

---

# P2P Networking

Tecnologie e Protocolli per Internet 2 (TPI2)

*rev 0.2*

**Andrea Detti**

University of Roma “Tor Vergata”

Electronic Engineering dept.

E-mail: [andrea.detti@uniroma2.it](mailto:andrea.detti@uniroma2.it)



---

***Un approccio è P2P quando chi fruisce  
di qualcosa offre anche qualcosa***

# Applicazioni P2P

---

- *“Un applicazione di rete è P2P quando chi genera le informazioni oggetto dell’applicazione sono gli stessi fruitori di queste informazioni; ovvero NON vi è una distinzione di ruolo fra la sorgente delle informazioni ed il fruitore”*
- Una applicazione di Web Browsing classica **NON** è una applicazione P2P in quanto vi è una chiara distinzione di ruolo fra chi scrive la pagina (i.e., web master) e chi la legge (i.e., utente finale)
- Un **Wiki** è una applicazione WEB P2P in quanto chi legge la pagina può anche scriverla
- Un **Messaging** ( o **Skype**) è una applicazione P2P in quanto sia le informazioni di presenza che testuali (o VoIP) sono generate ed usufruite dagli utenti finali

# Applicazioni P2P

---

- **Emule/Torrent** è una applicazione P2P in quanto sia le informazioni di disponibilità dei file che i file stessi sono generati dagli utenti finali
- **Facebook, Youtube** sono applicazioni P2P
- Una **telefonata** su PSTN è una applicazione P2P
- Una **mailing list** è una applicazione P2P
- La **E-mail** è una applicazione P2P

# Gestione delle Risorse HW P2P

---

- **Def. SRD:** *“Una applicazione P2P, o una sua funzionalità, ha una gestione delle risorse HW (memoria di massa, banda d’accesso ad internet, CPU) di tipo P2P quando i fruitori dell’applicazione condividono parte delle loro risorse HW per le finalità della applicazione stessa”*
- La condivisione delle risorse HW assicura una **elevata scalabilità** verso il numero di partecipanti

# Gestione delle Risorse HW P2P

---

- Un **Wiki** (così come Facebook, Youtube, mailing-list, email) **NON** è una applicazione di rete con gestione delle risorse HW P2P, in quanto la maggior parte delle risorse sono messe in capo dall'host che ospita il Wiki Server
- Un **Messaging** basato su **Server** che mantengono le informazioni dell'utente (e.s., presenza, lista amici, etc.) e gestiscono il forwarding multicast dei messaggi **NON** è una applicazione di rete con gestione delle risorse HW P2P, in quanto la maggior parte delle risorse sono messe in capo dall'host che ospita il Server di Messaging

# Gestione delle Risorse HW P2P

- **Emule Server-based** e **Torrent** sono applicazioni P2P di file sharing con gestione delle risorse HW **principalmente** P2P
  - » **NO HW P2P**: vi sono dei Server che mantengono (alcuni) metadati degli utenti (e.s., presenza, lista file condivisi, etc)
  - » **HW P2P**: il trasferimento dei file è fatto peer-to-peer su socket fra utenti
  - » Si osserva come la **principale** richiesta HW (i.e., banda) si ha nella funzionalità di data-transfer e pertanto, essendo questa P2P, l'approccio ha una buona scalabilità vs il numero di utenti . Questo approccio di **gestire in P2P solo le funzionalità che richiedono il principale sforzo HW** è spesso utilizzato
- **Emule Kab-based** è una applicazione con gestione delle risorse HW **totalmente** P2P, infatti anche il mantenimento dei meta-dati degli utenti e dei file è fatto dagli utenti stessi

---

# Bandwidth sharing: l'approccio BitTorrent

(gestione delle risorse HW P2P)

(ref. <http://wiki.theory.org/BitTorrentSpecification>)

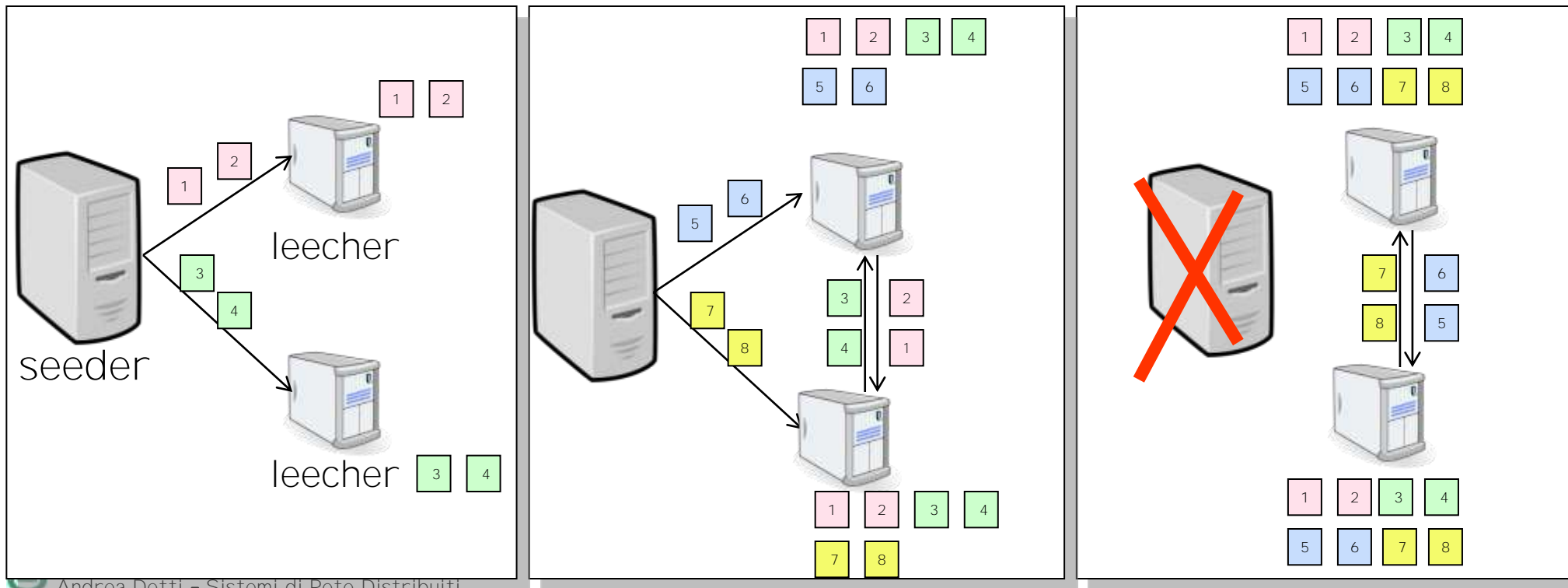
# BitTorrent

---

- **BitTorrent è una applicazione P2P per la condivisione di file**
- **Vi è una gestione delle risorse HW principalmente P2P,**
  - » **Le comunicazioni che riguardano (alcuni) metadati sono peer-server**
  - » **Le comunicazioni che riguardano i dati (ed altri metadati) avvengono direttamente fra peer**
- **L'architettura di rete overlay è “client-server e P2P” (discussione successiva ...)**

