondersi di vicino in vicino visitando l'intera rete o un suo sottoinsieme, e in questo modo si implementa effettivamente, se non per "forza bruta", un indice decentalizzato.

   Per ridurre il traffico sulla rete P2P (sistemi di questo tipo tendono a emanare molto traffico) si può usare il *Limited-Scope* query flooding, cioè inviare richieste non a tutti gli amici, ma a un sottoinsieme, o impostando un time-to-live per le richieste.

   Un'altro problema è chiaramente il *bootstrapping*: come trovare il primo set di vicini? In questo caso si possono usare scansioni, cache temporanee di peer, peer preimpostati (che hanno però il problema della centralizzazione) o l'intervento manuale dell'utente.

   Questa era la soluzione usata da servizi come *LimeWire*.

3. Un'approccio che combina il meglio dei due scorsi è un'approccio **gerarchico**. In questo caso prevediamo una rete simile a quella usata nel query flooding, ma che prevede dei *Super Nodi* (**SN**) (o *Super Peer*, **SP**), cioè peer con alta bandwidth e disponibilità alle connessioni. Ogni SN gestisce una rete locale, con un suo indice locale, e i peer di quella rete lo interrogano come avrebbero interrogato il server centrale del primo caso. A questo punto la trasmissione avviene normalmente fra peer, ammesso che i peer siano all'interno della stessa rete locale gestita dal SN. Non si esclude la possibilità che un SN non trovi il contenuto richiesto nel suo indice locale: in questo caso è libero di collegarsi ad altri adottanto sostanzialmente il query flooding, che però sarà più efficiente per diverse ragioni (meno supernodi formano una rete più compatta, i supernodi hanno specifiche migliori di trasmissione, i supernodi sono pensati appositamente per questa operazione (a differenza dei peer normali che vogliono perlopiù scaricare e non dare), ecc...).

   Chiaramente il problema sarà di non *"centralizzare"* troppo i SN, cosa che li renderebbe obiettivi di attacchi o cause legali.

   Questa soluzione veniva usata da servizi successivi a *LimeWire*, come ad esempio *FastTrack*.

4. Un approccio interessante è quello dato dalle **DHT** (*Distributed Hash Table*).

   Questo è l'approccio usato da *BitTorrent* e *eMule*.