Peer 2 Peer
Massively Multiuser VEs

Patrice Torguet
torguet@irit.fr
Université Paul Sabatier

Schedule part 1 P2P

- Introduction
- Problems of P2P overlays
- Protocol operations
- Bootstrapping the P2P overlay
- Unstructured P2P
- Structured P2P and DHTs

Introduction

- The precursor: Napster
  - File sharing using P2P transfers
  - But: one server indexed everything (peers and files) => subject to failure and... lawsuit
- Current applications:
  - File sharing (eMule, Bittorrent...)
  - Video conferencing (Skype...)
  - Distributed computing (SETI@home, bioinformatics...)
- Future applications:
  - Sensor networks
  - Distributed file systems
  - MMVEs

Introduction

- Characteristics:
  - fully distributed (no server) => Full P2P
  - heterogenous resources (P2P ≠ Grid computing)
  - highly dynamic => Peers can fail
- Similar to:
  - Ad-hoc networking
  - Grid computing

Problems of P2P overlays

- Main problem: churning
  - Peers come and go
  - Need to maintain connectivity
- Other problems:
  - Security (see for example: TOR)
  - Efficient routing (P2P overlay structure ≠ Internet structure)
  - Firewall crossing

Protocol operations

- Network oriented
  - Join: a peer joins the P2P overlay
  - Leave: a peer leaves the overlay
  - Lookup/query: messages used to find peers (lookup) or data (query)
- Data oriented
  - Put: new data is available on the overlay
  - Get: retrieve data
  - Delete: previously available data is now unavailable

Bootstrapping the P2P overlay

- Need to find an existing peer to connect to
  - Reliable peers: well known peers that work 24/7
  - Cached list: keep a cache of previously known peers IPs
  - IP broadcast/multicast: often only working on a LAN
  - Host cache service: independent well known name server
  - Manual: user supplied list of IP addresses (e.g. found on the web)

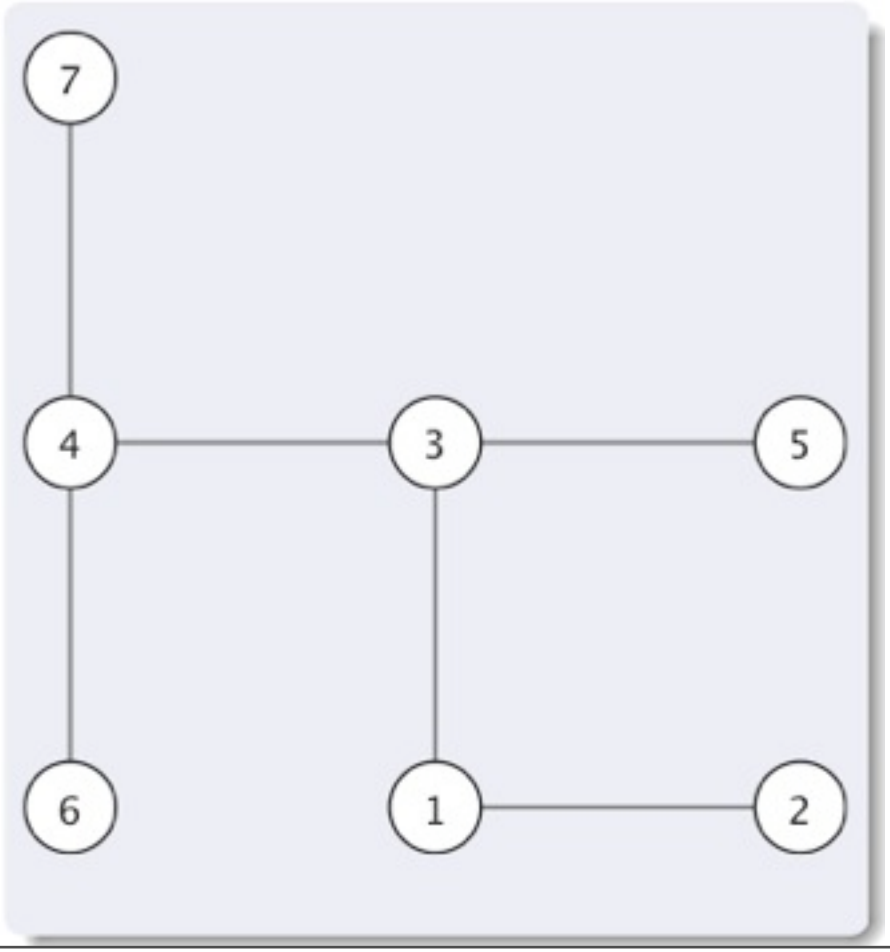
Unstructured P2P

- Simple approach
  - random graph of peers
  - use flooding or random exploration to find data/peers
- Can be very inefficient
- How do you maintain connectivity ?

Gnutella (*newtella*)

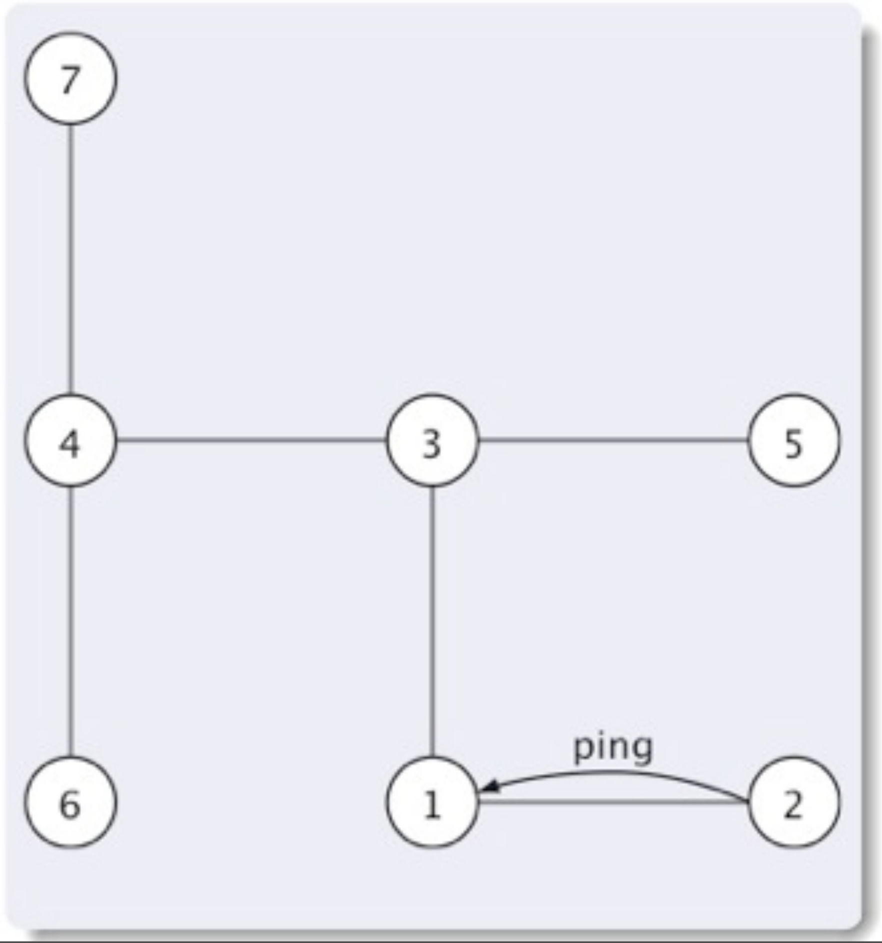
- Connection using a list of well-known peers
- Random connectivity using TCP
- Messages have unique IDs and the ids and origins are kept in local caches (to avoid re-propagation and to allow back propagation)
- PING/PONG messages used to find other peers
  - PING sent to current neighbors and forwarded
  - PONG are back propagated and contain IP addresses