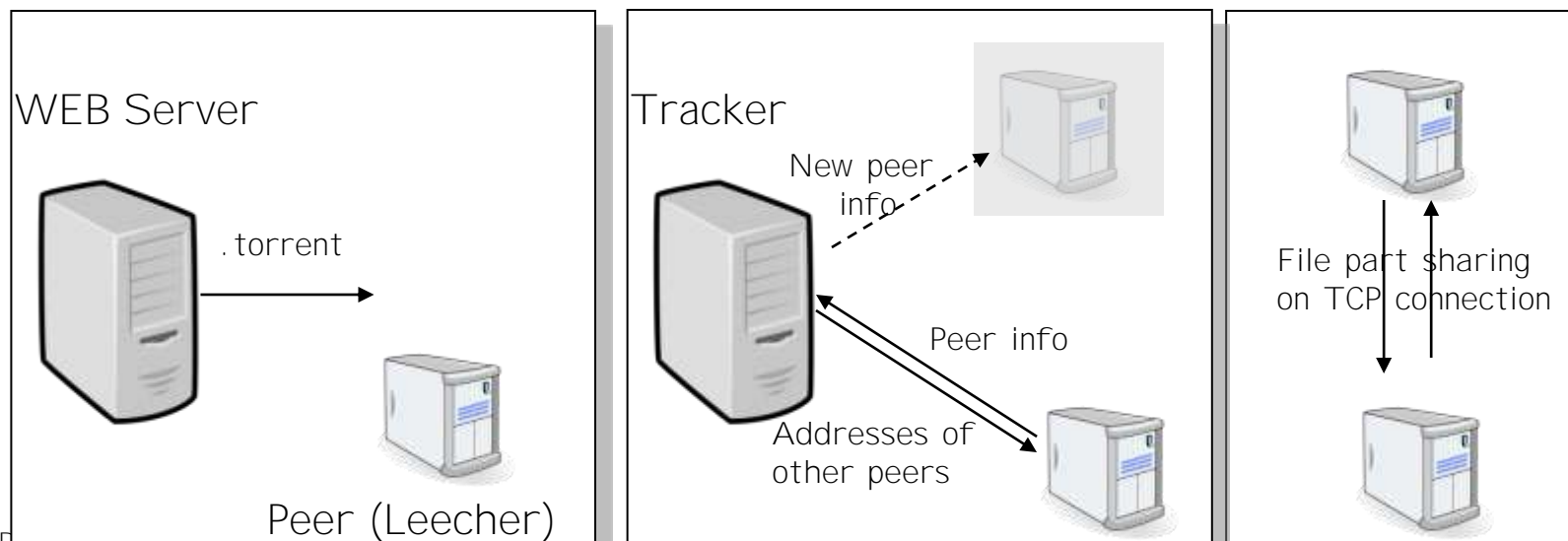
# Bit Torrent – Principio

- Un peer depositario di un file (*seeder*: alla lettera insemminatore) distribuisce parallelamente pezzi diversi del file a diversi peer che lo richiedono (*leecher*: alla lettera succhiatore)
- Di conseguenza, un *leecher* a cui manca il pezzo #*n* può richiederlo sia al *seeder* e sia ad altri *leecher* che stanno scaricando lo stesso file → Sharing della banda utilizzata per il trasferimento del file fra i partecipanti (i peer dello **SWARM** – alla lettera schiame)
- Inoltre, se il *seeder* va via e ha già “inseminato” (i.e., trasferito) nello swarm tutti i pezzi del file, i *leecher* possono autonomamente continuare con successo il download
- Quando un *leecher* possiede tutto il file diventa *seeder* e può (per spirito di condivisione) o meno uscire dallo swarm



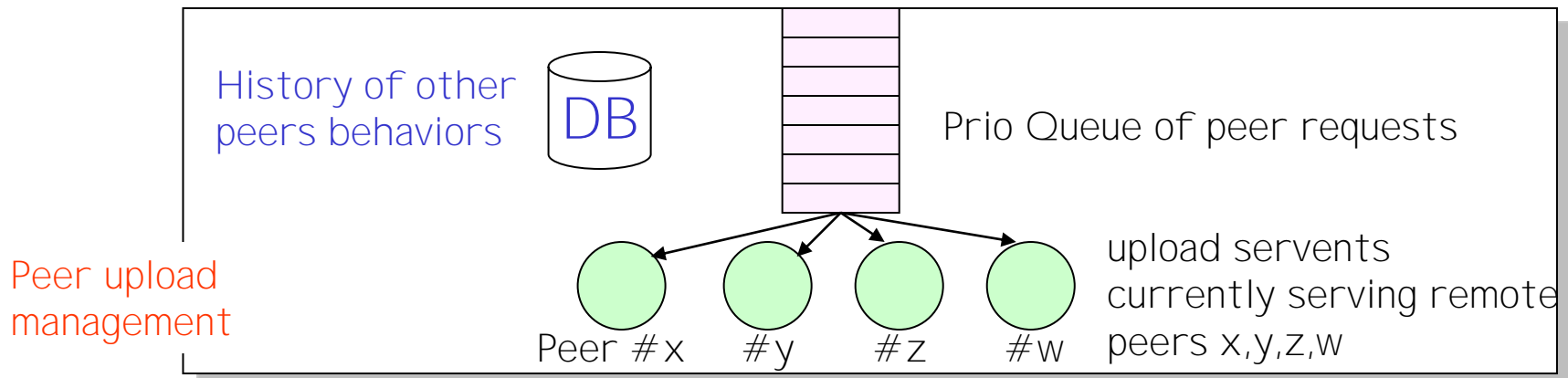
# Bit Torrent – Architettura di rete

- Ogni file viene decomposto in pezzi (e.s. 2MB), ogni parte è composta da blocchi da 16 kB
- Il seeder che intende condividere un file genera un descrittore del file, il file .torrent, e lo pubblica su un **server WEB pubblico**
- Il descrittore contiene un riferimento ad un ulteriore server pubblico, il **tracker** che coordina la distribuzione del file
- Il leecher D che desidera scaricare il file
  - » scarica il descrittore del file e lo apre con un client BitTorrentsi
  - » si connette al tracker e lo informa della propria esistenza (peer ID, IP address/port, .torrent)
  - » riceve dal tracker una lista di ulteriori leecher/seeders che stanno scaricando o condividendo il file
- I peer si scambiano informazioni relative ai pezzi del file che essi possiedono e ogni peer richiede i pezzi di file a cui è interessato su connessioni TCP



# Bit Torrent – Accodamento delle richieste – Tit-for-Tat

- Il numero di pezzi che parallelamente un peer trasmette in upload verso altri peer richiedenti è limitato ad un valore  $N$  (e.s., 4)
- Pertanto, modellando il “pezzo richiesto” come una richiesta di servizio, siamo di fronte ad un **sistema a coda con  $N$  serveri e fila d'attesa**. Il tempo di servizio dipende dalla banda di upload del peer, dalla banda che c'è nel percorso e2e peer-peer e dal numero di pezzi che il peer sta trasferendo in uscita (upload)



# Bit Torrent – Accodamento delle richieste – Tit-for-Tat

- Al fine di evitare la presenza di “free riders” (a un membro che evita di dare il suo contributo al bene comune poichè ritiene che il gruppo possa funzionare ugualmente, i.e. banda di upload), la strategia di coda è regolata da un meccanismo di scheduling che favorisce la cooperazione bilaterale fra peers
- L’approccio formale a questo problema di scheduling si basa sulla “Teoria dei Giochi”. Un algoritmo spesso efficace alla massimizzazione del bene comune è il “tit-for-tat” (pan-per-focaccia) in cui la mia prossima mossa è uguale a quella precedente del mio avversario, con il vincolo iniziale che la prima mossa del giocatore che entra nel gioco è “cooperativa” verso gli altri, invece che “non-cooperativa”
- Il tit-for-tat nel “gioco” BitTorrent, ma anche Emule, consiste nel favorire le richieste dei peer che mi hanno dato qualcosa (i.e., la loro mossa precedente è cooperativa nei miei confronti). Questo concetto si concretizza con specifici meccanismi protocollari (choke/unchoke per BitTorrent e sistema a crediti per Emule)
- In BitTorrent il “gioco” dura per tutto il periodo che sono nello *swarm*. Uscito dallo swarm, se dopo ci rientro il gioco ricomincia da 0

# Bit Torrent – Strategia dei pezzi da richiedere

- Pezzi (o macro-pezzi) diversi sono chiesti a peer diversi. A questo c'è una eccezione di *End Game*
- Verso un peer posso fare una richiesta alla volta ed ogni richiesta può far riferimento solo ad un macro-pezzo formato da pezzi contigui
- Una richiesta formata da  $N$  pezzi sarà comunque accodata dal peer ricevente come  $N$  richieste
- In che sequenza chiedo i pezzi dagli altri peer ?
- **Rarest First**: chiedo prima i pezzi più rari, in questo modo vengo in possesso di pezzi rari, quindi molti vorranno da me questi pezzi e nel tit-for-tat acquisisco fiducia
  - » Per evitare il collo di bottiglia sulla coda del peer che possiede attualmente il pezzo più raro, in realtà si sceglie in modo randomico un pezzo fra i più rari
- **End Game**: quando per un peer  $S$  il download è quasi completo, rimangono pochi pezzi da scaricare e quindi il numero di peer che parallelamente può trasferire ad  $S$  i pezzi mancanti diminuisce con una conseguente diminuzione del download-rate. Per velocizzare questa fase finale, quando il numero di pezzi mancanti va al di sotto di una certa soglia allora  $S$  richiede a tutti i peer  $I$  i pezzi mancanti, preoccupandosi di rimuovere la richiesta sugli altri peer quando è riuscito a scaricare un pezzo

# BitTorrent & NAT

---

- I peer che fanno upload devono poter essere raggiunti attraverso un indirizzo IP pubblico
- Se un peer si trova dietro ad un Router NAT e questo peer non vuole essere un free-rider, il router deve essere configurato in *port forwarding* sulle porte TCP sulle quali il peer accetta connessioni entranti (e.s. dalla 6881 alla 6889)
- Pertanto, **l'utente peer deve essere anche l'amministratore del Router NAT**

---

# Architettura di rete per applicazioni P2P

# Architettura di rete P2P

- Gli host che partecipano ad una applicazione P2P sono connessi fra loro da socket TCP/UDP. L'insieme delle connessioni forma una **overlay network**
- L'architettura della overlay può essere
  - » Client-Server pura
  - » "Client-Server e P2P"
  - » P2P pura
  - » P2P ibrida
- Tipologie di informazioni scambiate nella rete P2P
  - » **Meta-dato**: sono informazioni aggiuntive che caratterizzano il dato oggetto dell'applicazione (e.s., nome di un file, IP address/port dove lo posso trovare, etc., presenza di un peer che appartiene alla mia buddy list, etc)
  - » **Dato**: è l'informazione oggetto dell'applicazione (file, caratteri di una comunicazione chat, etc.)

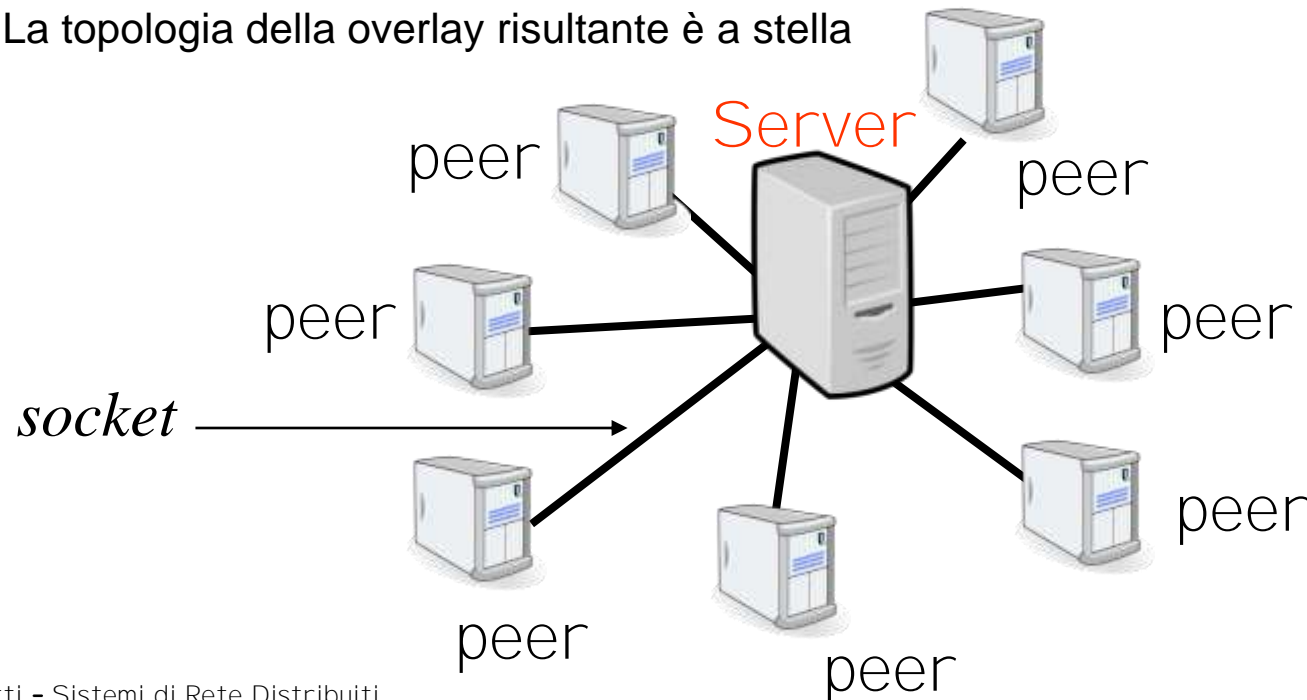
---

# **Overlay Client-Server**

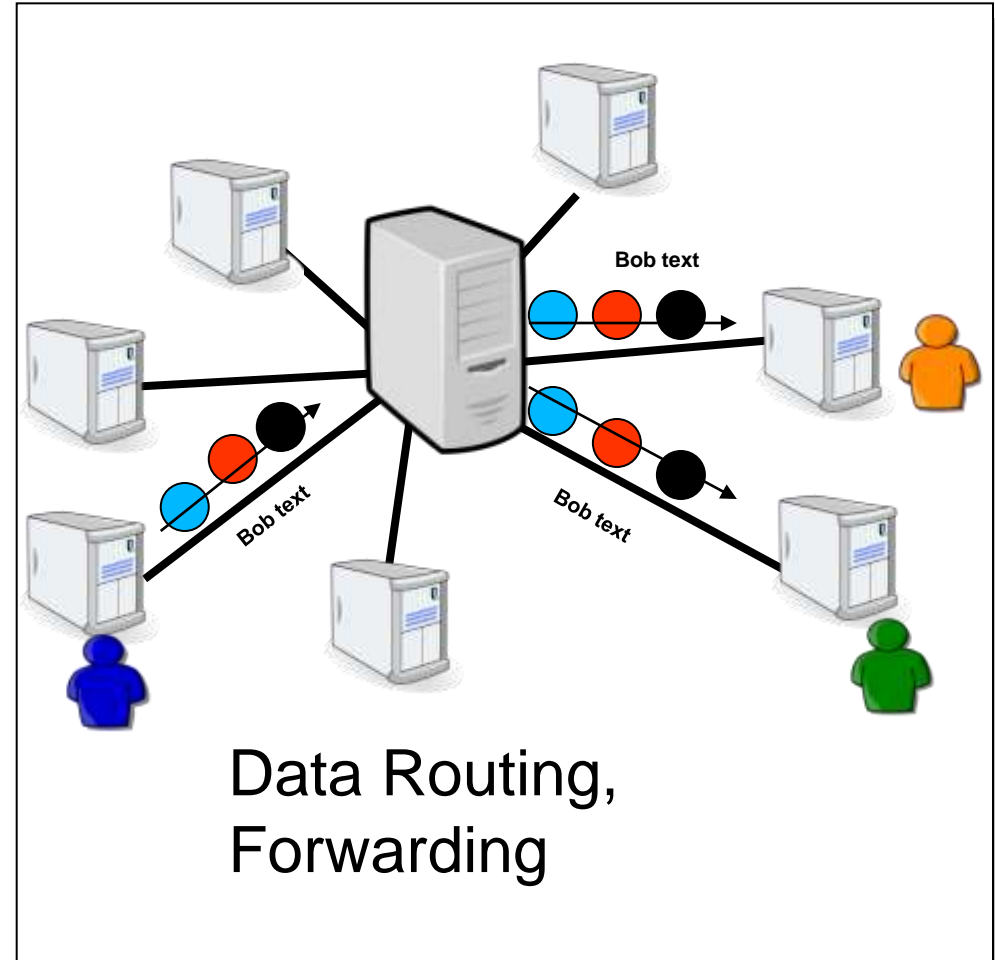
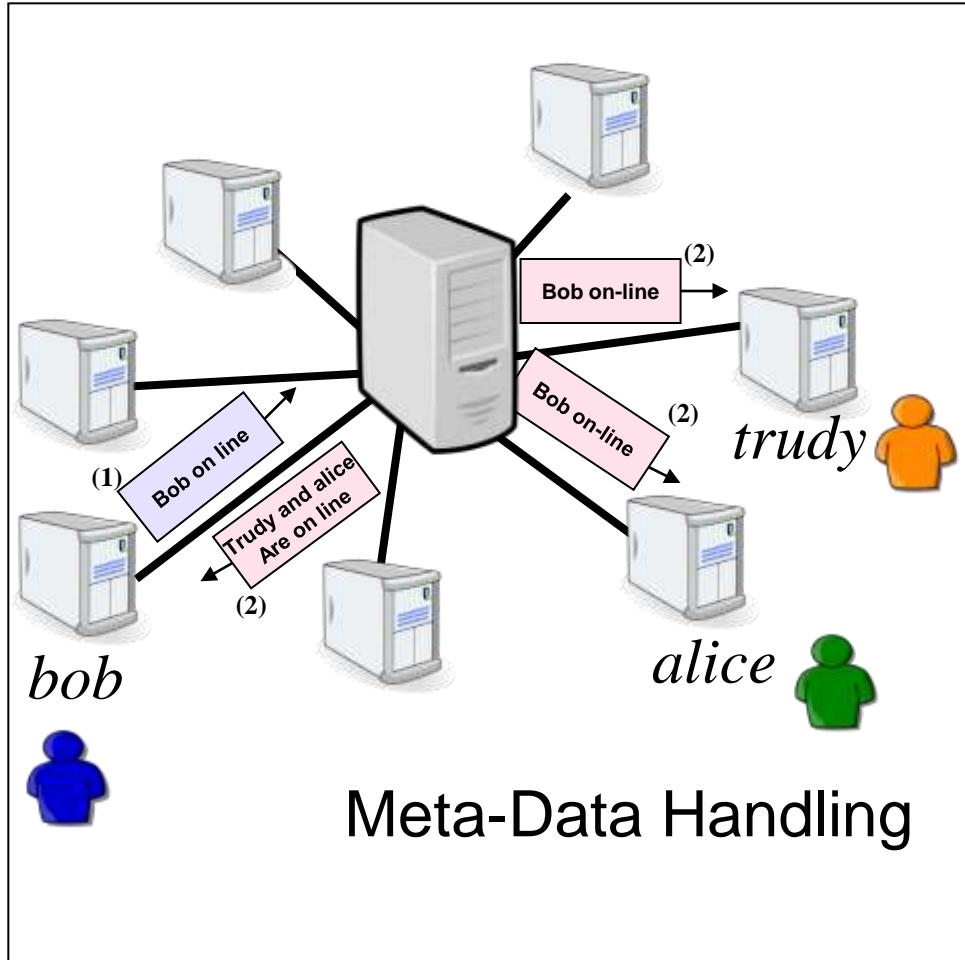
**(Architetture di rete per applicazioni P2P)**