Gnutella: example overlay



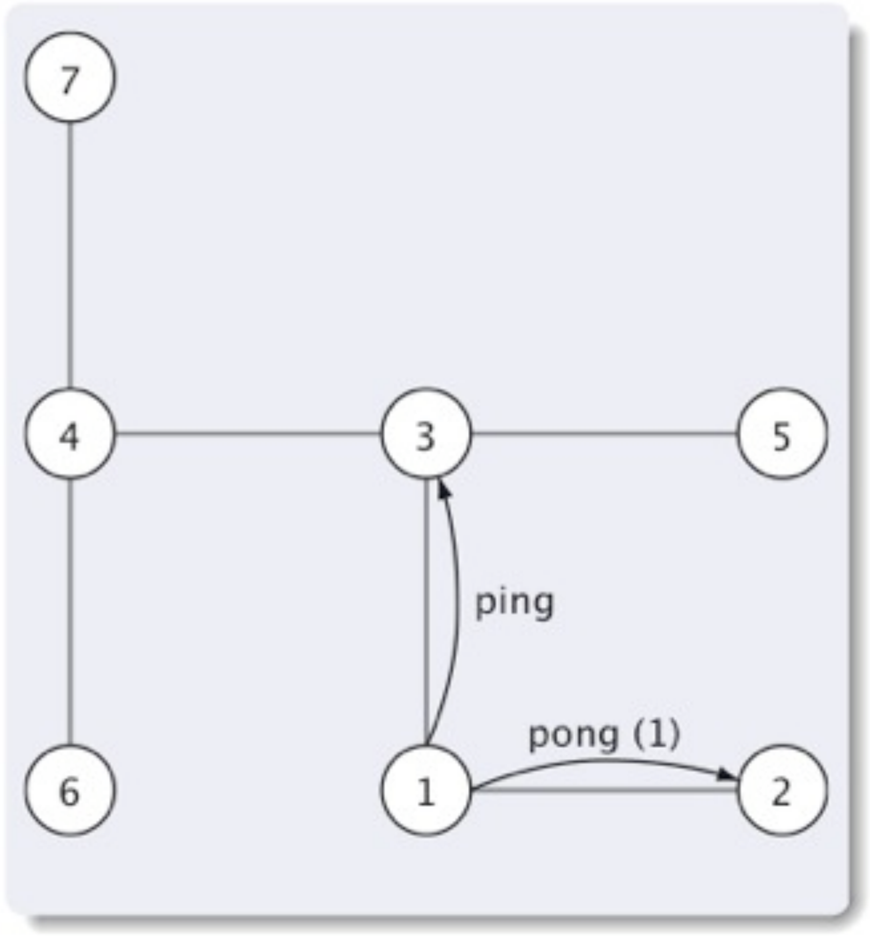
Figures courtesy of Aaron Harwood NICTA

Gnutella: PING/PONG



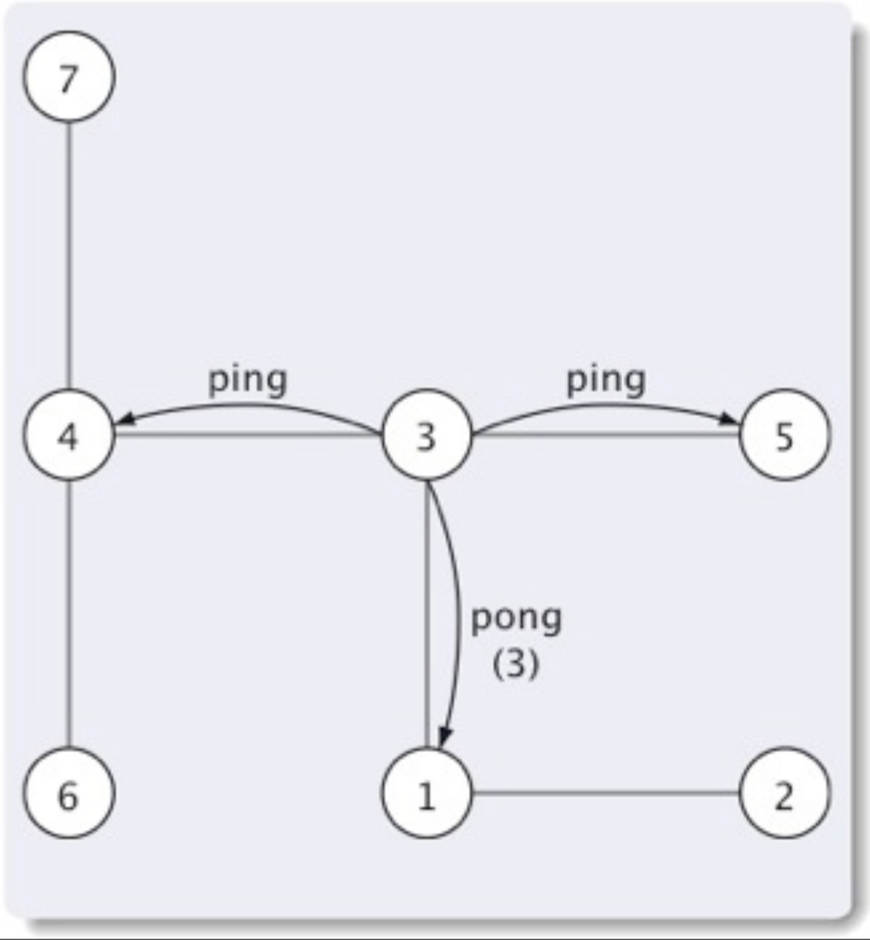
Figures courtesy of Aaron Harwood NICTA

Gnutella: PING/PONG



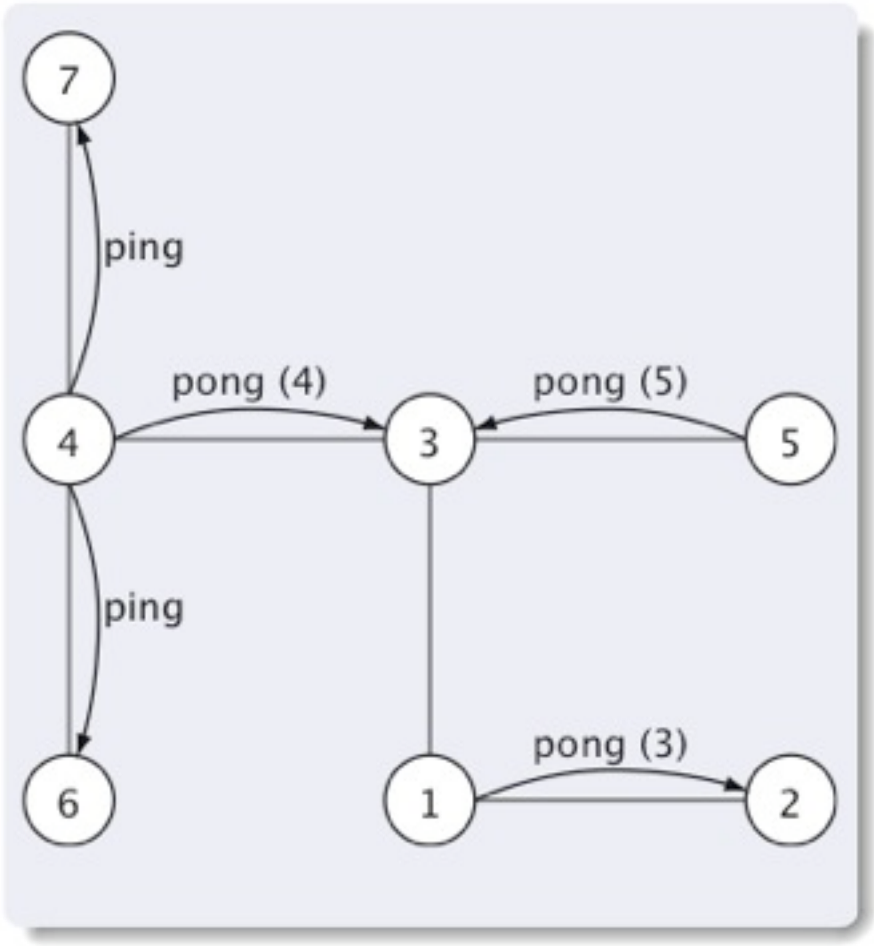
Figures courtesy of Aaron Harwood NICTA

Gnutella: PING/PONG



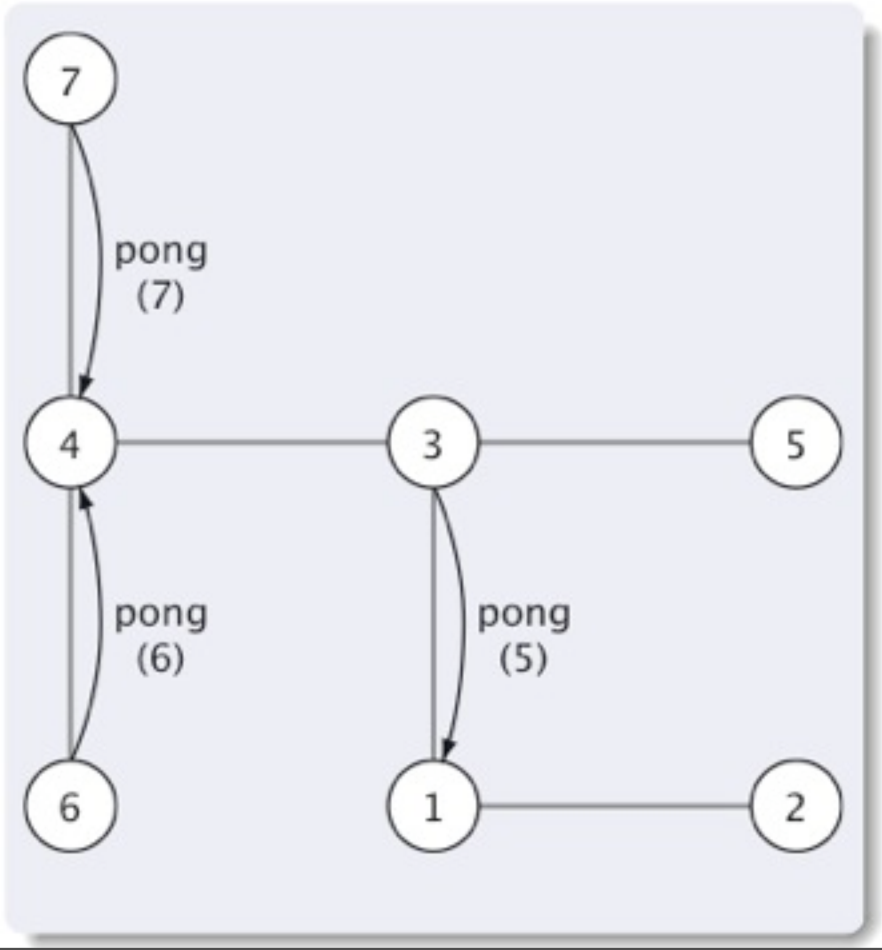
Figures courtesy of Aaron Harwood NICTA

Gnutella: PING/PONG



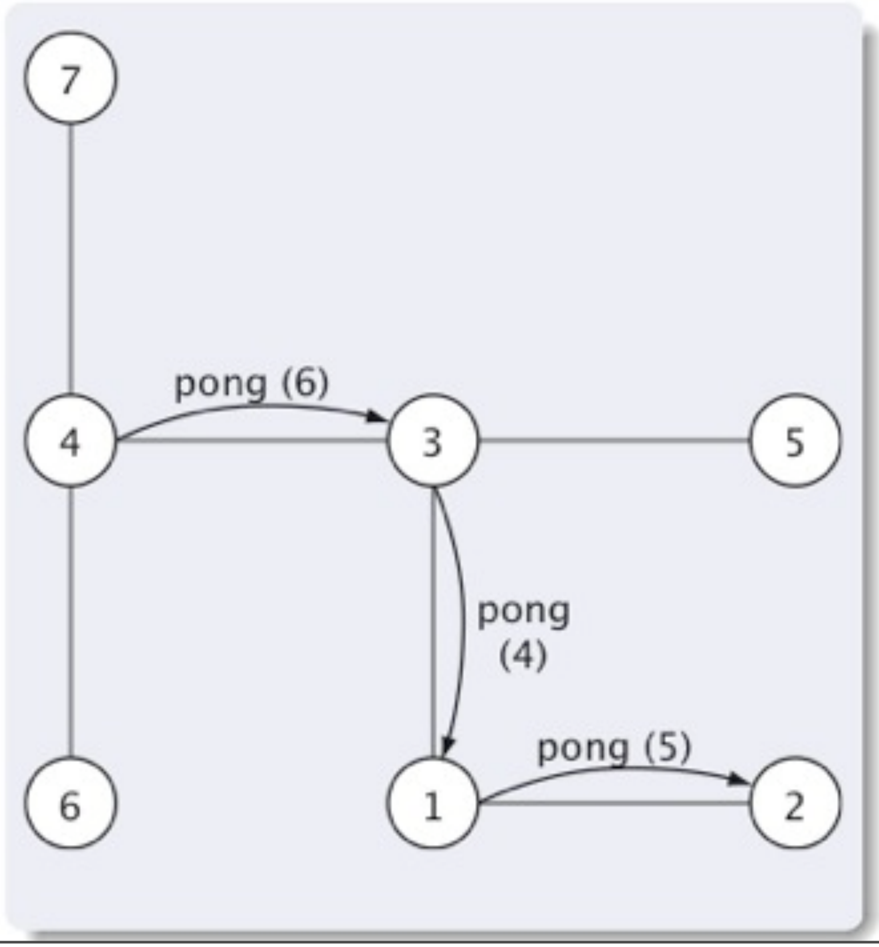
Figures courtesy of Aaron Harwood NICTA

Gnutella: PING/PONG



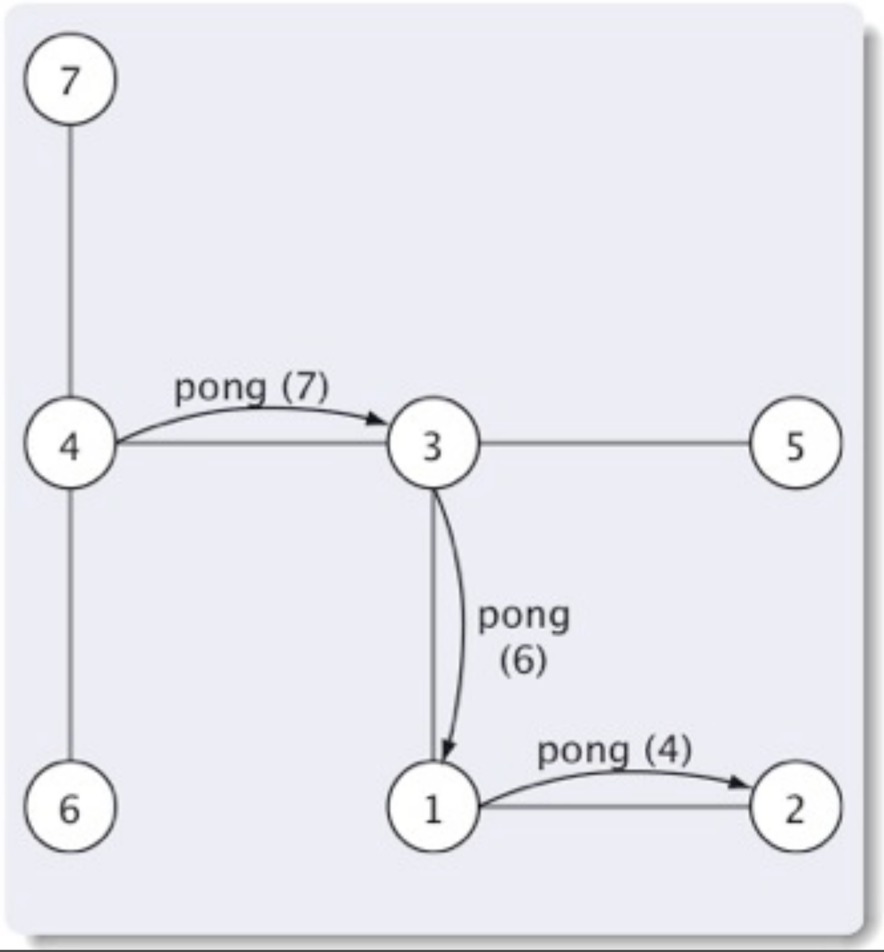
Figures courtesy of Aaron Harwood NICTA

Gnutella: PING/PONG



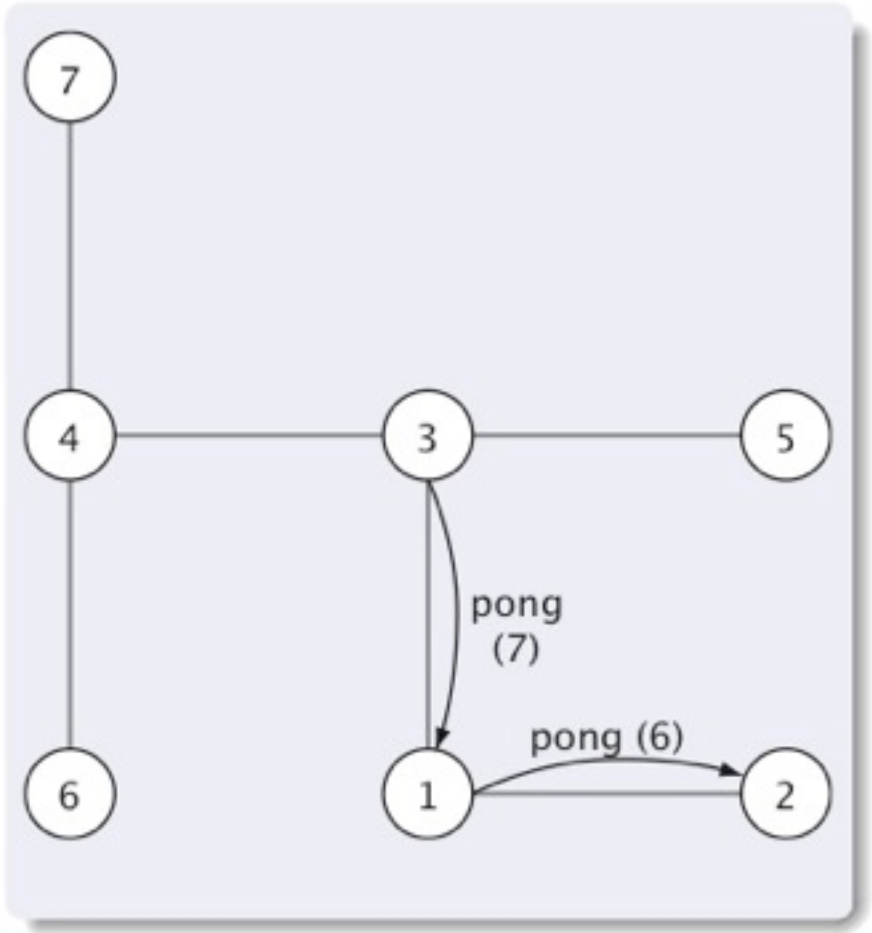
Figures courtesy of Aaron Harwood NICTA

Gnutella: PING/PONG



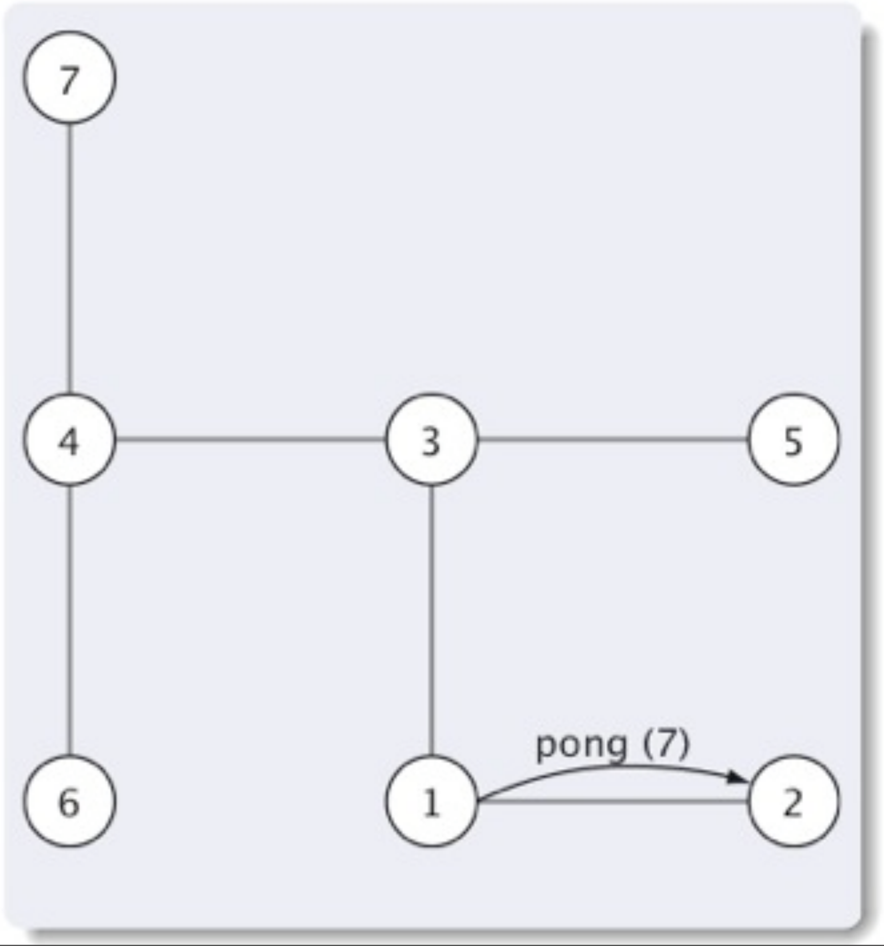
Figures courtesy of Aaron Harwood NICTA

Gnutella: PING/PONG



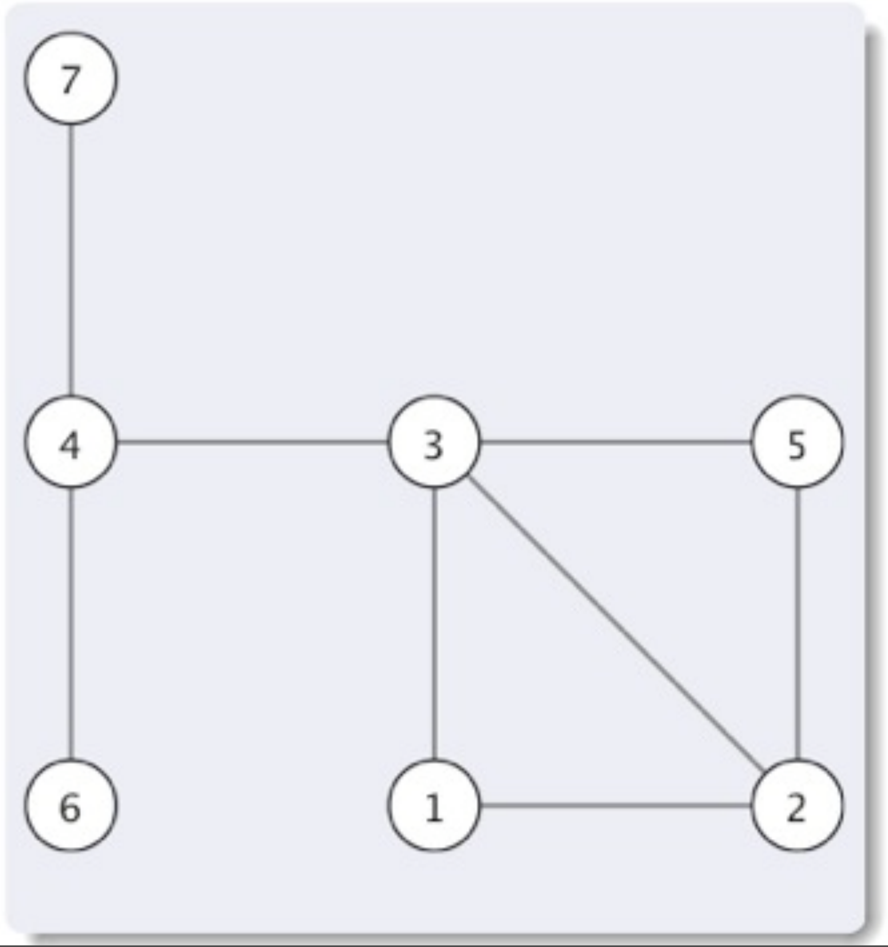
Figures courtesy of Aaron Harwood NICTA

Gnutella: PING/PONG



Figures courtesy of Aaron Harwood NICTA

Gnutella: new connections

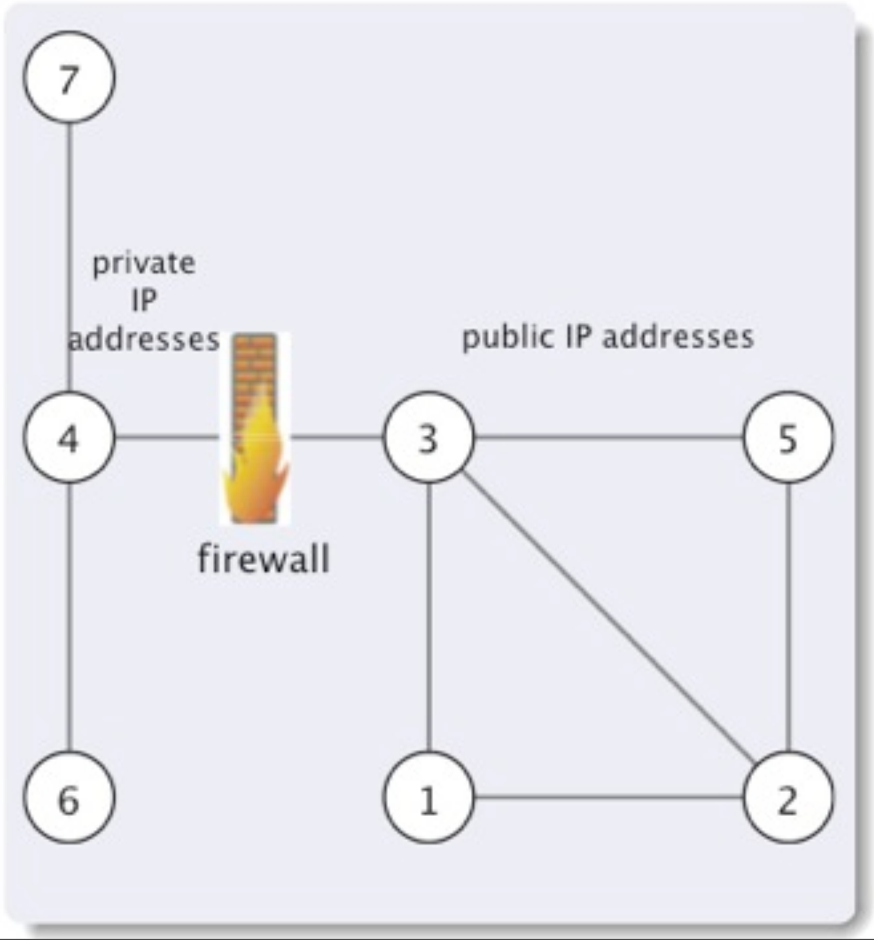


Figures courtesy of Aaron Harwood NICTA

Gnutella (*newtella*)

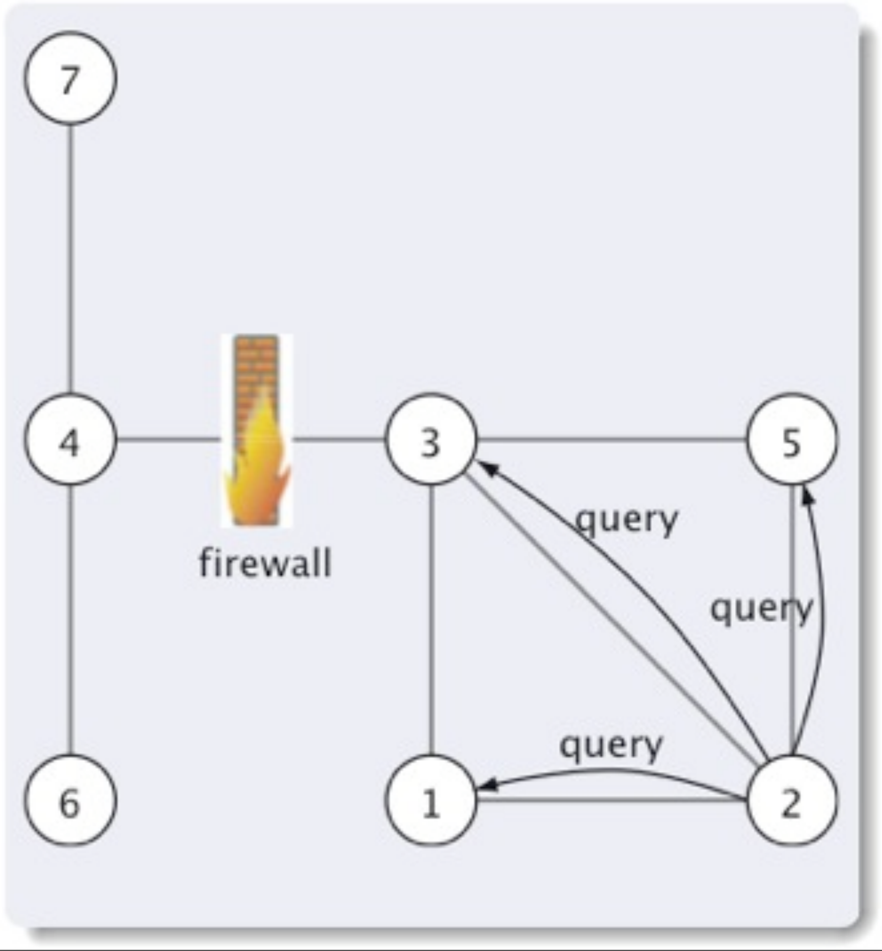
- QUERY messages are flooded
- HIT messages are back propagated
- GET messages are used to directly download data
- PUSH messages are used to require data to be sent back (used to cross firewalls)