# Overlay Client-Server

- Ogni peer si collega ad un server, ed è quest'ultimo che gestisce:
  - » Lo storage dei metadati di tutti
  - » Il lookup / advertisement dei metadati nei confronti di tutti i peer
  - » il routing/forwarding dei dati di un peer verso gli altri peer
  - » Funzionalità aggiuntive (e.s., transcoding, cyphering)
- Il server non è un fruitore dell'applicazione
- Il peer gestisce
  - » Lo storage dei propri metadati
  - » Il lookup / advertisement dei metadati solo verso il server
  - » Il forwarding dei dati verso il server
- La topologia della overlay risultante è a stella



# Overlay Client-Server - Esempio di servizio Messaging



# Overlay Client-Server – Pros & Cons

## ● Vantaggi

- » L'architettura client-server pura è la più veloce (**one-hop overlay lookup**) in termini di lookup ed assicura un **lookup completo** su tutti i metadati
- » Richiede un overhead di rete (one-hop overlay) minimo per l'aggiornamento ed il lookup dei metadati
- » È possibile gestire meccanismi di routing/forwarding con funzionalità aggiuntive complesse
  - » E.s. *transcoding*: cambiamento del formato dei dati in forwarding
    - Mixing di sorgenti voce nel caso di voice-conference in un unico stream voce
    - Adattamento del video coding alle capacità d'accesso dei peer
    - ...
- » Gestione della sicurezza semplificata poiché centralizzata

## ● Svantaggi

- » Single Point of Failure: il server
- » Implicitamente non c'è una gestione P2P dell'HW, pertanto il server può richiedere un HW significativo sia in termini di CPU che Banda d'accesso  
→ **limitata scalabilità**

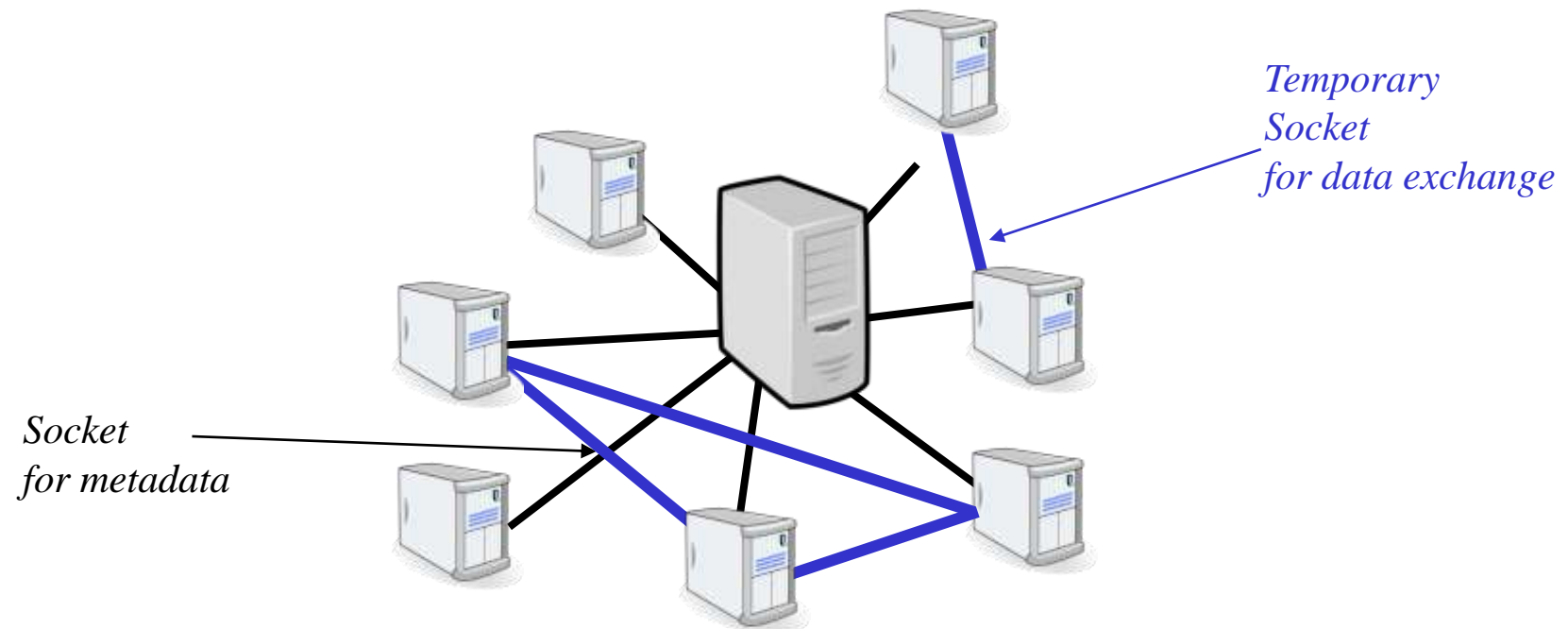
---

# Overlay “Client-Server e P2P”

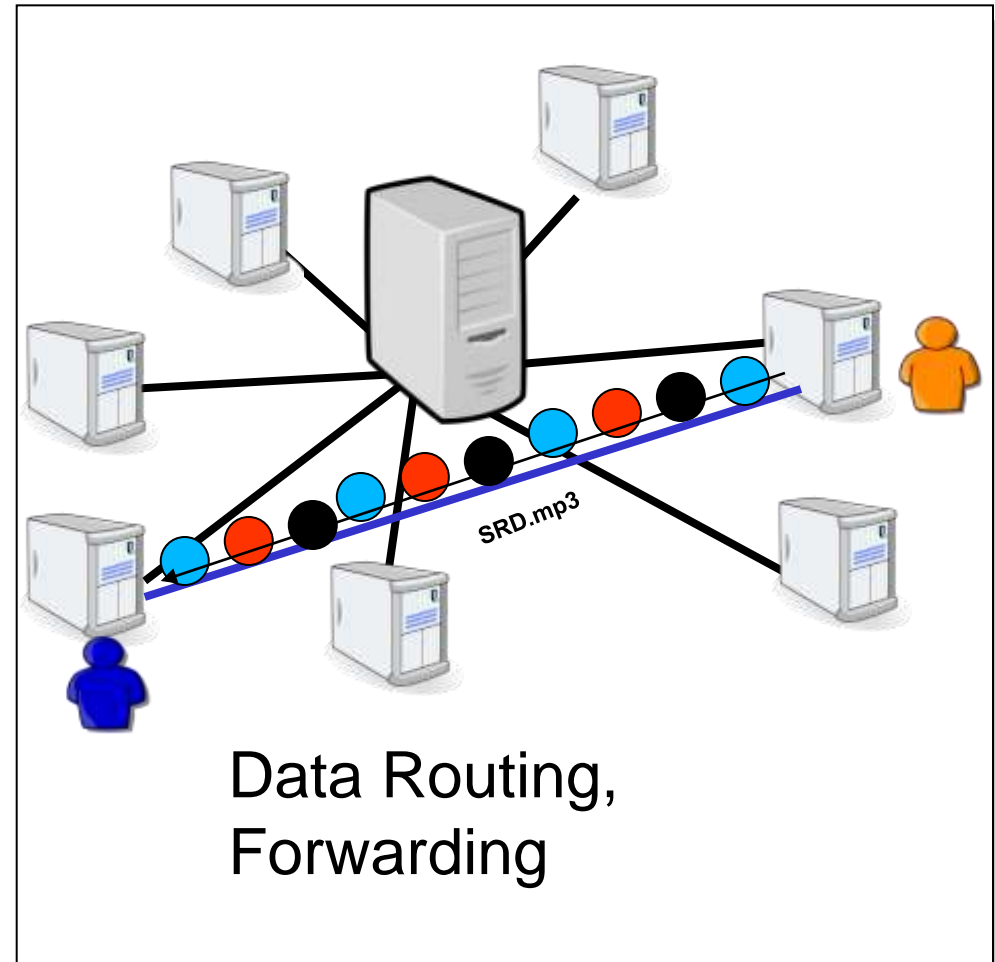
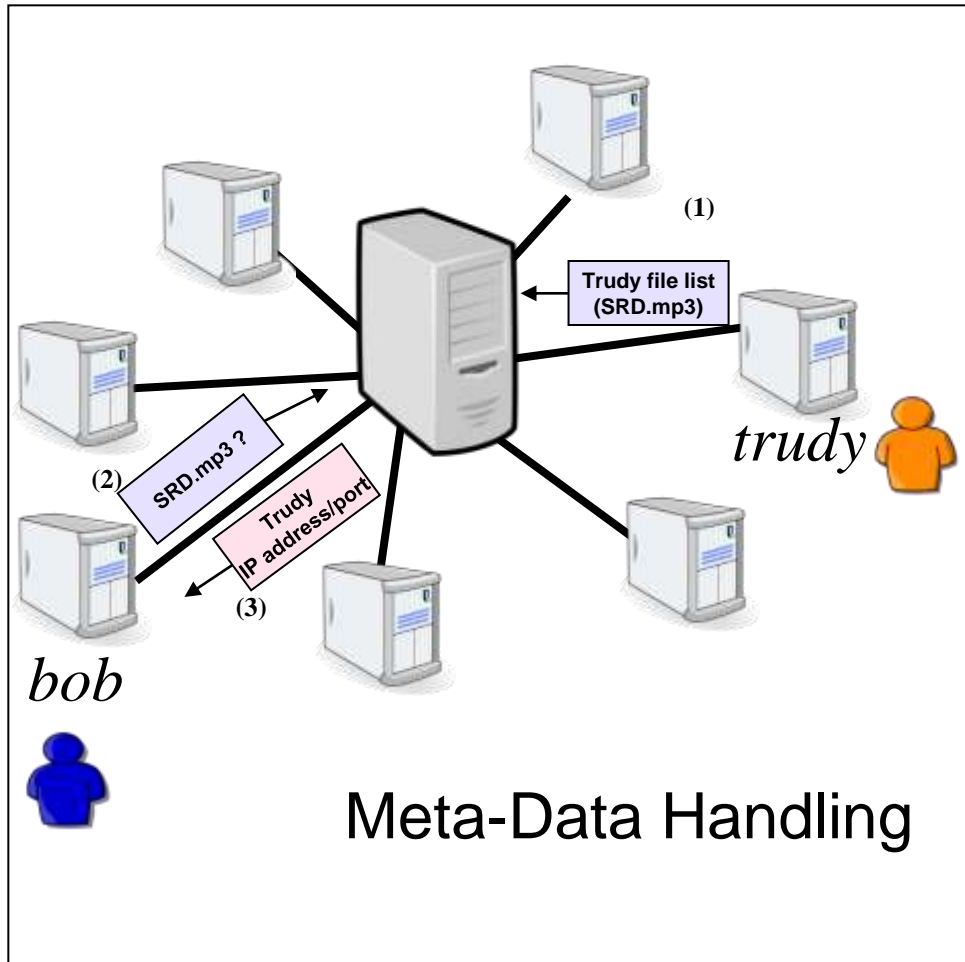
(Architetture di rete per applicazioni P2P)