Gnutella: QHGP



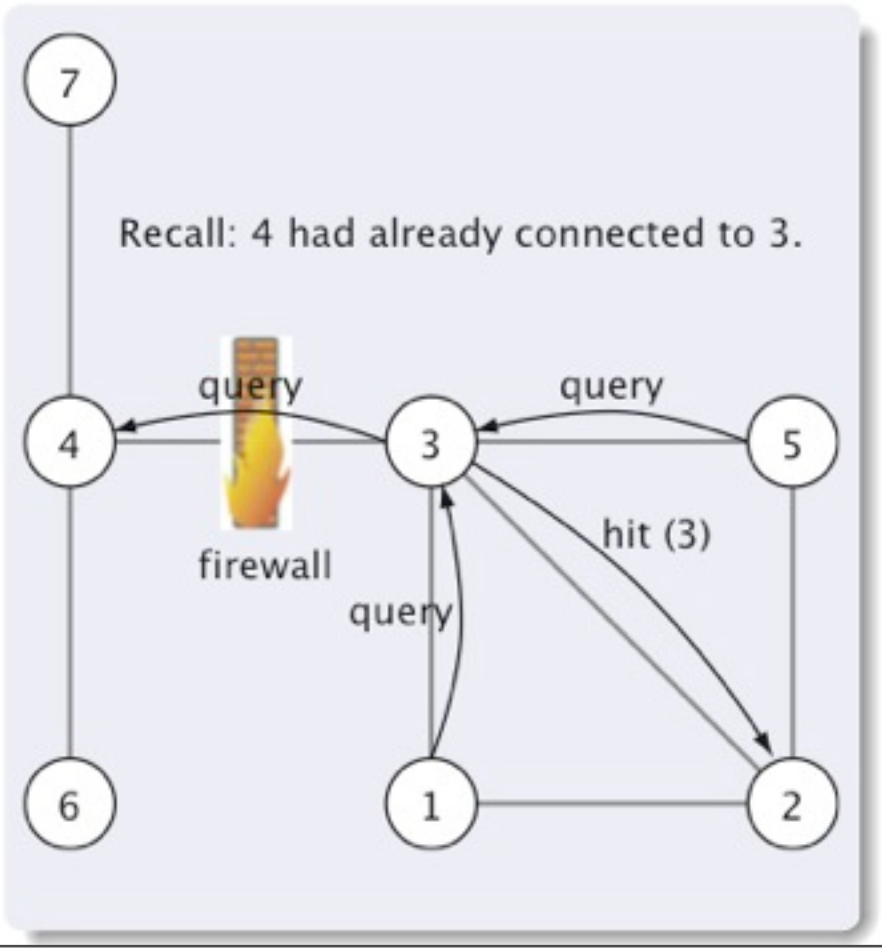
Figures courtesy of Aaron Harwood NICTA

Gnutella: Queries are flooded



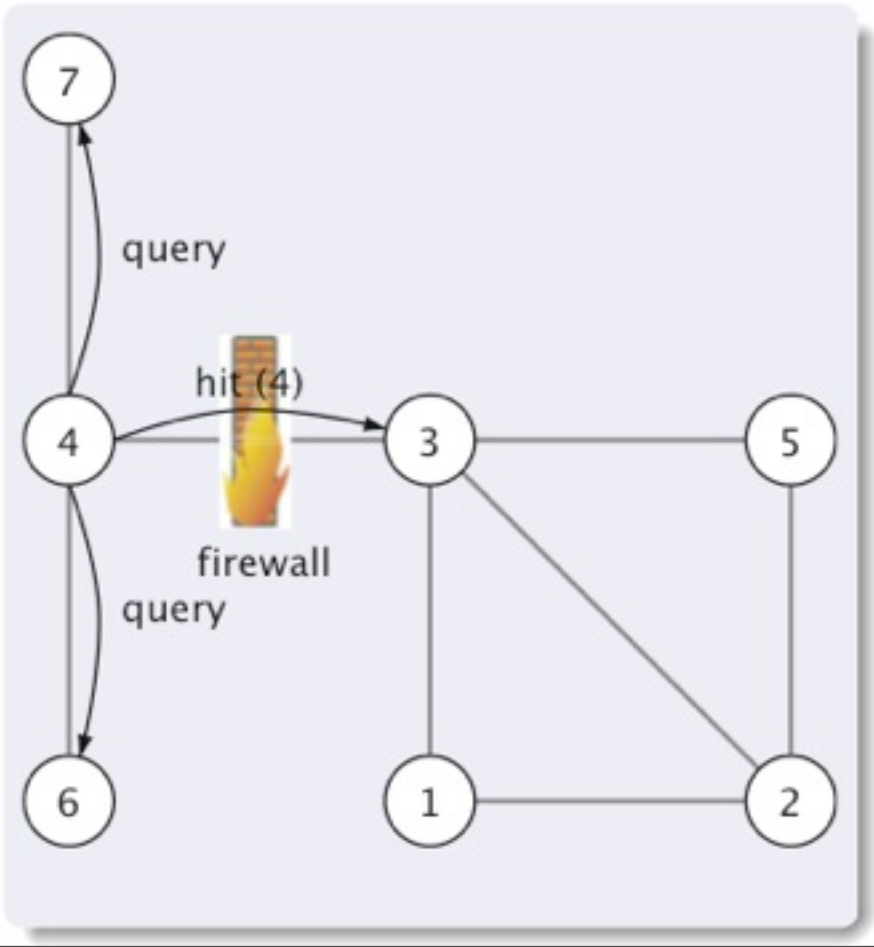
Figures courtesy of Aaron Harwood NICTA

Gnutella: Floods cross firewalls



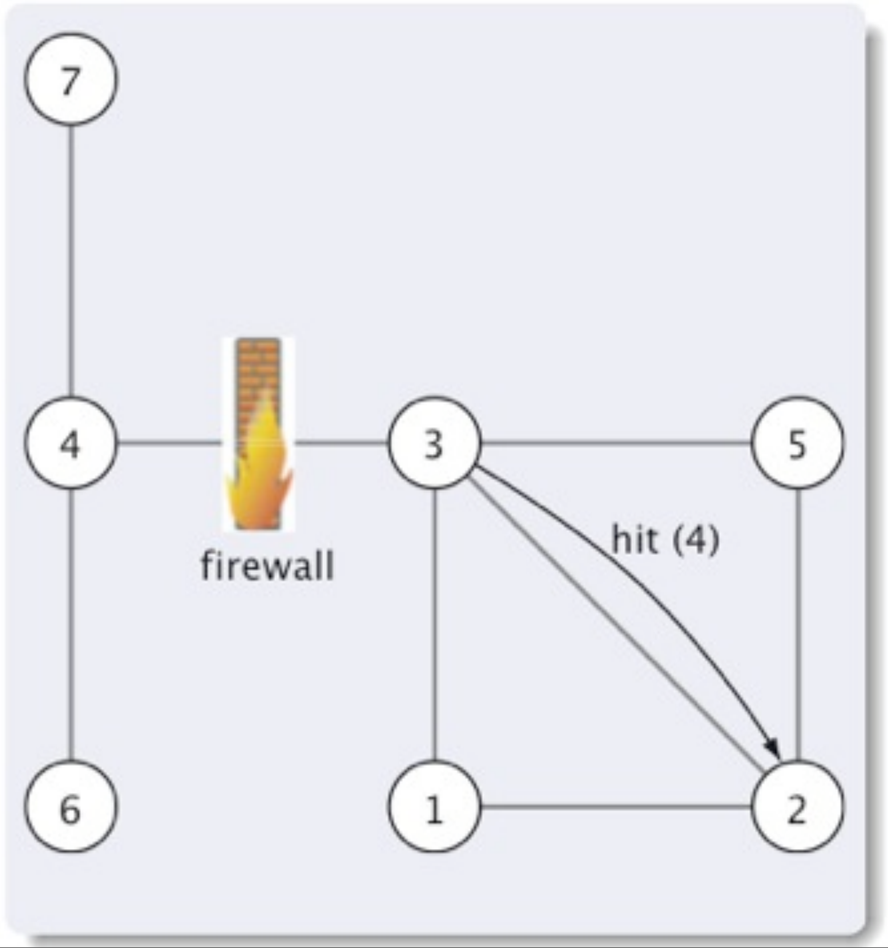
Figures courtesy of Aaron Harwood NICTA

Gnutella: Hits come back from some nodes



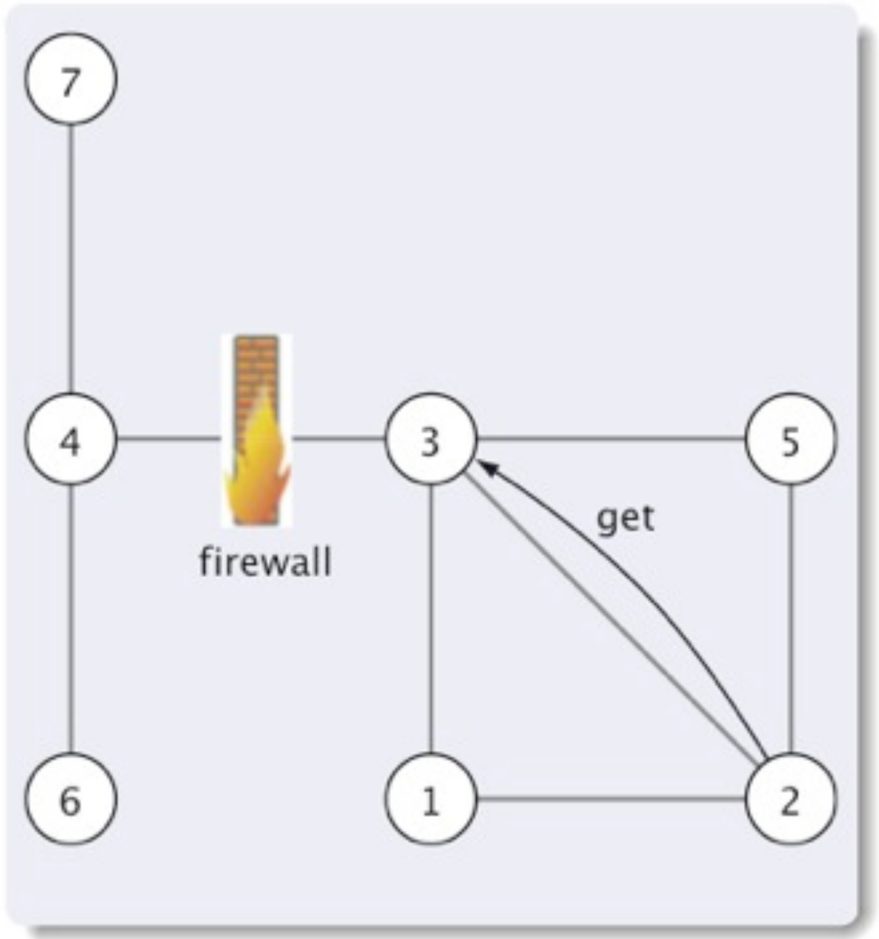
Figures courtesy of Aaron Harwood NICTA

Gnutella: Hits are back-propagated



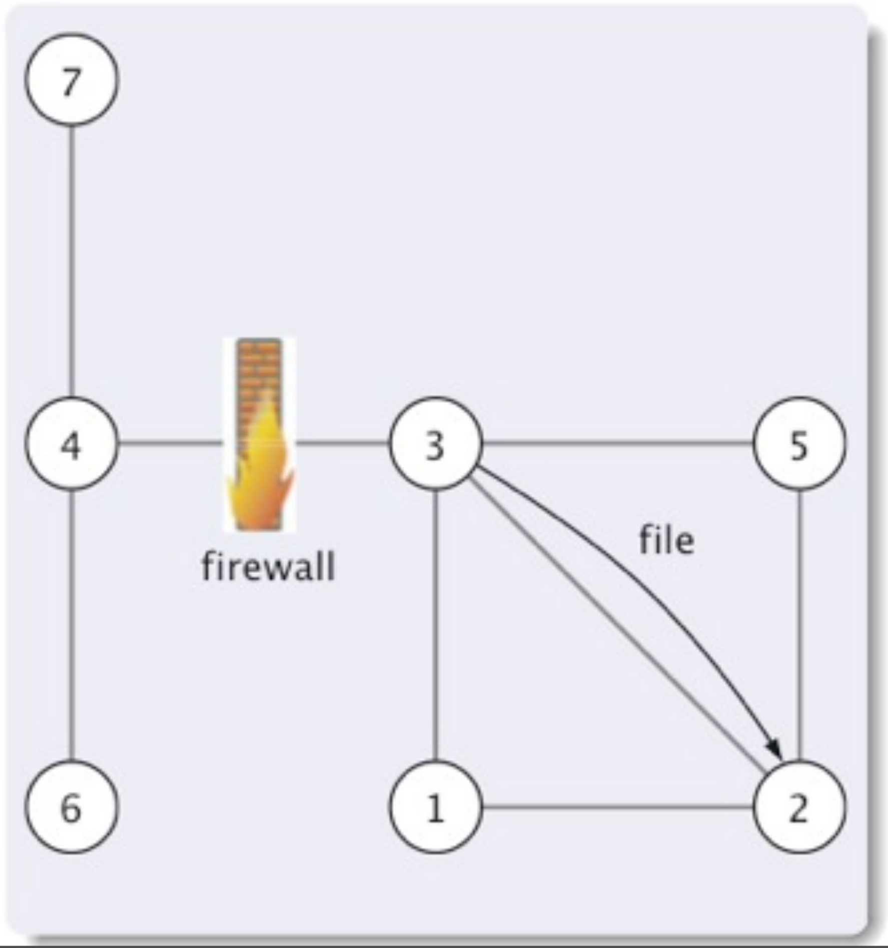
Figures courtesy of Aaron Harwood NICTA

Gnutella: Get to public IP works



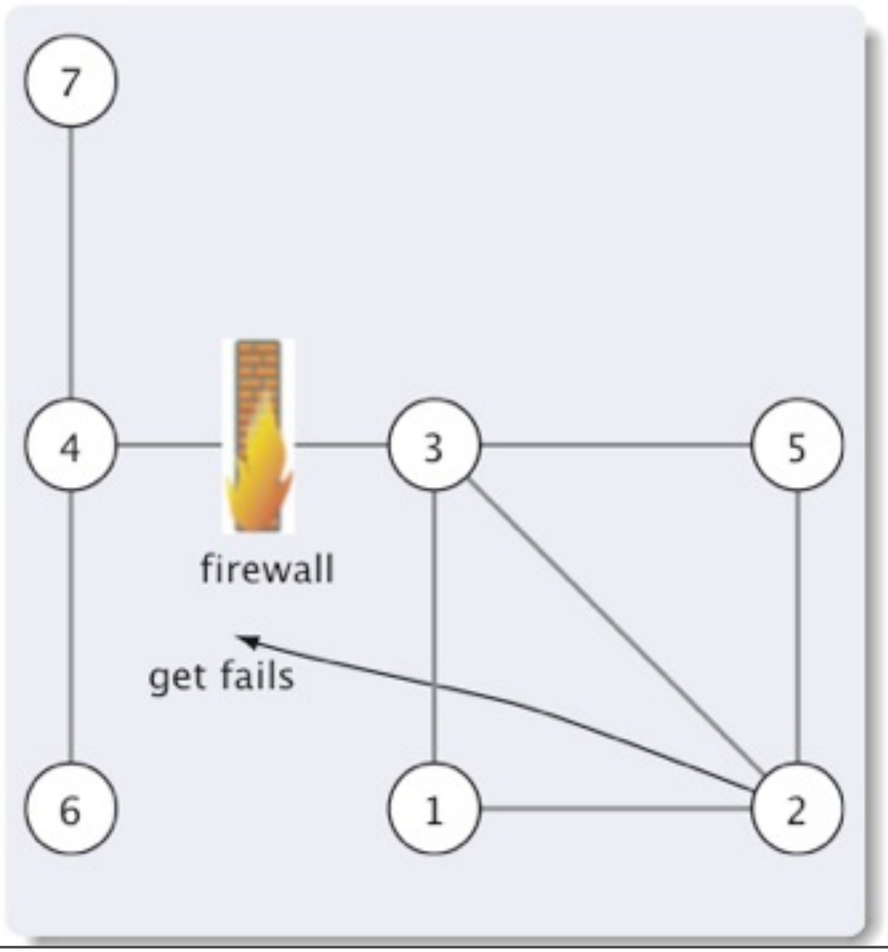
Figures courtesy of Aaron Harwood NICTA

Gnutella: Get to public IP works



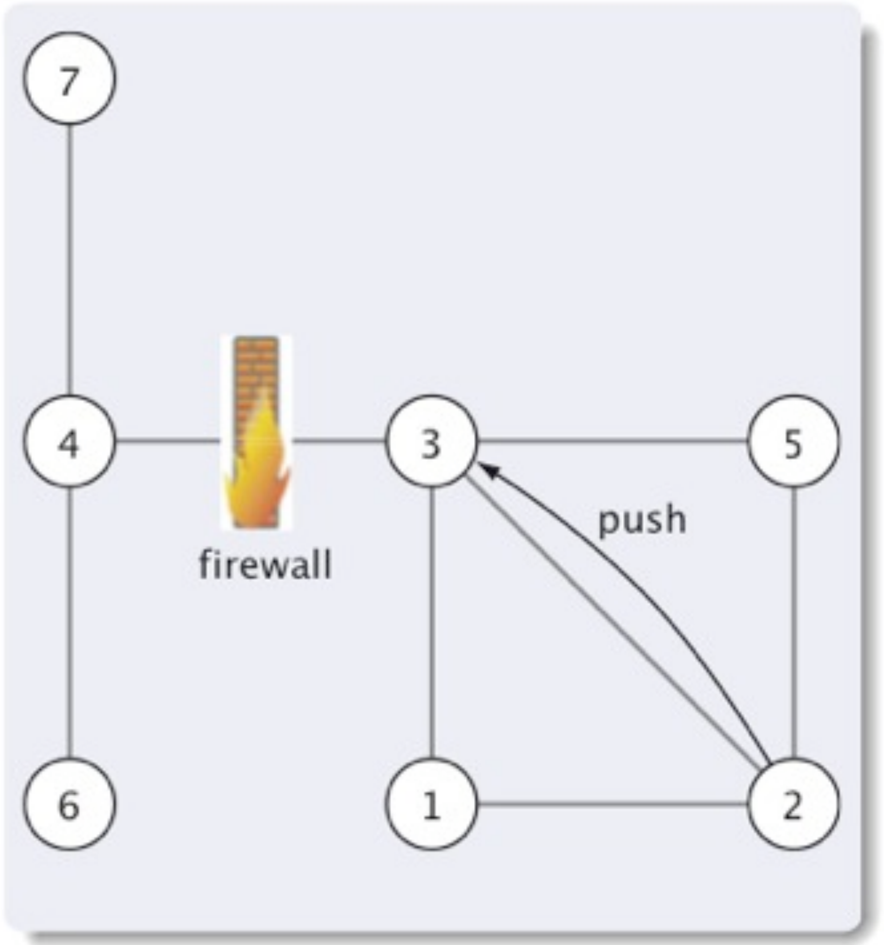
Figures courtesy of Aaron Harwood NICTA

Gnutella: Get to private IP fails



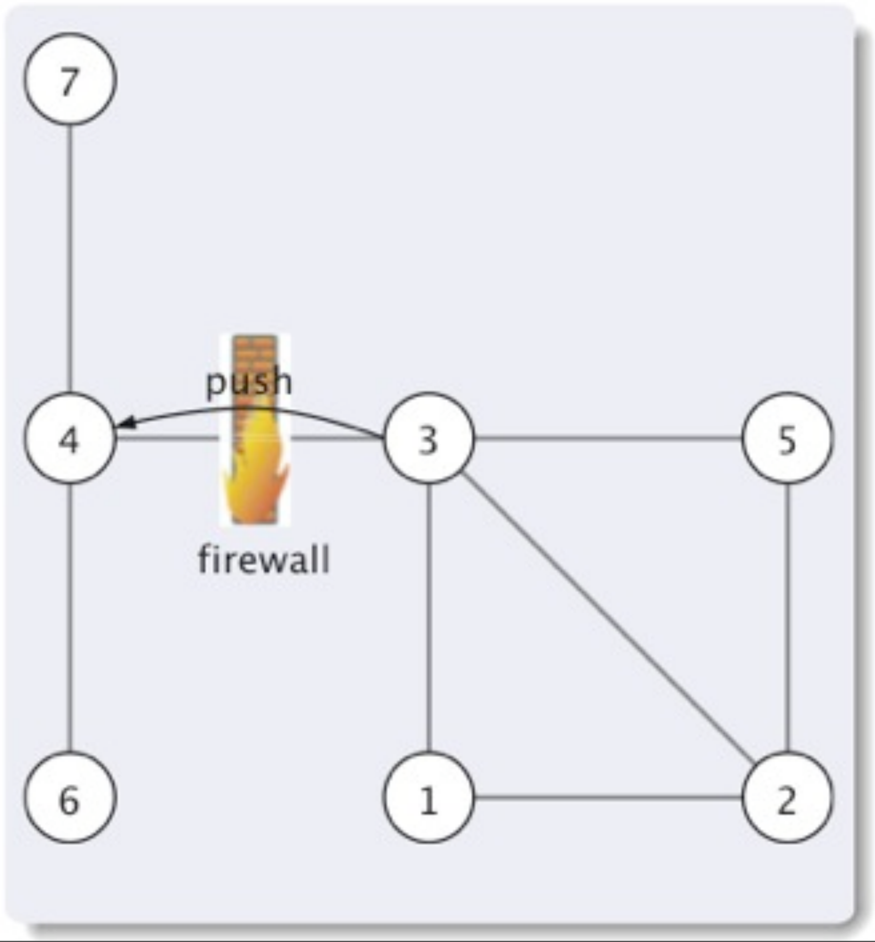
Figures courtesy of Aaron Harwood NICTA

Gnutella: Need to use PUSH



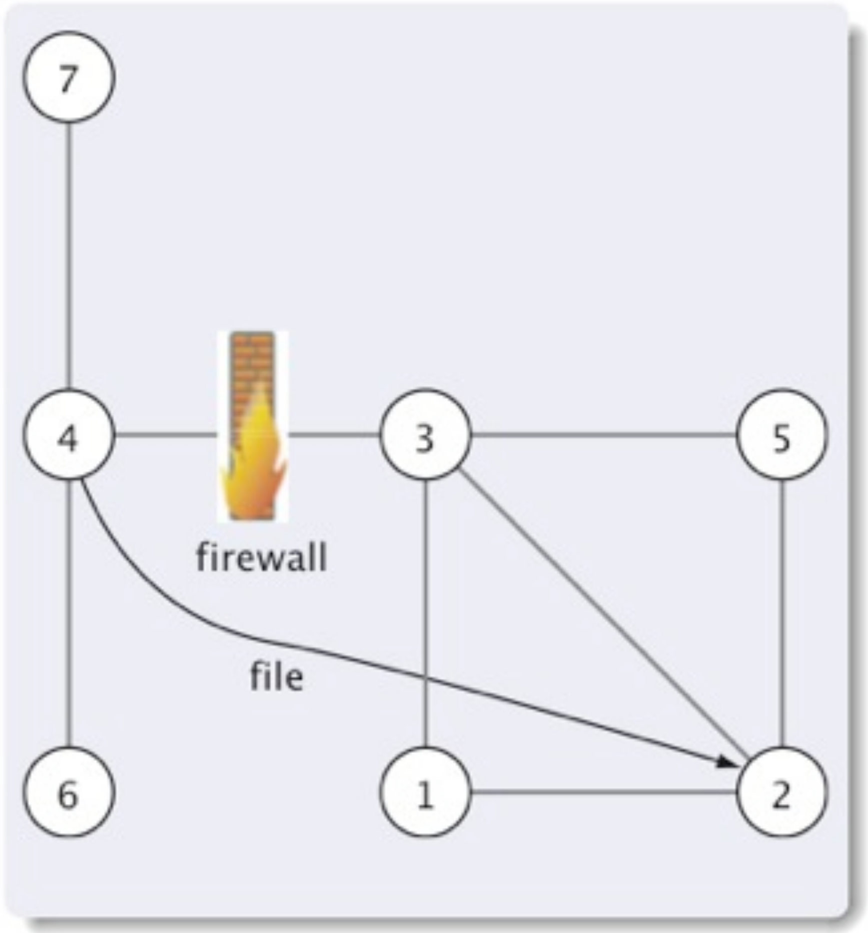
Figures courtesy of Aaron Harwood NICTA

Gnutella: Need to use PUSH



Figures courtesy of Aaron Harwood NICTA

Gnutella: Need to use PUSH



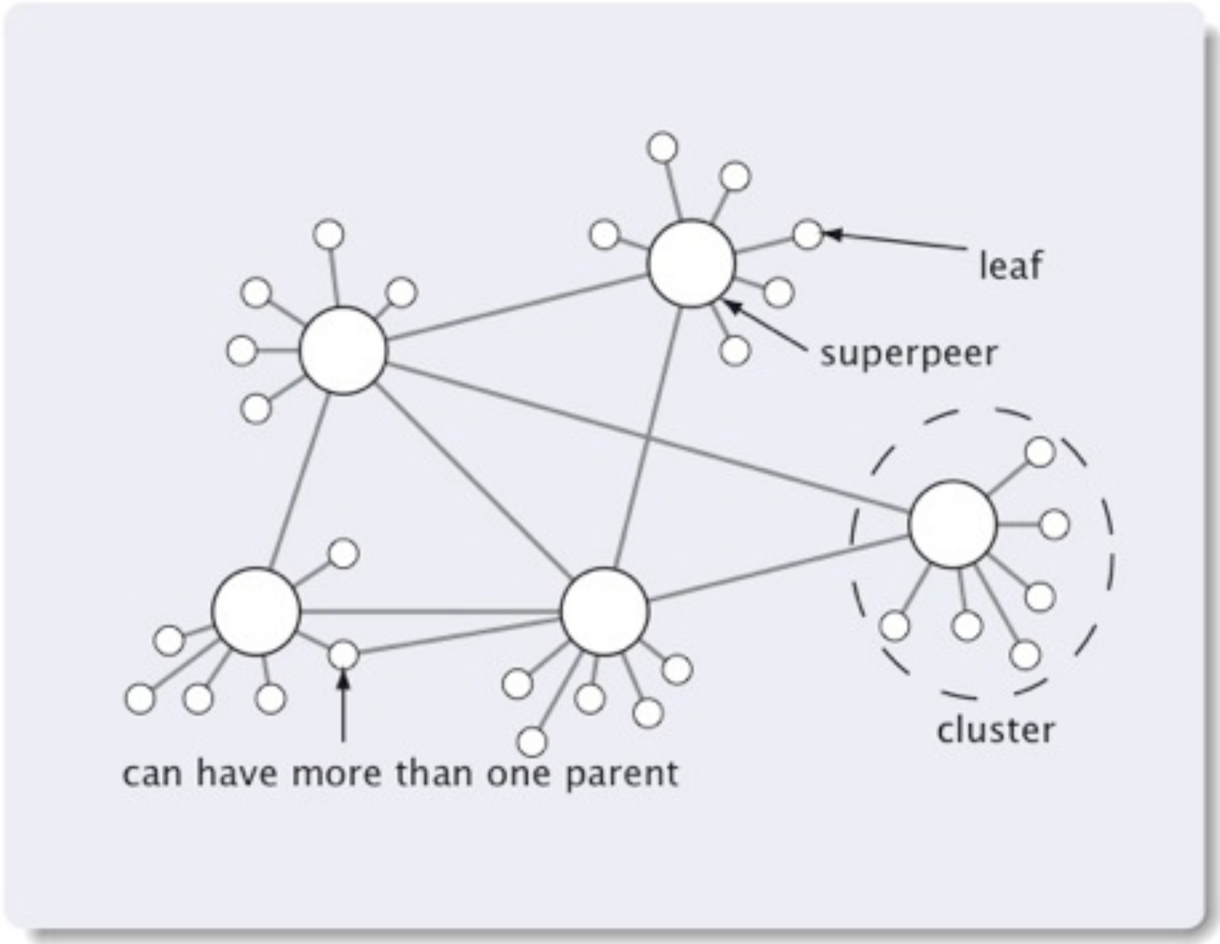
Figures courtesy of Aaron Harwood NICTA

Gnutella (*newtella*)

- To optimize Gnutella super-peers (ultra peers) have been introduced
  - They have more bandwidth
  - They index files for other nodes (and manage searching on behalf of them)
  - A peer can be connected to several super-peers (used for super-peer fault tolerance)

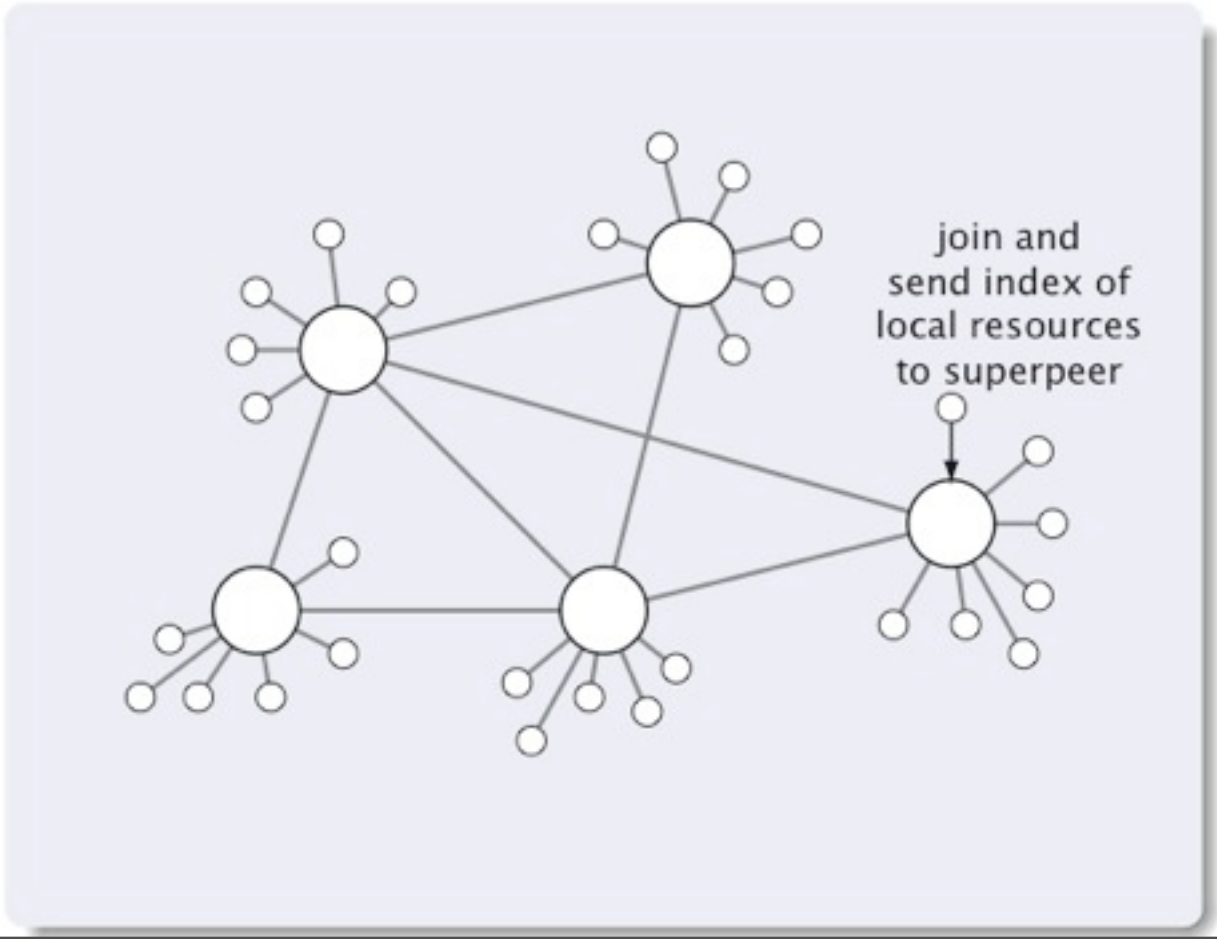
  - But flooding still exists between super-peers
  - Super-peer promotion is done only at connection