# Overlay Client-Server e P2P

- Ogni peer si collega ad un server per le comunicazioni inerenti i *meta-dati*
- Il routing/forwarding/... dei dati è gestito dai peers
- Il server non è un fruitore dell'applicazione
- La topologia della overlay risultante è:
  - » una **stella con il server** (utilizzata dai metadati)
  - » una **mesh dinamica fra peer** : quando due peers vogliono scambiarsi dati si attiva **temporaneamente** uno (o più nel caso di overlay multicasting) link overlay fra i peer



# Overlay Client-Server e P2P - Esempio di servizio File Sharing (*Emule Server based*)



# Overlay Client-Server e P2P – Pros & Cons

---

- Vantaggi

- » Lookup veloce e completo dei metadati
- » Richiede un overhead di rete (one-hop overlay) minimo per l'aggiornamento ed il lookup dei metadati
- » Sharing del carico HW della banda per il trasferimento dei dati fra i peers

- Svantaggi

- » Single Point of Failure: il server
- » Non c'è una gestione P2P dell'HW completa, pertanto il server può richiedere un HW significativo sia in termini di CPU che banda d'accesso, tuttavia lo HW del server deve gestire solo i metadati; quindi, rispetto alla soluzione Client-Server pura, si ha una migliore scalabilità

---

# Emule Server Based

**(Overlay Client-Server e P2P)**

Ref.: <http://www.cs.huji.ac.il/labs/danss/p2p/resources/emule.pdf>



# Emule Server Based

---

- **Emule è una applicazione P2P per la condivisione di file**
- **I file sono suddivisi in pezzi (chunk) da 9.28 MB e blocchi da 180 kB**
- **L'architettura di rete overlay è client-server e P2P**
- **Vi è una gestione delle risorse HW principalmente P2P concettualmente simile a BitTorrent)**

# Emule Server Based – Il peer

- Possiede un identificativo unico da 128 bit (user id) generato in modo pseudo-random all'accensione del software
- Possiede una lista di servers eMule precaricata/aggiornabile
- Si connette ad un unico server eMule mediante una connessione TCP.
- Il server può essere modificato dinamicamente con l'intervento dell'utente
- Apre centinaia di connessioni TCP con altri client eMule per effettuare il download/l'upload dei files con approccio di bandwidth sharing simile a bit-torrent
  - » può scaricare frammenti diversi dello stesso file da clients eMule diversi
  - » può effettuare l'upload di frammenti di un file F anche se il download di F non è ancora stato completato
- invia pacchetti UDP ai servers presenti nella sua lista per verificare la loro presenza sulla rete, per migliorare la ricerca di files,....
- invia pacchetti UDP agli altri peers per controllare il proprio stato nelle code di richieste degli altri peers

# Emule Server Based – Il server

---

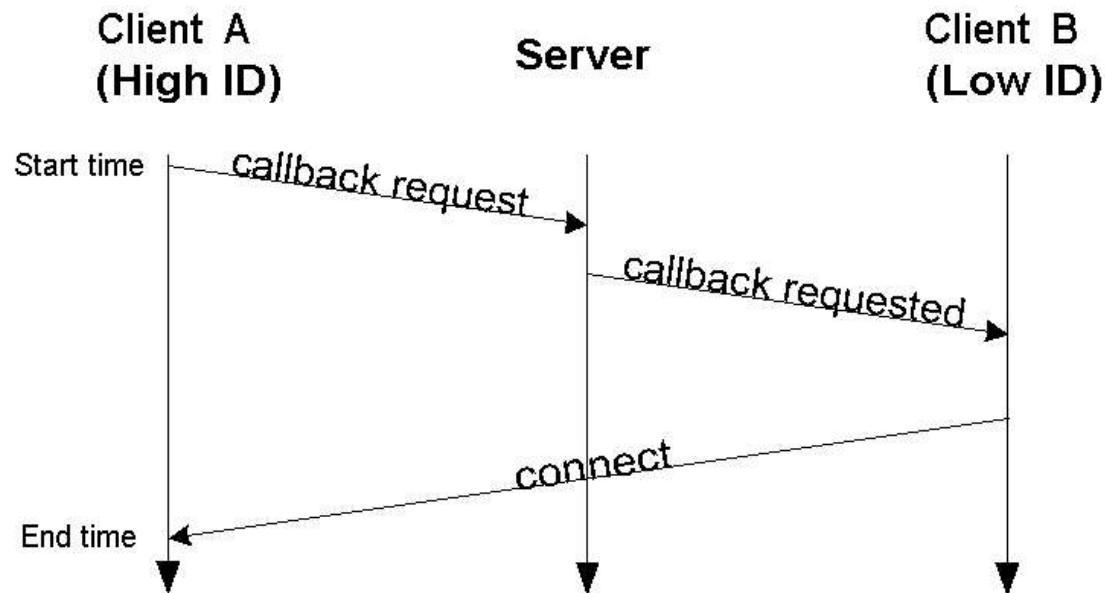
- È un host con indirizzo IP pubblico
- Possiede un database in cui memorizza i metadati dei files messi in condivisione dagli utenti ad esso connessi
- Invia ad ogni peer (client) eMule, al momento della sua connessione, informazioni circa il numero di utenti connessi ad esso e sul numero di files condivisi
- Non interagisce con gli altri server eMule
- **Può essere utilizzato come 'intermediario' per utenti a monte di NATs/firewalls**

# Emule Server Based & NAT

- È possibile ripetere lo stesso approccio descritto per BitTorrent & NAT, i.e. port-forwarding.
- In aggiunta però, nei casi in cui l'utente peer NON può amministrare il Router NAT, Emule permette *l'intermediazione* di un server Emule per le comunicazioni peer to peer – meccanismo delle *callback*
- Ogni client emule connesso con un server è caratterizzato da un *client ID* assegnato dal server
  - » *Client ID basso* : il peer non è direttamente raggiungibile dall'esterno (e.g., è dietro un NAT senza port forward)
  - » *Client ID alto*: viceversa
- Per capire se un peer può o meno essere raggiunto dall'esterno, il server invia un messaggio di *HELLO* al peer su un socket TCP aperto dal server, se il peer risponde allora è raggiungibile dall'esterno