Gnutella: Super-peers



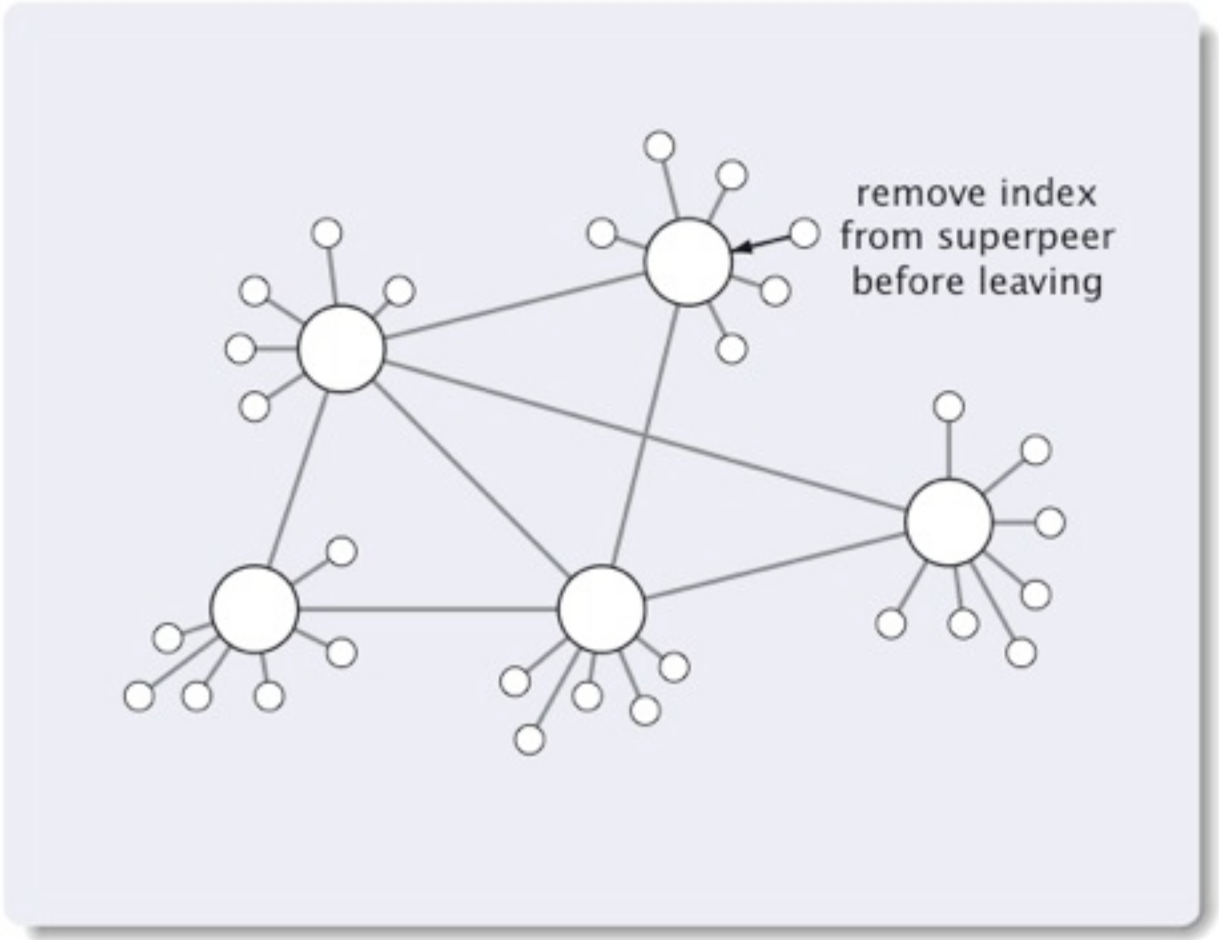
Figures courtesy of Aaron Harwood NICTA

Gnutella: Super-peers



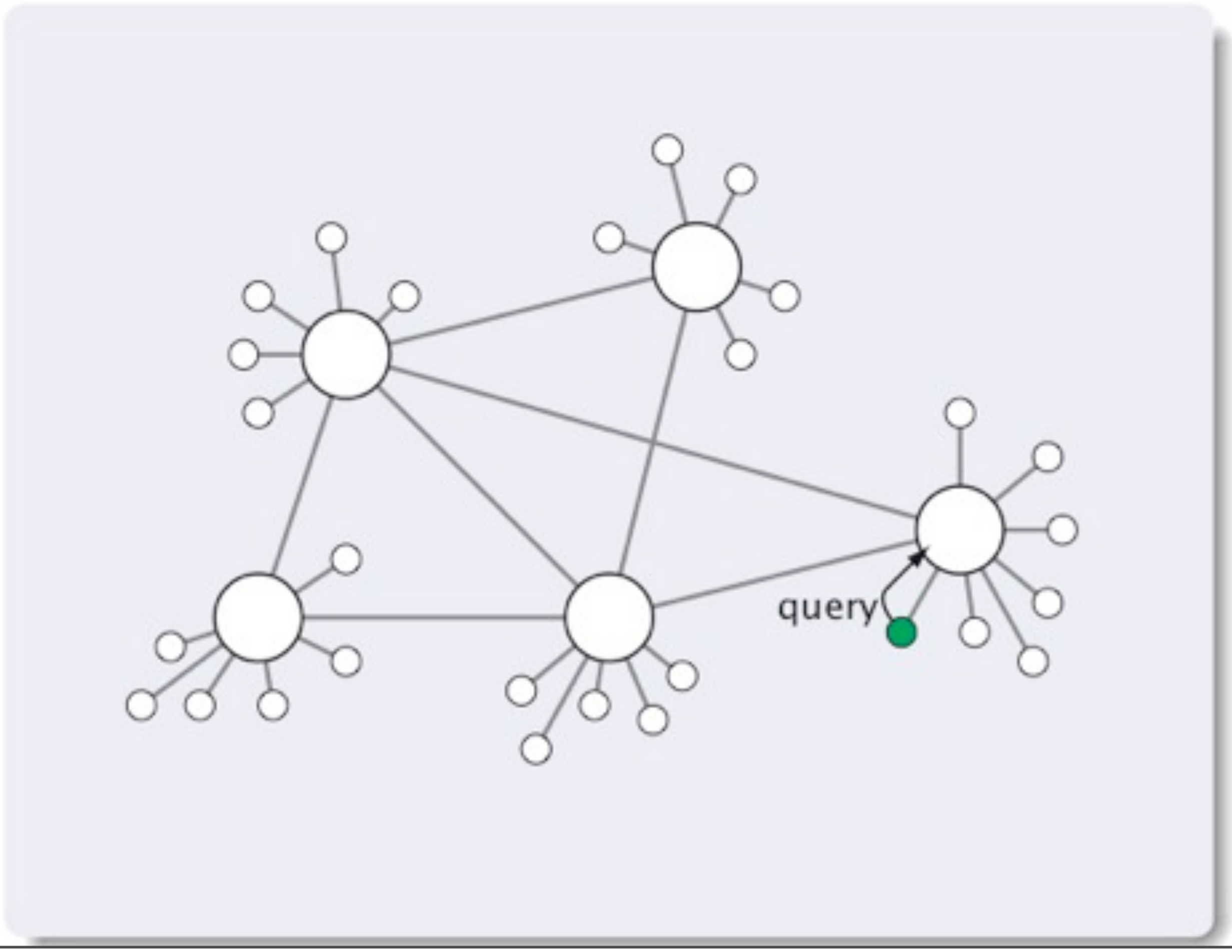
Figures
courtesy of
Aaron Harwood
NICTA

Gnutella: Super-peers



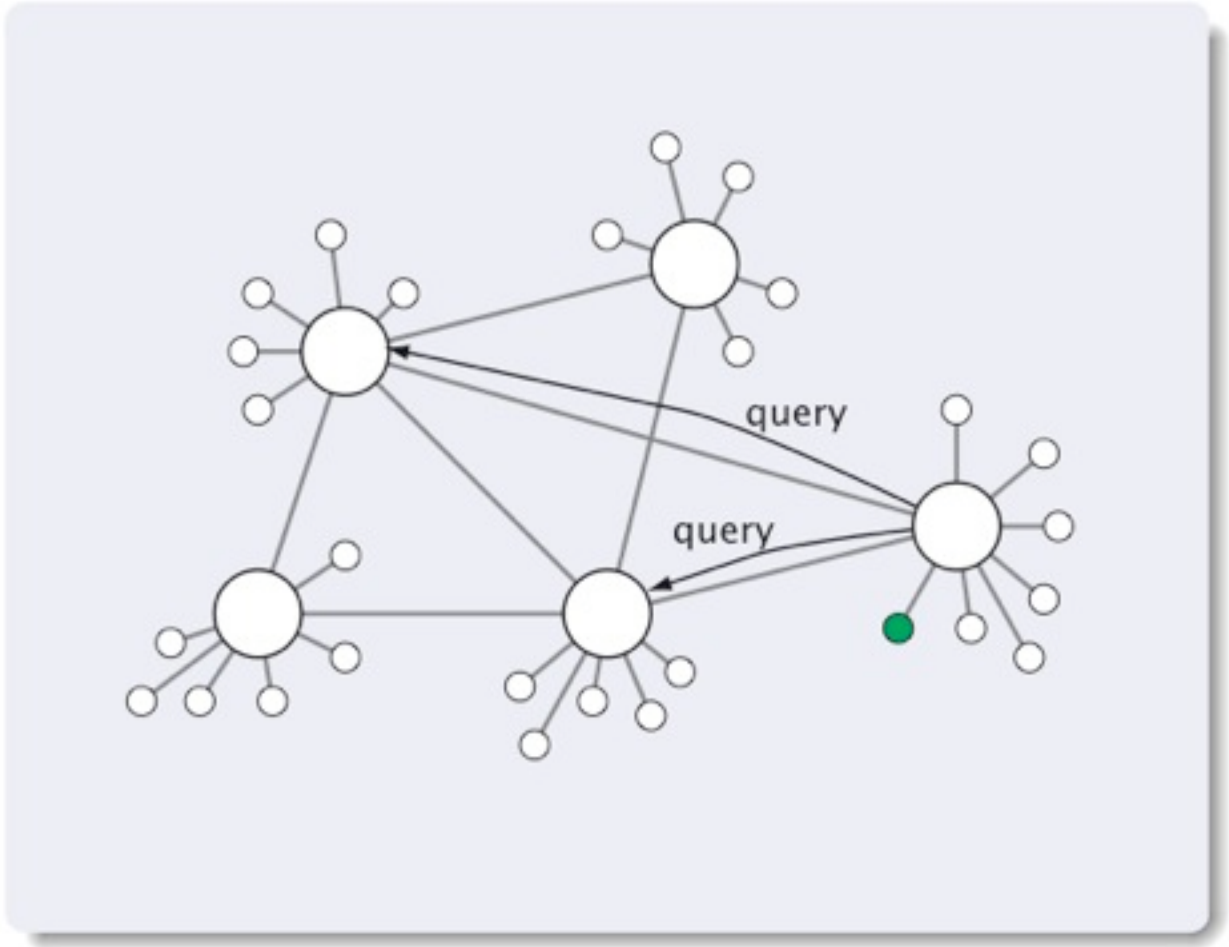
Figures courtesy of Aaron Harwood NICTA

Gnutella: Super-peers



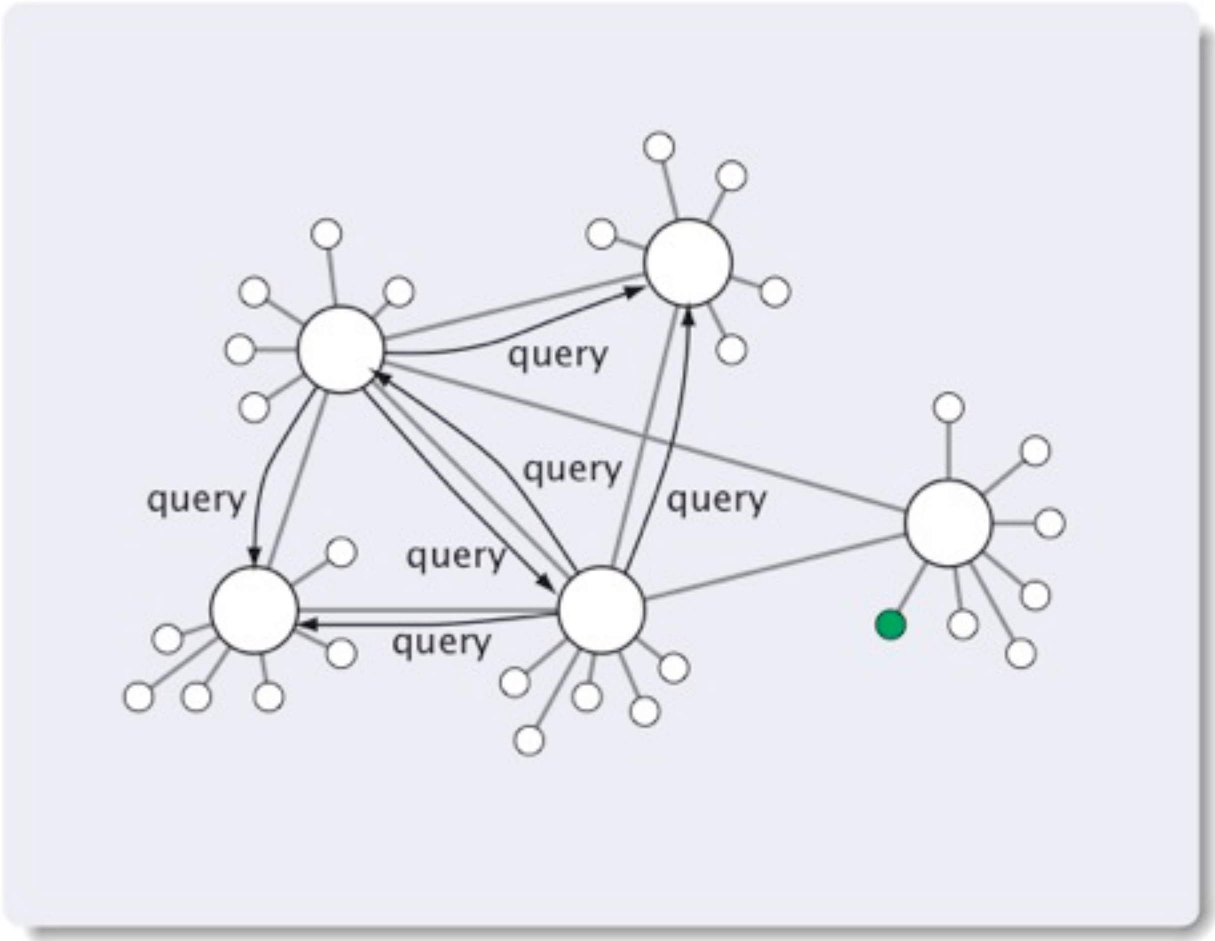
Figures courtesy of Aaron Harwood NICTA

Gnutella: Super-peers



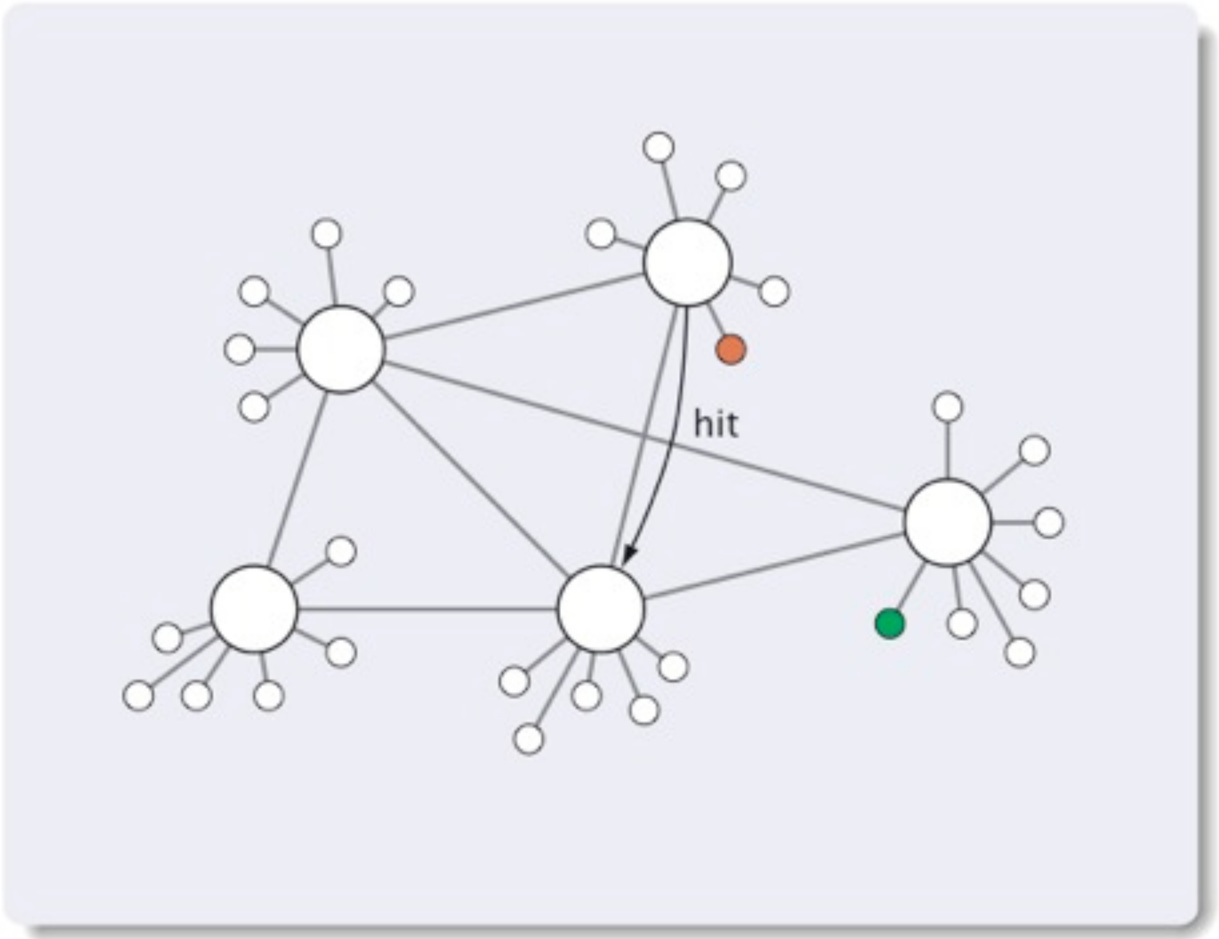
Figures courtesy of Aaron Harwood NICTA

Gnutella: Super-peers



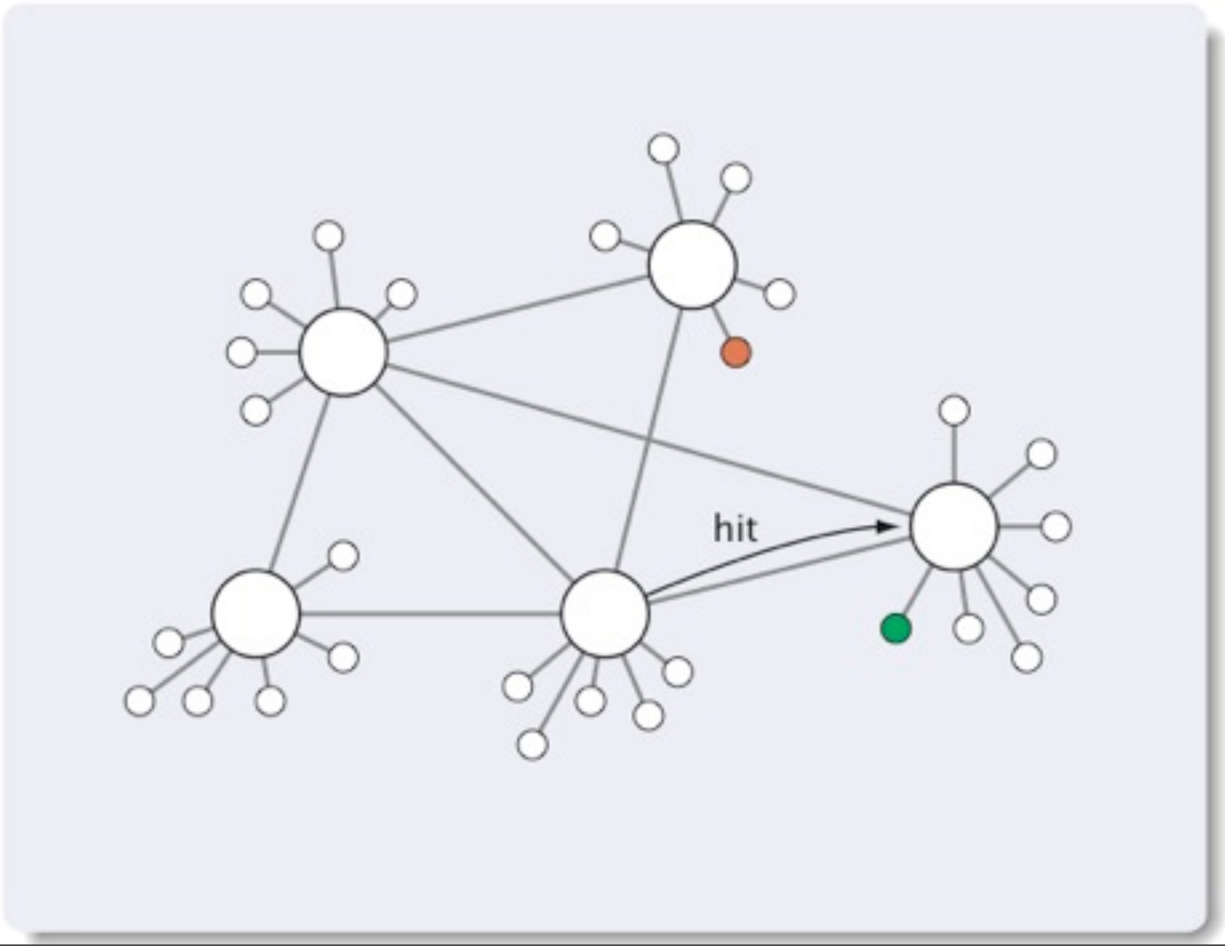
Figures courtesy of Aaron Harwood NICTA

Gnutella: Super-peers



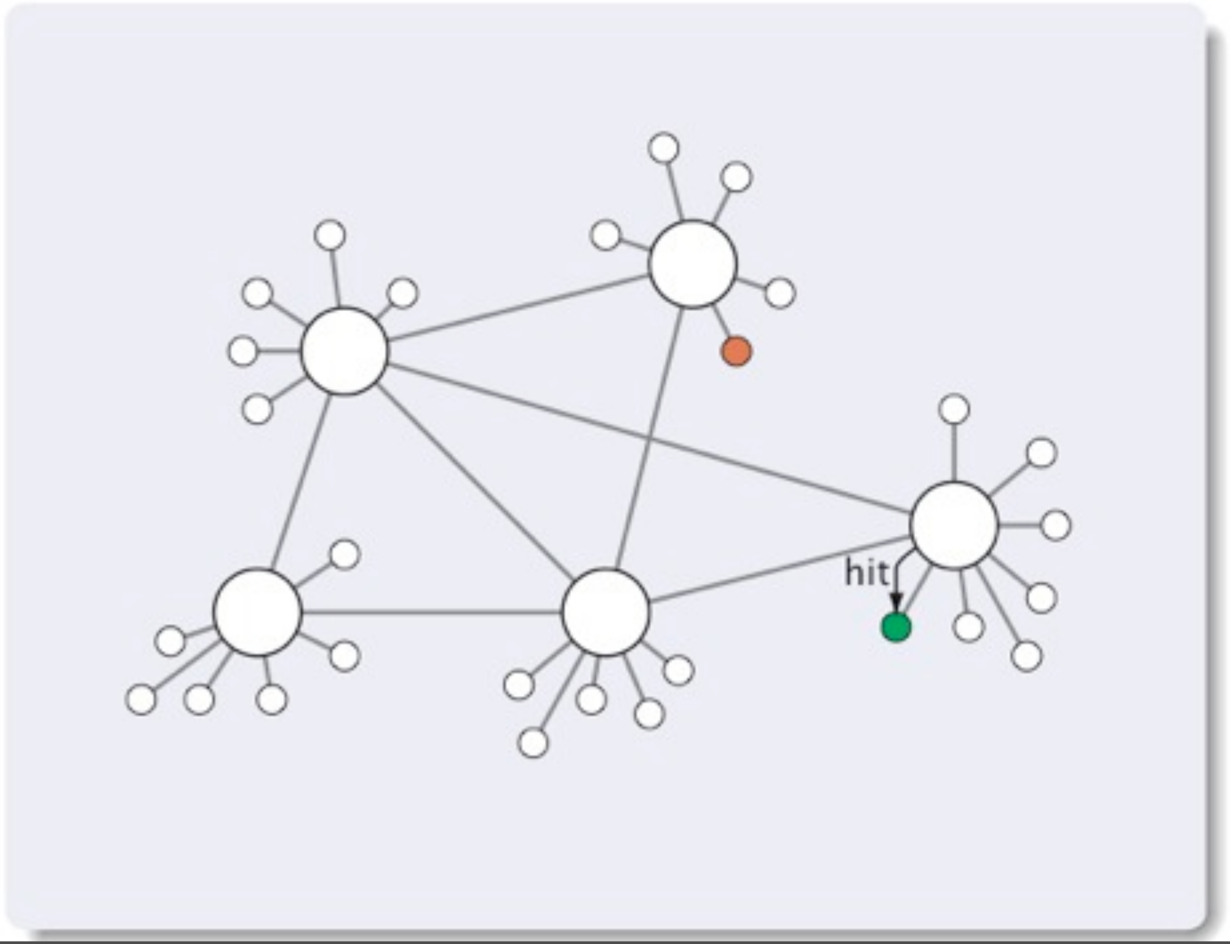
Figures courtesy of Aaron Harwood NICTA

Gnutella: Super-peers



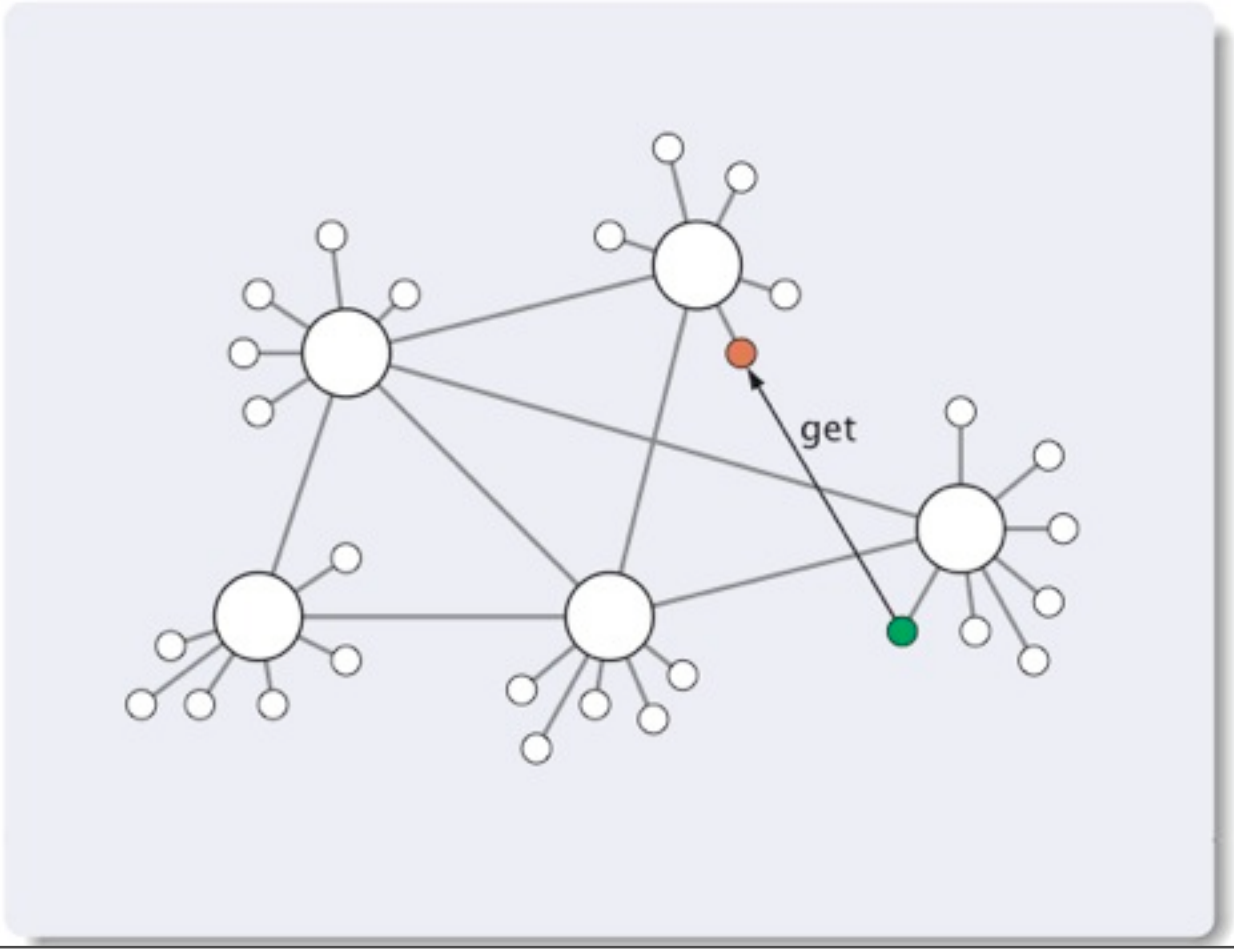
Figures courtesy of Aaron Harwood NICTA

Gnutella: Super-peers



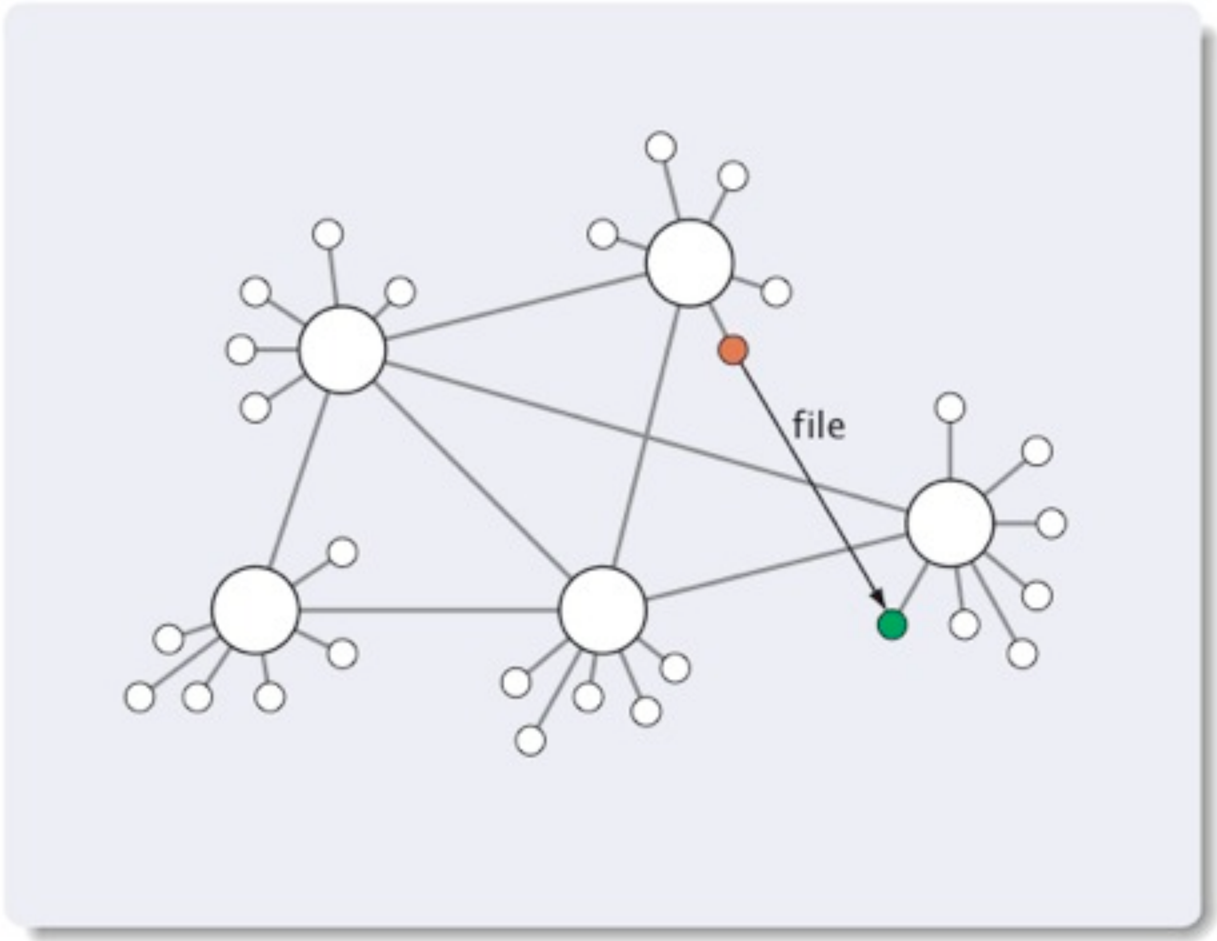
Figures courtesy of Aaron Harwood NICTA

Gnutella: Super-peers



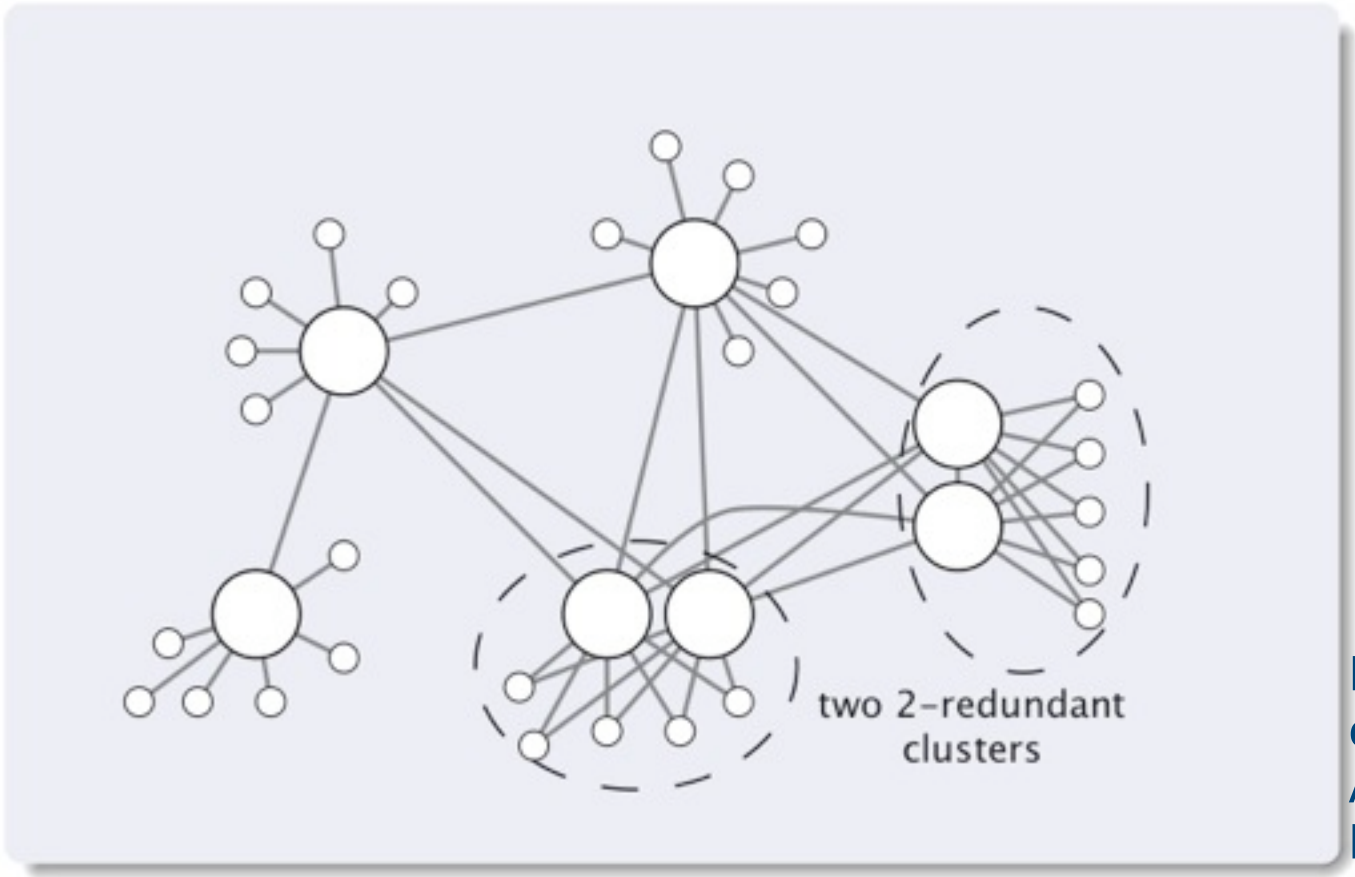
Figures courtesy of Aaron Harwood NICTA

Gnutella: Super-peers



Figures courtesy of Aaron Harwood NICTA

Gnutella: Redundancy



Figures courtesy of Aaron Harwood NICTA

Gnutella (*newtella*)

- Other issues
  - to avoid too much flooding, a TTL can be used but of course there is no assurance that a file will be found
  - if too many peers (super-peers) fail then partitions may occur. But the power law assures that the system is fairly robust to random fails
  - IP addresses, associations, full path names are all visible. Induced denial of service via fake push and fake pong messages is possible. Fake content.

ed2k

- eDonkey 2000 network is a P2P network with super-peers (called servers) like gnutella but there are no connections between super-peers
- Servers index files of their connected peers and allow peer to search for files
- All download is done from peer to peer
- Peers are connected to several servers and learn more servers through server lists exchanges
- The most popular client of the ed2k network is eMule

ed2k

- the ed2k architecture manages
  - searching files through keywords on the servers
  - concurrent download of files from several peers
  - partial sharing of files during download
  - verification of downloaded data through hashes

Bittorrent

- Bittorrent is a P2P network with lots of servers (in the sense of ed2k) managing a set of peers interested in a file. Those servers are called trackers.
- Trackers are found through the web or through the Kad network
- The protocols favor peers that upload against peers that only download (leechers) through choking
  - Choking is a temporary refusal to send
  - The peer might still be able to download later on

Bittorrent

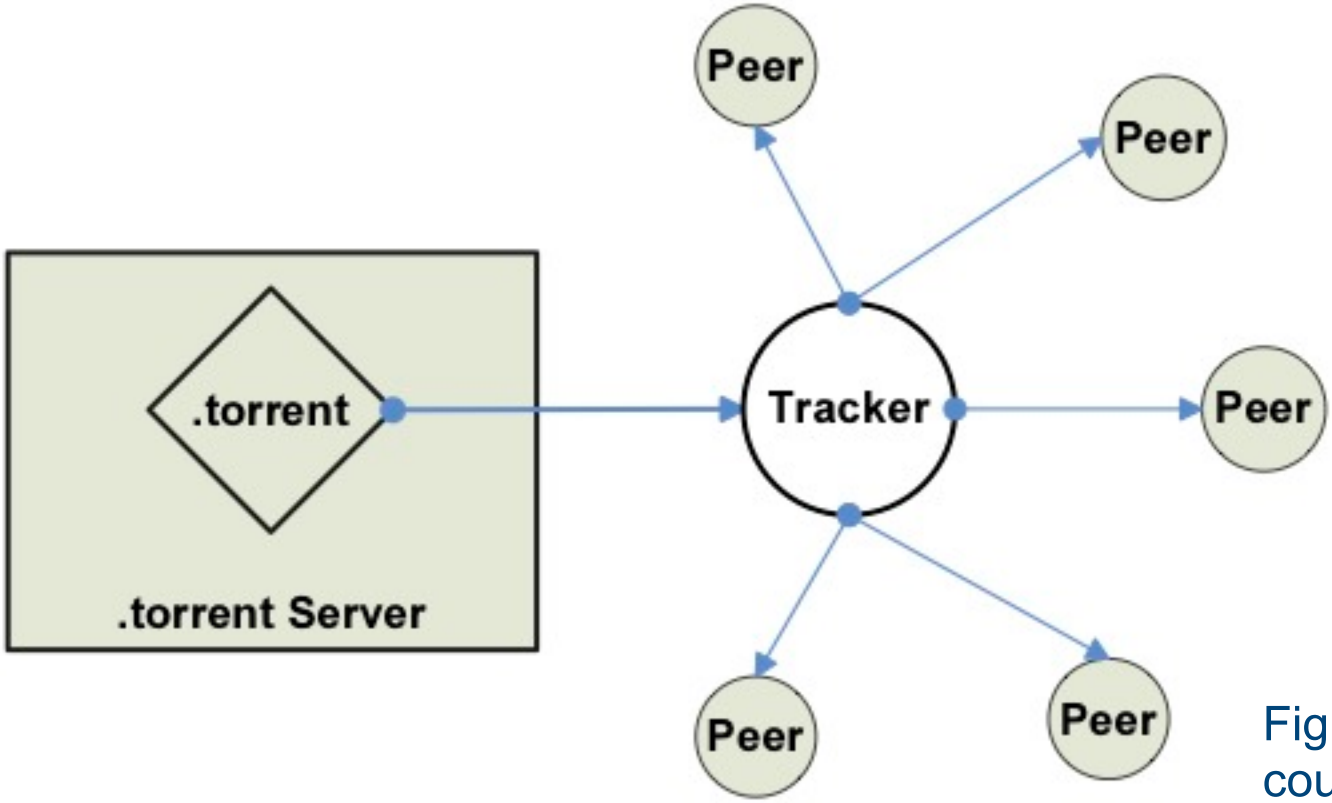


Figure courtesy of E. K. Lua et al.

Bittorrent

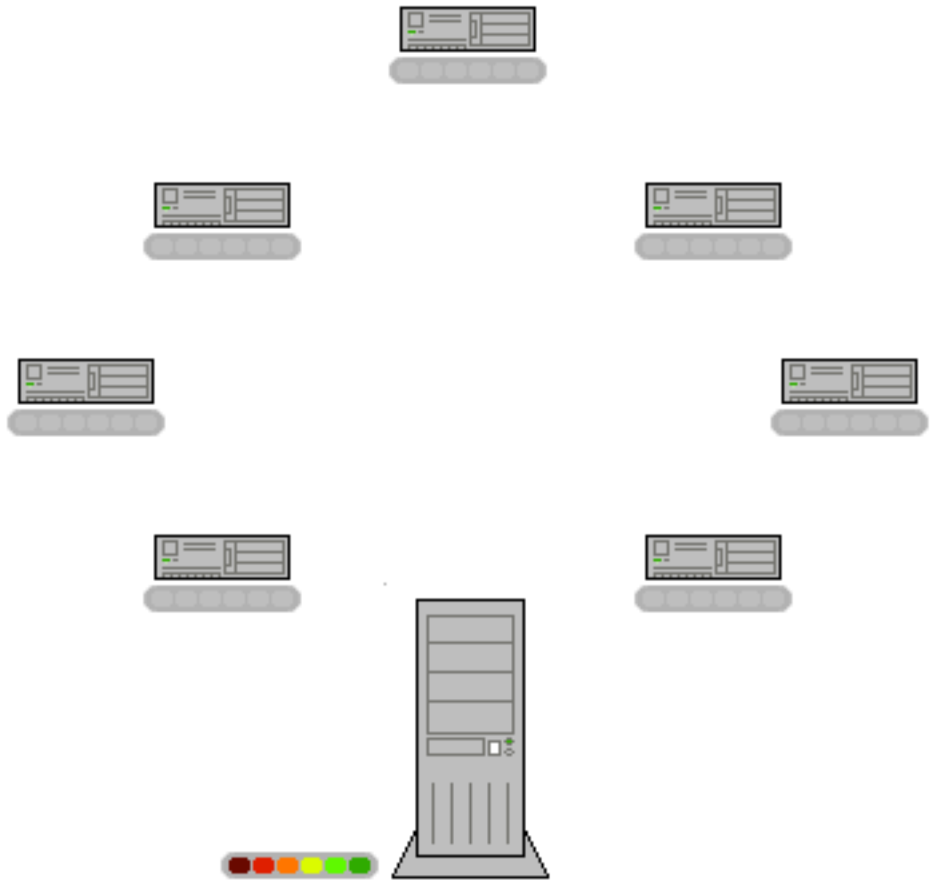


Figure courtesy of Wikipedia

Freenet

- Freenet is an anonymous network
  - it provides a virtual distributed FS storing files anonymously (and also freesites and chats)
  - each peer provides a part of the storage space
  - replication is used to make sure that data is not lost when peers fail
  - ciphering is used to make sure that data cannot be modified by anyone but its author and to make sure data is exchanged anonymously
  - the architecture is fully distributed (like gnutella) but Freenet choses where the data is stored

Freenet

- GUIDs
  - each file is associated a global unique id which is used as a key to find the data
  - ids are calculated using SHA-1 secure hash
  - there are mainly two kinds of keys
    - CHK (content hash keys) are hashed on the content of an encrypted file and are used by freenet to locate a file
    - SSK (signed subspace keys) are used by users to manage a set of files. Only the owner of the SSK can write to the subspace while it can be read by anyone. In fact a SSK points to a text file that lists CHKs of the files in the subspace. They can be used to form a hierarchy or to split big files.