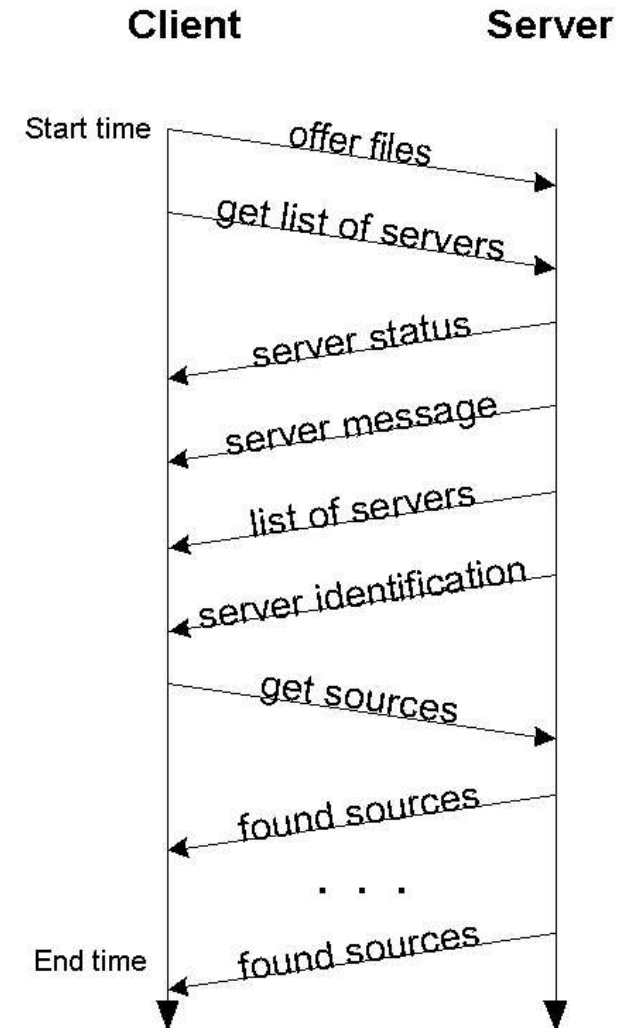
# Emule Server Based - Callback

- Problema: come fa un un peer con client ID alto a comunicare con un altro peer che ha client ID basso ?
- Soluzione: fa iniziare la connessione dal peer con client con ID basso comunicando questa richiesta sul socket che il peer con client con ID basso ha aperto con il server
- I client con ID basso possono fare upload solo verso i client con ID alto dello stesso server → i client con ID basso acquisiscono pochi crediti (tit-for-tat)



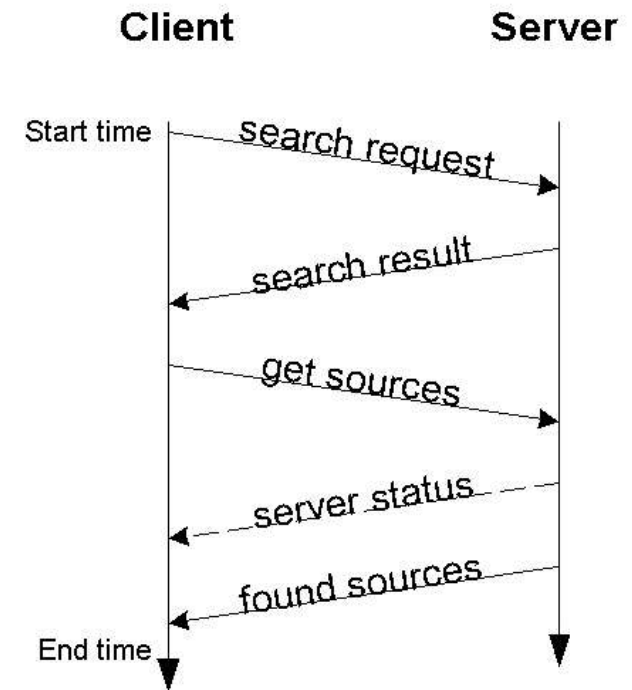
# Interazioni peer-server allo startup

- L'obiettivo è quello di aggiornare su entrambi le parti lo "stato del peer"
- Il peer (client) invia la lista dei files condivisi e la lista dei file che sta scaricando
- Il server invia una lista di servers aggizionali
- Il server invia automaticamente nuove fonti (i.e., IP address/port) per tutti i files che il peer sta scaricando
- Il server può rifiutare la connessione in base a sue politiche
  - » E.s.: oltre una certa soglia di peer connessi, accetto solo peer con client ID alto, poiché risparmio banda sulle callback
- *Server status* contiene informazioni sul numero di utenti e di files gestiti dal server
- *Server message*: segnalazione testuale fra server e peer



# Interazioni peer-server per ricerca file

- L'obiettivo è quello di ottenere una lista di peer sorgenti di un file
- Il client invia una query al server
- il server risponde con una lista di files che soddisfano la query
- il client sceglie il file da scaricare e chiede informazioni sui peer sorgenti
- il server comunica una lista di fonti



# Accodamento delle richieste e selezione delle parti (chunk)

- Concettualmente simile agli approcci seguiti da bit torrent BitTorrent
- Scheduling di tipo Tit-fot-Tat su base credito, dove il credito che un peer P1 “deve” nei confronti di P2 è

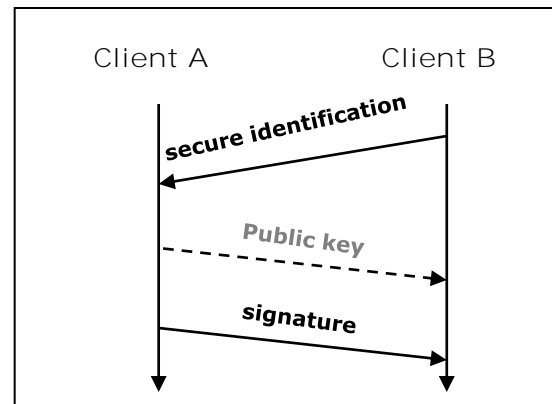
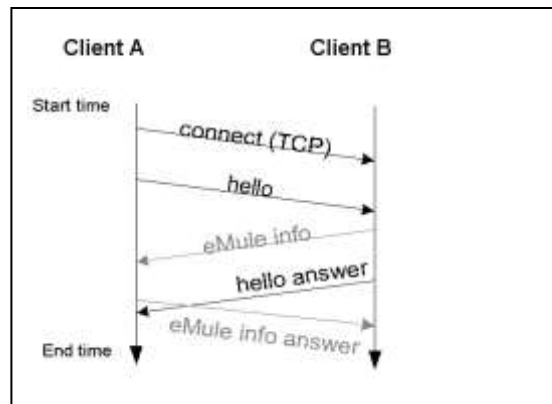
uploaded total / downloaded total (10 se downloaded total = 0)

$\sqrt{\text{uploaded total} + 2}$ , 1 se uploaded total

- Il peer P1 memorizza questa informazione di credito vs P2 per un periodo molto lungo ( e.s. 5 mesi)
- Durante la “presentazione” di un peer dowloader ad un altro peer uploader, per evitare l’impersonificazione di un peer con molti crediti (tit-for-tat) da parte di un peer maligno, si fa uso di un approccio a chiave asimmetrica che permette l’autenticazione del peer

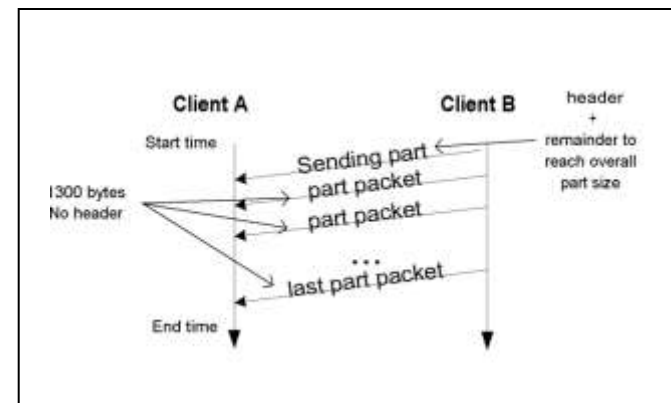
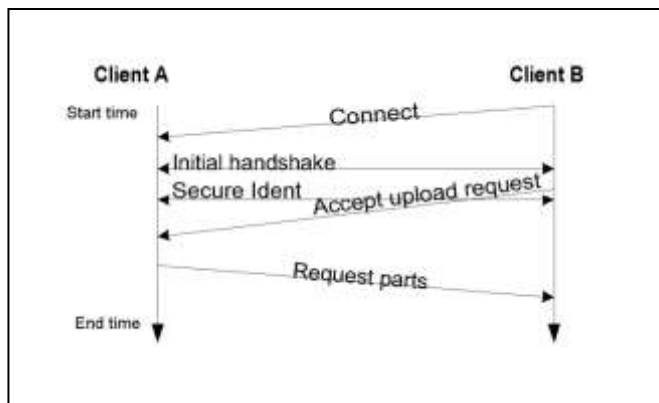
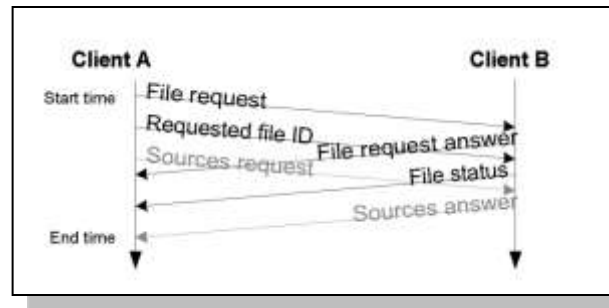
# Interazioni peer-peer