Freenet

- Freenet basically support two operations:
  - PUT that sends the data to one peer that will store it. It will then move or be replicated as required. The data will be tied to a GUID (CHK or SSK).
  - GET that, given a GUID, is forwarded through the network to one peer that holds the data tied to it. The data is sent back using back-propagation.
- Anonymity
  - Messages sent on the network are encrypted for each pair of peers that move them.
  - Routing is only local and based on GUID, thus no peer can tell if the next one (resp. previous one) is the final receiver (resp. original sender) of the message

Freenet: routing

- Routing tables store only GUIDs and the connected peer that is supposed to be nearer to the data
  - They are only local
  - They are constructed dynamically
  - At the beginning a peer is only connected to another peer and it asks it everything
  - When data is found all nodes that move it back to the requester update their tables
  - Freenet also uses path folding (i.e. two non connected nodes that exchange data may connect to each other)
  - Thus the network auto-organizes

Freenet: routing

- Routing works as follows
  - Each node is associated a location (L $\in$ {0 and 1})
  - The GUID of the request is translated into a number between 0 and 1
  - The request is first sent to the connected node which location is nearest to the GUID number.
  - Then Freenet uses greedy routing as follows

Freenet: routing

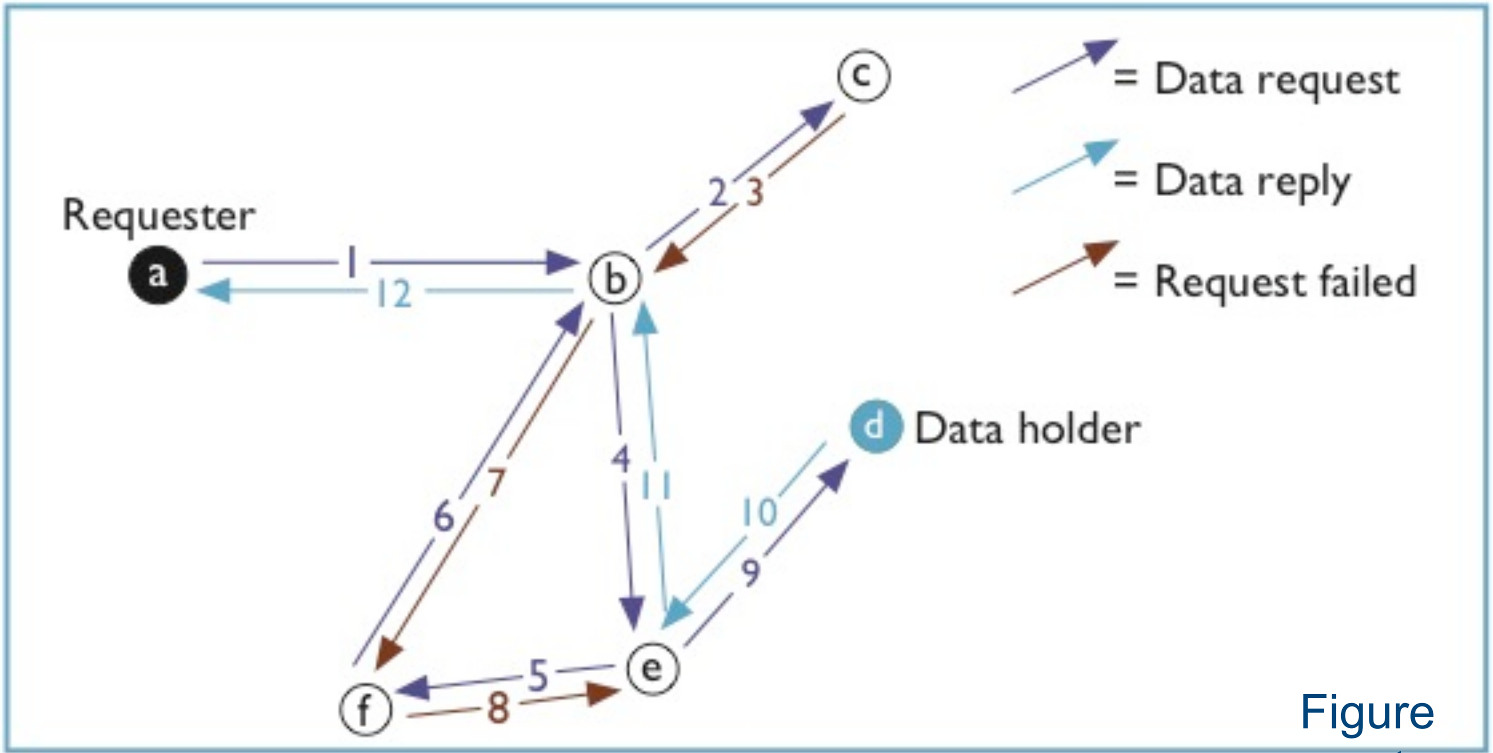


Figure courtesy of Ian Clarke

Freenet: routing

- Query requests are associated with HTL (hops to live) values in order to avoid waiting for a long time
- When the data comes back along the network it may be replicated
- Moreover the nodes locations will change due to path folding (two nodes connected will have similar location values)

Freenet: routing

- When storing data:
  - The request is sent along the network
  - Either a node has already the data and the new data is discarded. The original data comes back to the requester and is thus cached on the path.
  - Or the HTL will go down to 0. Then the data is sent to the node which decremented HTL to 0. The data will also be cached along the path.

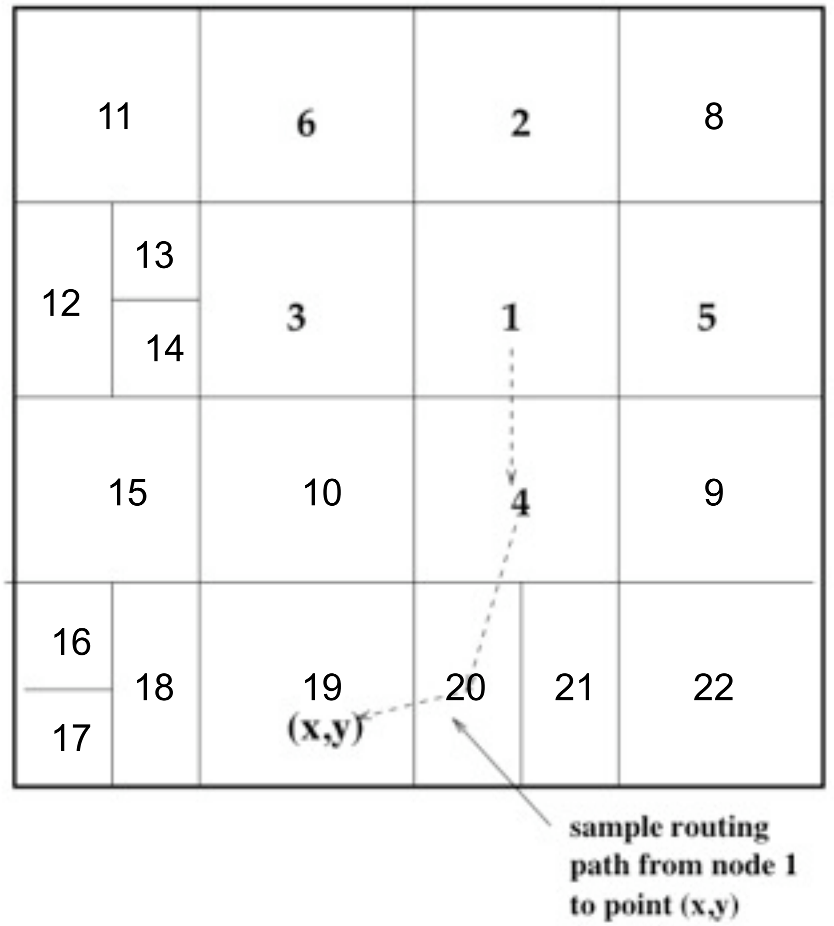
Structured P2P

- 4 Research papers appeared roughly at the same time : CAN, Chord, Pastry and Tapestry
- Those system exhibit almost the same features
  - Data location is controlled (as in Freenet)
  - But the P2P overlay structure is also controlled
- They support two main applicative methods which are similar to Hashtables methods (hence the name DHT distributed hash tables)
  - PUT: puts some data in the system tied to a key
  - GET: retrieve some data from the system given a key

CAN

- The CAN (Content Addressable Network) architecture is based on a mapping between the keys and a d-dimensional Cartesian space managed as a multi-torus ($d>1$)
- Each peer is associated with a rectangular region of the d-dimensional space and manages all data which keys map to this region
- Each peer maintain connections with the peers which manage neighboring regions
- Routing is simply done by choosing the neighbor that is nearer to the destination key

CAN



1’s neighbor set = {2,3,4,5}

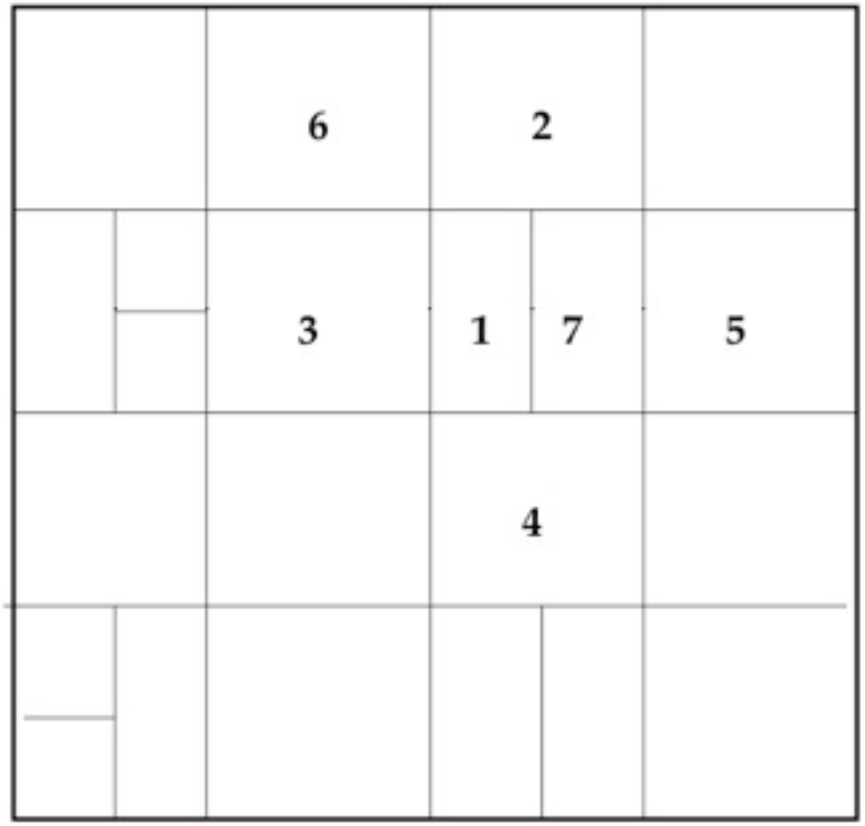
8’s neighbor set = {22,2,5,11}

Figure courtesy of S. Ratnasamy et al.

CAN

- When a new peer joins the overlay
  - it chooses a random point in space
  - it connects to any peer in the CAN
  - starting with this peer the CAN will route the join request to the peer that manages the region which contains this random point
  - this last peer region will be cut in half and one subregion will be given to the new peer. All data in this subregion will be transmitted to the new peer
  - the neighborhood will be updated

CAN



After node 7 joins

1’s neighbor set = {2,3,4,7}

7’s neighbor set = {1,2,4,5}

Figure courtesy of S. Ratnasamy et al.

CAN

- When a peer leaves the overlay
  - if the region of one neighbor can be merged with the departing peer zone to produce a valid single zone, this neighbor gets the new merged zone and all the data is transfered
  - if this isn't possible then the neighbor whose zone is smallest will get the zone and the data. Then the overlay will be recomputed in background so that each peer as only one valid region
  - In both cases, neighbors will communicate to rebuild the neighbor sets and connections

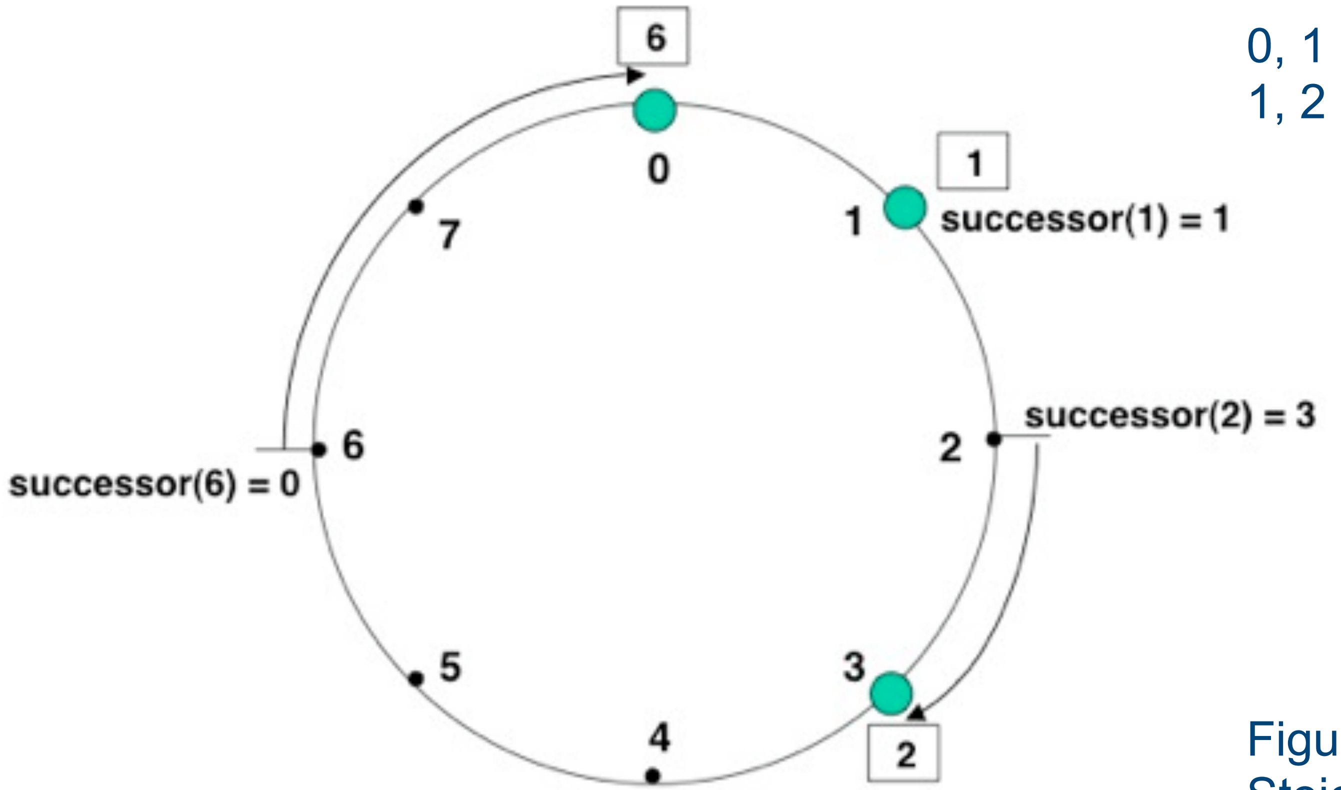
CAN

- In order to manage crashes and data loss several realities (each having its d-dimensional Cartesian space) can be setup so that each data is present in the different realities on different peers
- Thus if we have k realities, there will be k copies of all data in the CAN
- The realities can be used to optimize routing (using the cartesian distance to the key point) and/or to parallelize requests

Chord

- Chord uses consistent hashing to assigns keys to its peers
  - each peers receives roughly the same number of keys
  - when a peer leaves or join there is a minimal number of transfers
- The consistent hash function assigns each key a m-bit value using SHA-1
- Each peer is also associated a m-bit value called its identifier using SHA-1 hashing of its IP address
- Each key is associated to the first peer whose id is equal to or follows (the identifier of) k. This peer is called the successor of k. If ids are drawn on a circle from 0 to 2^m-1, successor(k) is the first node clockwise from k.

Chord



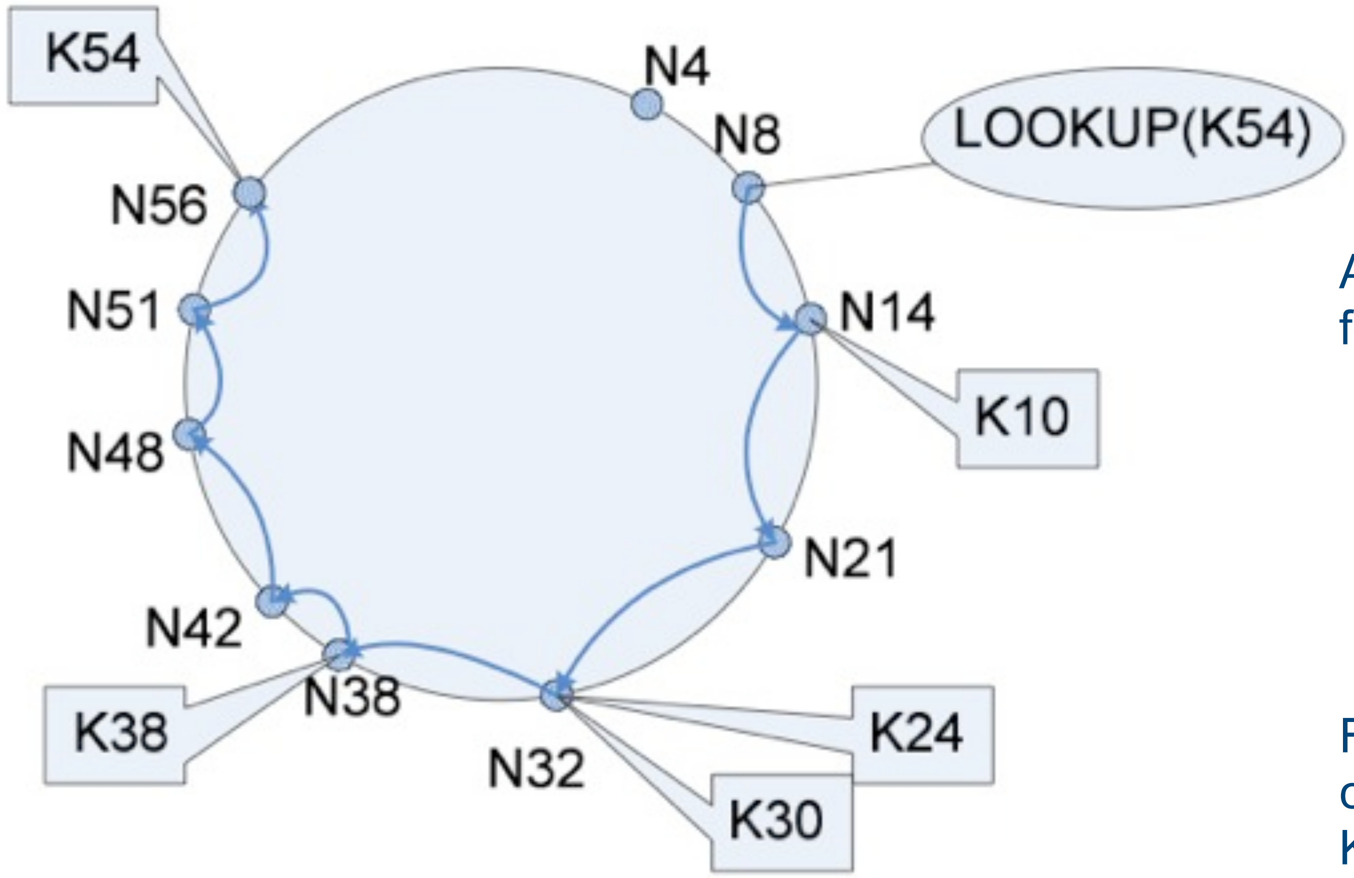
0, 1 and 3 are nodes
1, 2 and 6 are keys

Figure courtesy of I. Stoica et al.

Chord

- When a node n joins the network some keys owned by n's successor will now be managed by n
- When a node n leaves the network all its keys will be moved to n's successor
- Each peer in the network needs to know how to contact its successor
- Lookup queries are passed along the Chord-ring from successors to successors up to the time where successor(n) > k then k is stored on successor(n)
- The data is then moved back to the requester using back propagation

Chord



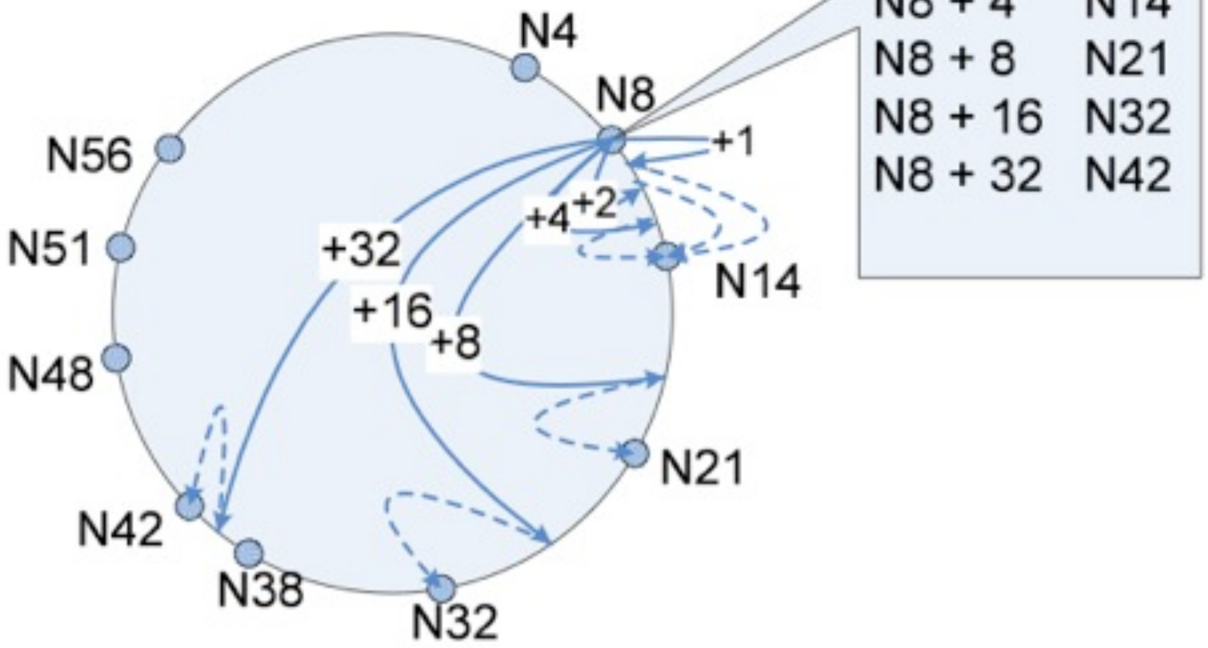
A Chord-ring for m = 6

Figure courtesy of E. K. Lua et al.