- Vi è una fase iniziale di presentazioni fra A e B, in cui A è il richiedente
- Nell'handshake iniziale B può richiedere l'autenticazione sicura di A
  - » All'installazione A si è generata una coppia chiave pubblica/privata
- L'autenticazione di A consiste dei seguenti passi:
  - » B genera un numero random *challenge* e sfida A a firmarlo con la sua chiave privata (secure identification)
  - » Se la risposta di A è decifrabile (challenge rx=challenge tx) con la chiave pubblica di A, allora A è autenticato da B
  - » Durante questa fase se A non possiede la chiave pubblica di B, allora B la trasferisce e A la memorizza (client.met)



# Interazioni peer-peer

- Dopo l'autenticazione, A richiede il file a B. B inserisce A nella sua upload queue (coda delle richieste) e quando A raggiunge la testa della upload queue (tit-for-tat) di B, B lo richiamerà...
- Al richiamo di B, A chiede a B il chunk d'interesse (Rarest First + altri approcci accessori) e B trasferirà le
- Esiste una procedura più compatta in cui alla richiesta di B, subito A abilita B all'upload. E' utilizzata quando la coda di upload di B è vuota
- Esistono delle comunicazioni aggiuntive su UDP in cui i peer si notificano il loro ranking nella coda di upload



---

# Overlay “P2P pure”

**(Architetture di rete per applicazioni P2P)**



# Overlay P2P pure – Pros & Cons

---

- Vantaggi

- » Non c'è un single point of failure
- » Completo sharing delle risorse HW

- Svantaggi

- » Lookup “lento” e possibilmente “incompleto” con probabilità auspicabilmente bassa
- » Overhead per mantenere la struttura della overlay dei metadati

---

# DHT

Overlay “P2P pure”

# DHT - keyspace

- Una DHT realizza un **database distribuito** formato da tuple **<key,value>**
- Ogni nodo DHT (i.e., dispositivo memorizzante) è identificato in modo **unico** da una key speciale denominata **identifier (ID)**
- È definita una funzione di **HASH** che trasforma la key in una stringa da m-bit
- È definita una funzione di distanza  **$d(\text{HASH}(k1), \text{HASH}(k2))$**  che restituisce la **distanza** tra due chiavi in uno spazio virtuale, denominato **keyspace**
- La tupla **<k1,value>** è memorizzata nel nodo ***i*-th** DHT la cui distanza  **$d(\text{ID}_i, \text{HASH}(k1))$**  è la più piccola possibile nei confronti degli altri nodi; i.e.  **$d(\text{ID}_i, \text{HASH}(k1)) < d(\text{ID}_j, \text{HASH}(k1))$**  per ogni ***j***
- Un nodo DHT memorizza tutte le tuple per le quali lui è il nodo DHT più vicino
- L'inserimento di un nuovo nodo DHT cambia il mapping delle chiavi sui nodi (key remapping)
- Per aumentare la **persistenza** dell'informazione, la tupla **<key,value>** può essere duplicata anche su un insieme di DHT nodi vicini al nodo DHT più vicino

# DHT overlay routing

- I nodi della DHT sono connessi fra loro da una **overlay network**
- La topologia della overlay è l'aspetto principale su cui le diverse implementazioni della metodologia DHT differiscono.
- La overlay di una DHT deve possedere la seguente proprietà: **per ogni key  $k$  un nodo DHT**
  - » **o possiede  $k$**
  - » **o ha un link overlay verso un nodo DHT più vicino di lui ad  $k$**
- Conseguentemente, la ricerca (query) del valore di una chiave  $k$  (content routing) può essere effettuata con il seguente approccio di routing:
  - » **Ad ogni passo, un nodo inoltra (forwarding) il *query-message* verso un suo vicino (i.e. altro nodo DHT con cui ha un link overlay) che è più vicino di lui alla chiave  $k$**
  - » **Quando non esiste più un tale vicino allora il nodo corrente è il più vicino e quindi possiederà lui la chiave  $k$ . Di conseguenza, risponderà *value* al nodo originario della query**

# DHT overlay routing

- La topologia della overlay DHT ha un impatto prestazionale contrastante
- Maggiore è il numero di overlay link (i.e., di vicini) che un nodo DHT possiede...
  - » maggiore è l'overhead di segnalazione per mantenere questi overlay link
  - » minore è il tempo di lookup perché il numero medio di overlay-hop per raggiungere la chiave è minore
- Molte delle attuali implementazioni DHT (CHORD, PASTRY, KADEMLIA) offrono un numero medio di overlay-hop di lookup che scala come  $O(\log(n))$  dove  $n$  è il numero di nodi DHT
- Notiamo che un hop-overlay può consistere di molti hop di livello network (path-stretch)

# DHT nelle applicazioni P2P di file sharing

- L'obiettivo è quello di scoprire quali sono i peer che posseggono il file
- La *key* è, per esempio, associata al nome di un file
- *Value* è un insieme di metadati associati al file che permettono il successivo setup di un overlay link fra peer per il trasferimento del file (e.s., dim file, IP addresses/port dei peer che lo posseggono, etc.)
- Lo identifier del nodo è determinato da una ulteriore funzione specifica delle diverse applicazioni P2P:
  - » HASH dell'indirizzo IP
  - » Numero random
  - » Ecc.
- I nodi sono connessi fra loro da una overlay su cui instradano le query sulle *key*
- Quando il nodo che possiede la *key* (non il file) riceve la query risponde direttamente all'originante con la tupla <key,value>

---

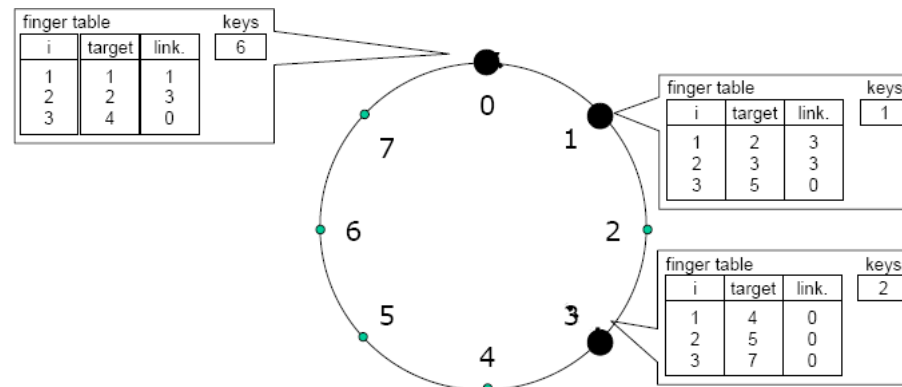
# CHORD

DHT

(ref. [http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord\\_sigcomm.pdf](http://pdos.csail.mit.edu/papers/chord:sigcomm01/chord_sigcomm.pdf) )

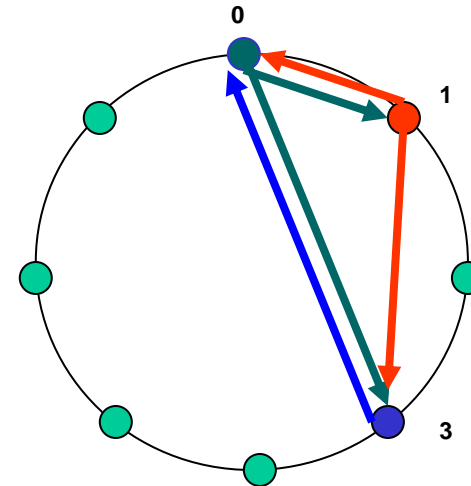
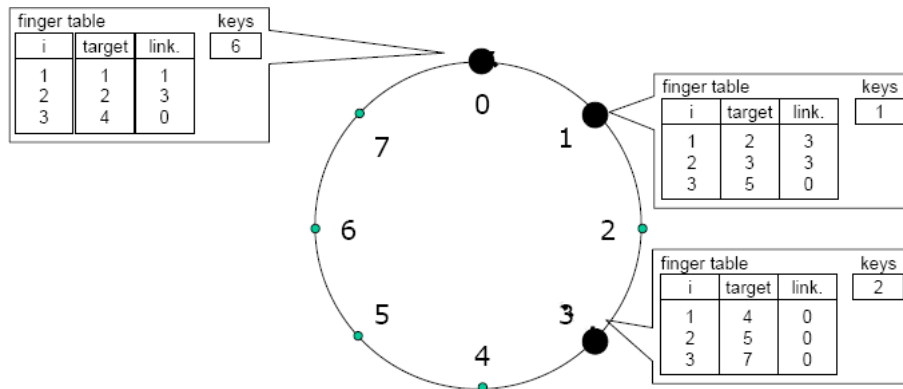
# CHORD

- Le  $HASH(key)$  hanno una lunghezza da  $m$ -bit
- Lo ID di un nodo è ottenuto mediante  $HASH(IP\ address)$
- Il keyspace è un cerchio modulo  $2^m$ , la funzione di distanza è la distanza oraria geometrica sul cerchio assumendo che due punti del keyspace vicini abbiano distanza pari ad 1
- La key  $k$  è mantenuta dal primo nodo il cui identificativo è uguale o segue  $HASH(k)$ . Se posizioniamo i nodi in base al loro ID sul keyspace, allora il nodo che possiede la chiave  $k$  è il primo nodo che segue  $HASH(k)$  in senso orario
- E.s. nel caso in cui  $m=3$ , ed ho solo tre nodi DHT con ID 0,1,3 → la key il cui  $HASH(k)=6$  è memorizzata sul nodo con ID=0



# CHORD overlay routing

- Ogni nodo ha (al massimo) un numero  $m$  di link overlay unidirezionali
- Il link  $i$ -esimo è diretto verso un altro nodo  $s$  che dista da lui almeno  $2^{i-1}$  passi sul cerchio orario
- La tabella di routing overlay di un nodo è memorizzata in un vettore di  $m$  righe chiamato *finger table*. Le informazioni sul nodo  $s$  sono memorizzate nella riga  $i$  ( *$i$ -th finger*).
- Nel caso di pochi nodi due entry della finger table possono riferirsi allo stesso nodo remoto o addirittura a se stesso
- Se un nodo ricerca la key  $k$ , se  $\text{HASH}(k)$  non è il suo ID allora inoltra il messaggio sul nodo  $i$ -th della finger table con  $\text{ID} + 2^{i-1} \leq \text{HASH}(k) < \text{ID} + 2^i$ , modulo  $2^m$ 
  - » E.s. se il nodo 0 cerca la chiave con  $\text{HASH}(k)=2$ ,  $\rightarrow$ finger  $i=1$ , inoltra la query a 1 e 1 la re-inoltra a 3 che possiede la chiave. Il nodo tre restituisce la coppia  $\langle \text{key}, \text{value} \rangle$  a 0 con uno collegamento diretto IP



# CHORD- Join and Leave Procedure

- **Un nodo entrante**
  - » Genera il suo ID
  - » Si connette ad un “known host” casuale della DHT denominato “entry point”
  - » Determina il suo successore sull’anello inviando una query del tipo  $\langle \text{key}=\text{ID} \rangle$  allo entry point. Lo entry point instrada questa query sull’anello; alla query risponderà il nodo  $\text{successor}(\text{ID})$  che è responsabile della  $\langle \text{key}=\text{ID} \rangle$ .
  - » Si fa trasferire da  $\text{succesor}(\text{ID})$  le eventuali tuple  $\langle \text{key}, \text{value} \rangle$  che ora sono diventate di sua competenza
  - » Instaura un overlay link con  $\text{succesor}(\text{ID})$  ed aggiorna la sua finger table. Ora è nell’anello e gli rimane da aggiornare i rimanenti finger
  - » Per ogni  $(\text{ID}+2^{i-1})$  invia una query sull’anello del tipo  $\langle \text{key}=\text{ID}+2^{i-1} \rangle$  il nodo che è responsabile di questa key risponde e sarà inserito nel finger  $i$ -th
- **Tutti nodi eseguono periodicamente altre procedure per verificare la presenza di altri nodi e per aggiornare il deposito delle chiavi sui nodi remoti (*resilience*)**
- **All’uscita “friendly” di un nodo, il nodo sposta le chiavi di sua responsabilità sul suo successore**

---

# **KADEMLIA (Emule KAD)**

## **DHT**

(ref. <http://xlattice.sourceforge.net/components/protocol/kademlia/specs.html>)

# KADEMLIA

- È la base della DHT KAD di Emule (il KAD ha delle estensioni)
- Sistema distribuito per la memorizzazione e la ricerca di coppie <key, value>
- Ogni nodo ha un identificatore da 160 bit
- Assegna mediante SHA (Secure Hash Algorithm) un identificatore di 160 bits alle chiavi
- Ogni coppia <key,value> viene assegnata al nodo il cui identificatore è più vicino al HASH(key), secondo una metrica definita sull'OR ESCLUSIVO
  - » Ad esempio la distanza tra l'identificatore ID1 = 0100 e la HASH(key) = 0111 è la seguente  $D(0100,0111) = 0100 \text{ XOR } 0111 = 0011 = 3$ ; i.e., è una stringa binaria che contiene 1 dove i bit sono diversi
- Kademlia appartiene alla famiglia delle DHT basate su prefix-matching
- Ad ogni hop la query di lookup viene inoltrata da un nodo N1 verso un nodo N2 il cui identificatore è più vicino alla chiave (secondo la metrica definita)

# KADEMLIA Overlay routing

- Per ogni  $0 < i < 160$  ogni nodo mantiene una **lista** (e non uno come in CHORD) di triple  $\langle \text{IP address, UDP port, Node ID} \rangle$  per i nodi che distano da se stesso tra  $2^i$  e  $2^{i+1}$ . La lista è denominata ***k-bucket*** e può contenere al massimo  $k$  elementi ( $k=20$  per Emule)
- Ogni *k-bucket* è ordinato temporalmente in base all'ultimo contatto avuto con la entry. L'ultimo contattato è in testa alla lista
- Quando un nodo riceve un qualsiasi messaggio kademlia, aggiorna il *k-bucket* relativo inserendoci il nodo *source ID* del messaggio (***autolearning***)
  - » Se il *k-bucket* associato a Source ID è pieno effettua un PING su tutti gli elementi del *k-bucket*. Se qualcuno non risponde, lo elimina ed inserisce il nuovo *source ID*; altrimenti non inserisce il nuovo *source ID*

# KADEMLIA lookup

- I messaggi protocollari sono PING, STORE (key,value), Find Node (ID), Find Value (Key)
  - » STORE(key,value): individua i  $k$  nodi più vicini alla chiave e memorizza su di essi la coppia  $\langle \text{key}, \text{value} \rangle$  (ridondanza)
  - » FIND\_NODE (ID): dato un identificatore logico ID appartenente allo spazio logico di Kademlia, restituisce  $k$  triple (Indirizzo IP, porta UDP, ID logico) corrispondenti ai  $k$  nodi più vicini al nodo
  - » FIND\_VALUE (ID): si comporta come la FIND\_NODE e restituisce il valore associato alla chiave
- Supponiamo che il nodo N1 ricerchi la chiave  $k$ 
  - » N1 ricerca all'interno della propria routing table un'entrata che punti ad un nodo N2 identificato da un ID la cui configurazione binaria coincide almeno nei  $b$  bits più significativi con la configurazione di  $\text{HASH}(k)$  (Emule  $b=1$ )
  - » La query viene inoltrata verso N2 e N2 ricerca all'interno della propria routing table un'entrata che punti ad un nodo N3 identificato da un ID la cui configurazione binaria coincide almeno nei  $2b$  bits più significativi con la configurazione di  $\text{HASH}(k)$
  - » Il lookup prosegue ad ogni passo un 'match' di lunghezza sempre maggiore tra  $\text{HASH}(k)$  e l'identificatore del nodo
  - » Il lookup termina non appena non è possibile individuare all'interno della tabella di routing un match di lunghezza maggiore. Il nodo più vicino alla chiave è stato raggiunto e questo risponde alla sorgente del lookup con la coppia  $\langle k, \text{value} \rangle$
- Parallel Routing:
  - » una query per una chiave  $k$  ricevuta al passo  $h$ -esimo da un nodo viene inoltrata in parallelo ad  $\alpha$  nodi prelevati dal  $k$ -bucket che contiene nodi con almeno i  $2h$  bits più significativi uguali ad  $\text{HASH}(k)$
- Strategie di routing
  - » Routing Iterativo (Emule): il nodo che invia una richiesta di lookup coordina l'intero processo di ricerca ad ogni passo, un nodo invia una richiesta di lookup ed attende una risposta la risposta ricevuta indica quale è il successivo passo di routing
  - » Routing Ricorsivo: la richiesta di look up viene inoltrata automaticamente da un peer al successivo Kademlia utilizza un routing iterativo

# KADEMLIA JOIN & Leave

---

- Per effettuare un join ad un'overlay Kademlia, un nodo  $n$  deve conoscere qualche nodo  $w$  appartenente all'overlay
- Procedura di join:
  - »  $n$  inserisce  $w$  nel  $k$ -bucket relativo
  - » effettua un lookup di se stesso (i.e. FIND\_NODE) attraverso  $w$
  - » in questo modo trova alcuni (max  $k$ ) nodi vicini e poi lavora in autolearning
- L'uscita di un nodo non richiede operazioni aggiuntive
- Periodicamente (e.s. ogni ora) le coppie  $\langle \text{key}, \text{value} \rangle$  sono ripubblicate

---

# Overlay “P2P ibride”

(Architetture di rete per applicazioni P2P)

# Overlay P2P ibride

- Uguali alle P2P pure con la seguente differenza: la overlay dei metadati è formata da una MESH fra super-peer (nodi con elevato HW) ed una stella di peer su un super-peer
- Solo i super-peer gestiscono e condividono i metadati loro e dei peer connessi
- Rispetto alle P2P pure hanno un minor tempo di lookup ma anche un minore sharing HW
- Skype ha una architettura simile, con un server addizionale di autenticazione