Chord

- To optimize routing, each node has a routing table (called a finger table) containing at most m entries
  - entry i stores successor($n+2^{i-1}$) where n is the node's id
  - its stores both the node id and its IP address and port number
- The finger tables need to be updated each time a node joins or leaves
  - This is managed by a background process
- When a peer fails it is possible that a node no longer knows its successor
  - To avoid this, each node stores the lists of its r successors

Chord



Finger Table	
N8 + 1	N14
N8 + 2	N14
N8 + 4	N14
N8 + 8	N21
N8 + 16	N32
N8 + 32	N42

A Chord-ring for m = 6

Figure courtesy of E. K. Lua et al.

Pastry

- Pastry and Tapestry are similar
- They are both based on Plaxton-like prefix routing (a Plaxton mesh is a distributed data structure optimized to support a network overlay for locating named data objects which are connected to one root peer - both Pastry and Tapestry actually uses several roots to be more fault tolerant and scalable)

Pastry

- Pastry uses an external routing metric to optimize routing (this can be latency - using ping - number of hops - using traceroute - bandwidth...)
- It is a DHT where keys and node ids are 128 bit unsigned integers organized on a ring like Chord
- Node ids are chosen randomly and uniformly so that peers which are adjacent in node ids are geographically diverse
- The overlay is based on peers maintaining 3 data structures:
  - Leaf nodes set
  - Neighborhood set
  - Routing table

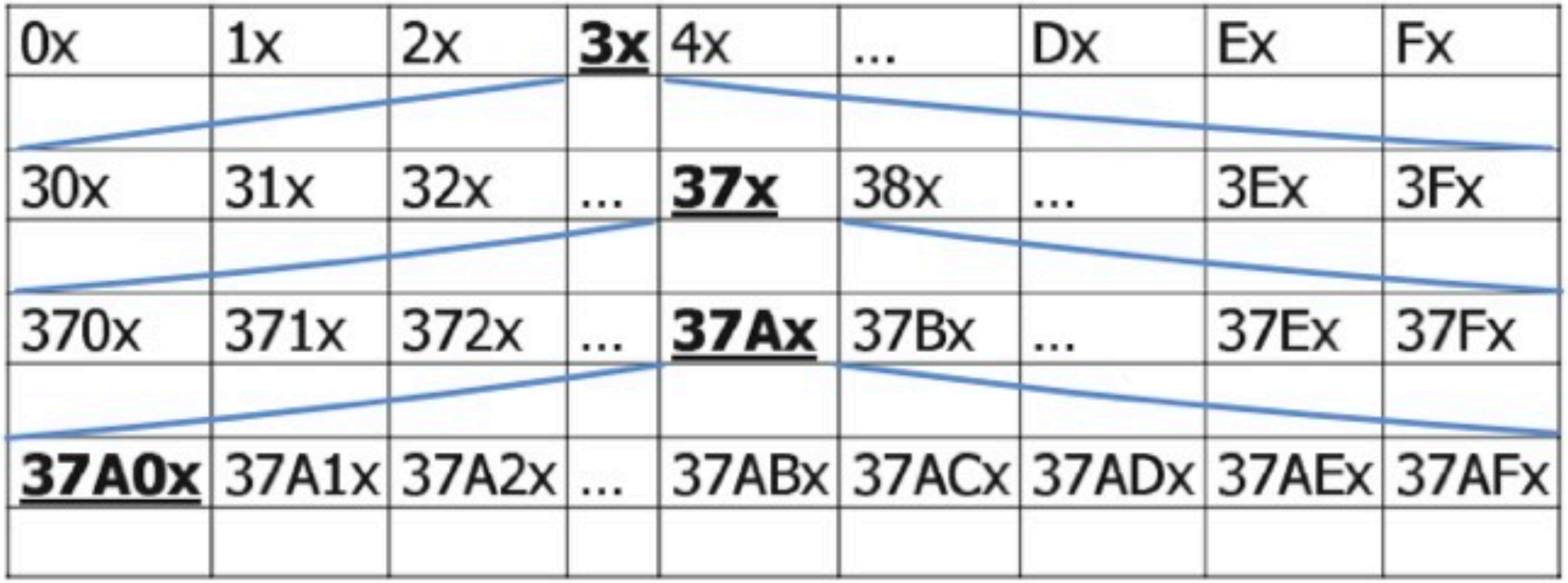
Pastry

- Leaf set: L/2 nodes before and after the node N on the ring
- Neighborhood set: M closest peers based on the metric. It is not used directly but allows to maintain information in the routing table
- Routing table contains one entry for each address block assigned to it
  - Address blocks are formed by splitting the 128 bit integers in blocks of b bits (usually 4 bits => 32 hexadecimal digits)
  - This partitions the addresses into levels. Level 0 contains 0 common digit with the node's address, Level 1 contains 1 common digit...
  - The table contains the address of the closest known peer for each digit at each address level (except for the digit of the node itself at any level). It typically contains 15 contacts per level.

Figure courtesy of E. K. Lua et al.

Pastry

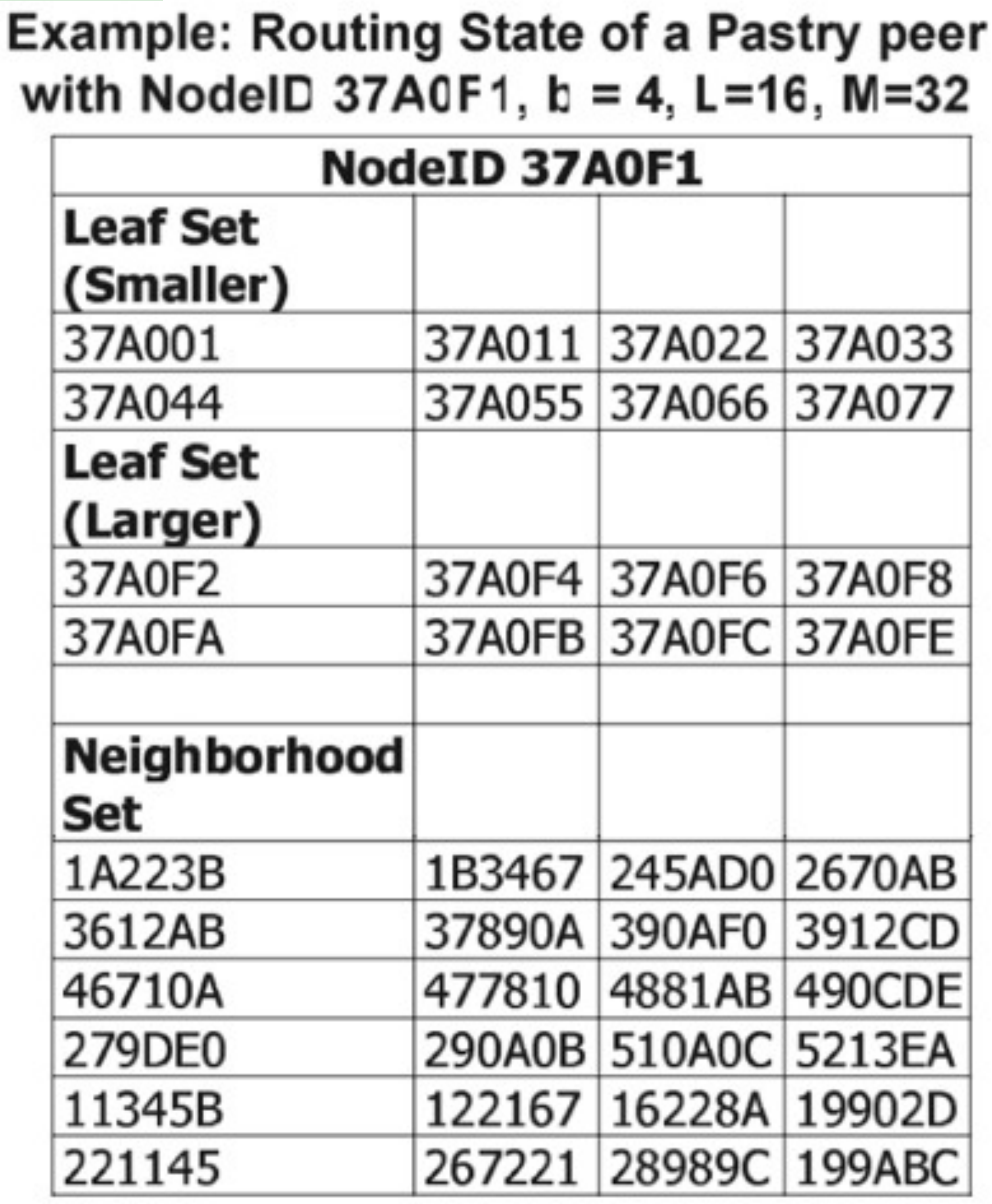
Routing Table of a Pastry peer with NodeID 37A0x, b = 4, digits are in hexadecimal, x is an arbitrary suffix



0x	1x	2x	**3x**	4x	...	Dx	Ex	Fx
30x	31x	32x	...	**37x**	38x	...	3Ex	3Fx
370x	371x	372x	...	**37Ax**	37Bx	...	37Ex	37Fx
37A0x	37A1x	37A2x	...	37ABx	37ACx	37ADx	37AEx	37AFx

Pastry

Figure courtesy of E. K. Lua et al.



Example: Routing State of a Pastry peer with NodeID 37A0F1, b = 4, L=16, M=32

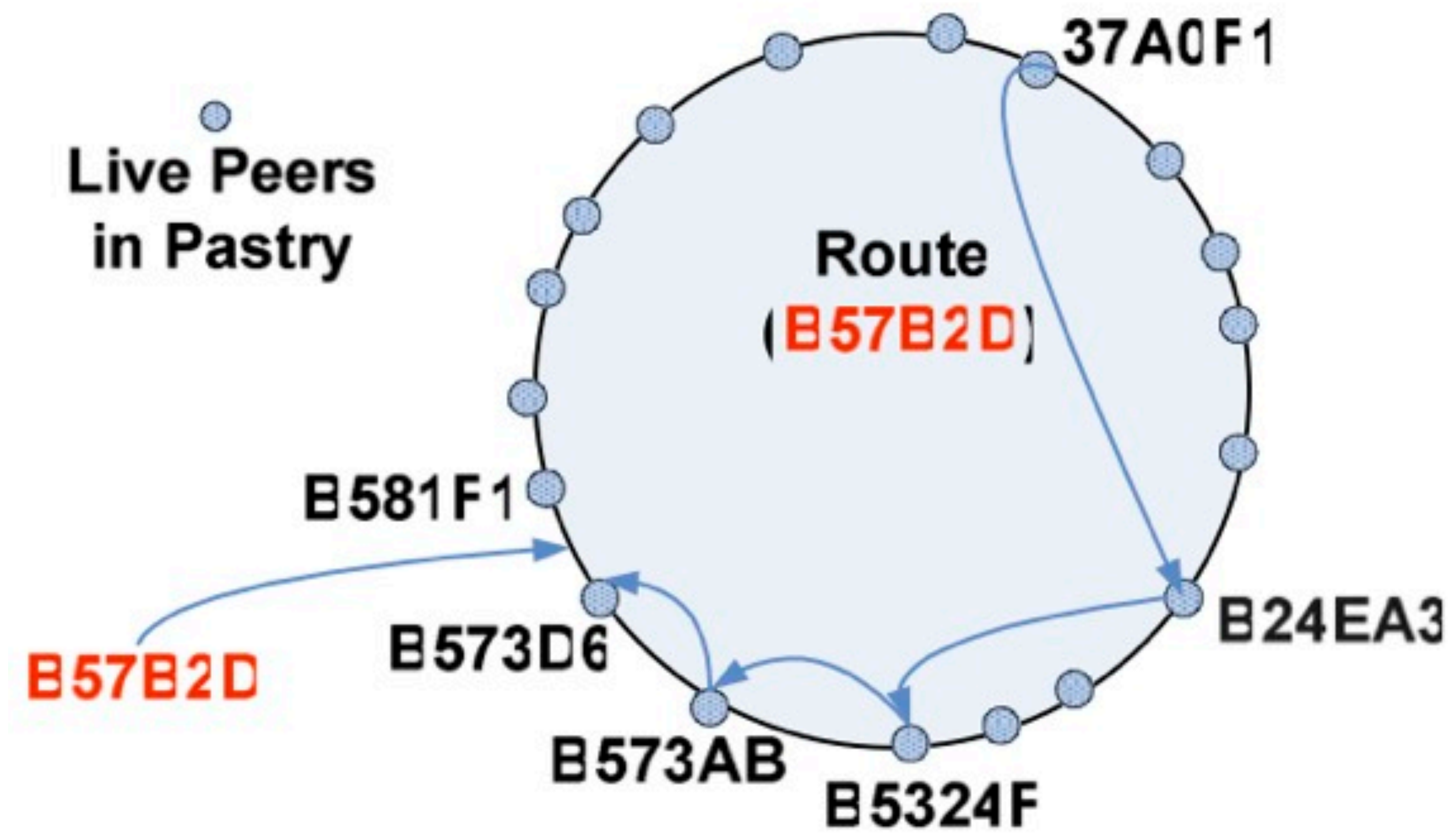
NodeID 37A0F1			
Leaf Set (Smaller)			
37A001	37A011	37A022	37A033
37A044	37A055	37A066	37A077
Leaf Set (Larger)			
37A0F2	37A0F4	37A0F6	37A0F8
37A0FA	37A0FB	37A0FC	37A0FE
Neighborhood Set			
1A223B	1B3467	245AD0	2670AB
3612AB	37890A	390AF0	3912CD
46710A	477810	4881AB	490CDE
279DE0	290A0B	510A0C	5213EA
11345B	122167	16228A	19902D
221145	267221	28989C	199ABC

Pastry

- A packet can be routed on the ring towards the peer that is nearest to the destination address (there can be no peers at the exact address)
- Routing works as follows:
  - First search in the leaf set to find the dest address
  - If this fails, use the routing table to find a node that matches better the dest address (ie it has at least x digits in common with the dest address with x > y: the number of digits that the current node has in common to the dest address)
  - If this fails (either there is no one that matches better or the peer that matches better is dead), send the packet to someone in the leaf set which is nearer to the address

Figure courtesy of E. K. Lua et al.

Pastry



Pastry

- When peer fails, the peers that use them for routing or for maintaining the contact lists use peers in their neighborhood and leaf sets to find a replacement peer

Kademlia

- Kademlia is a DHT using 160-bits keys and NodeIDs (both are taken in the same space as Chord and Pastry)
- However it uses XOR has a distance metric for routing. XOR has the following geometric (non euclidean) distance features:
  - XOR(a,a) = 0 and XOR(a,b)>0 if a≠b
  - XOR(a,b) = XOR(b,a)
  - XOR(a,b)+XOR(b,c) ≥ XOR(a,c)
- Most interestingly as XOR(a,b) = XOR(b,a) Kademlia's routing is symetric and therefore nodes learn information from the nodes that sends them messages which is not the case in Chord

Kademlia

- Routing in Kademlia uses the concept of k-buckets
- K-buckets are lists of at most k nodes (k can be 20)
- Each node has 160 such lists (one for each bit in the address space)
- The n-th k-bucket contains nodes that have bits 0 to n-1 equal to the node id an the n-th bit different
- A lot of the k-buckets are empty when n increases towards 160 as the key space is smaller and smaller. For n=0 the key space is the half of the whole k-space...
- The following tree shows 4 k-buckets for a node starting with address 0011

Kademlia

Figure courtesy of P. Maymounkov et al.

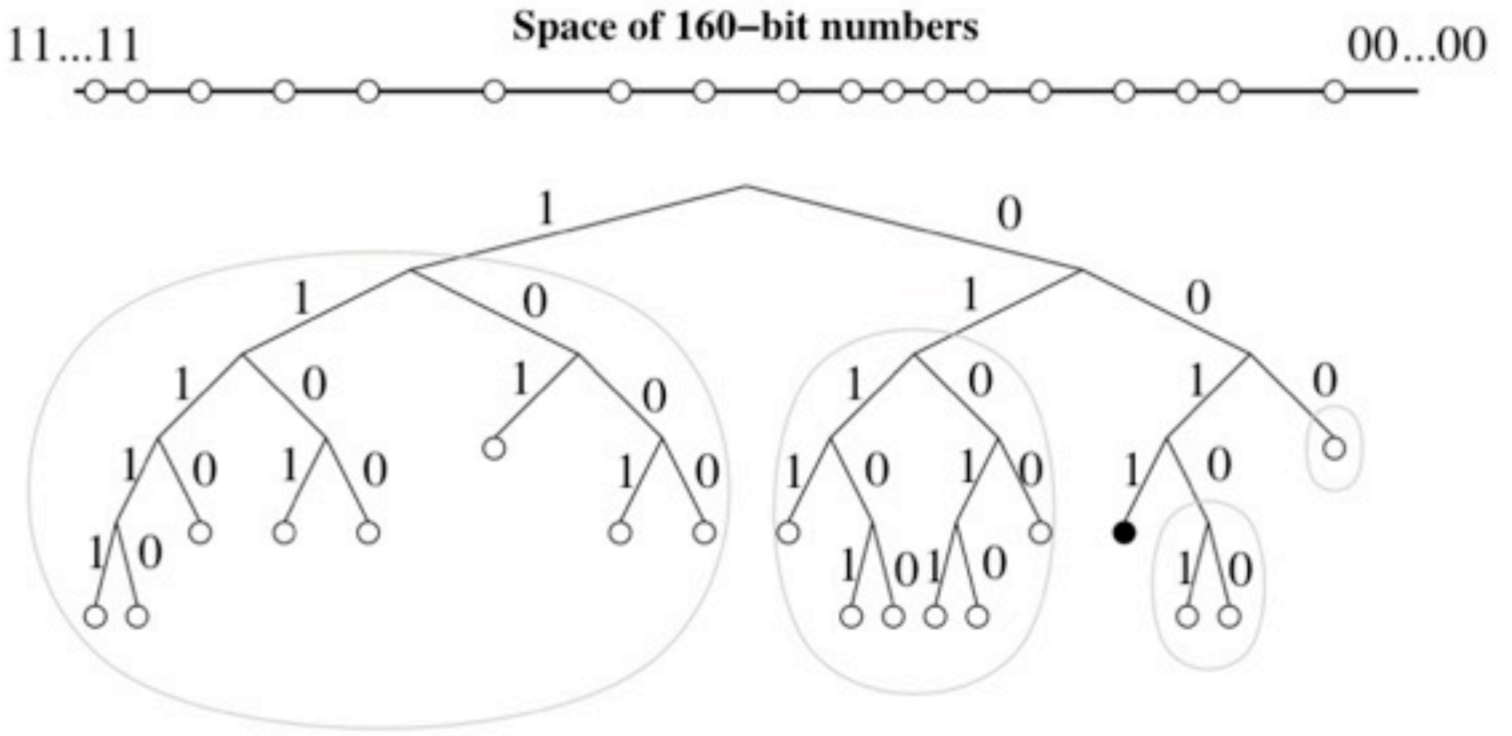
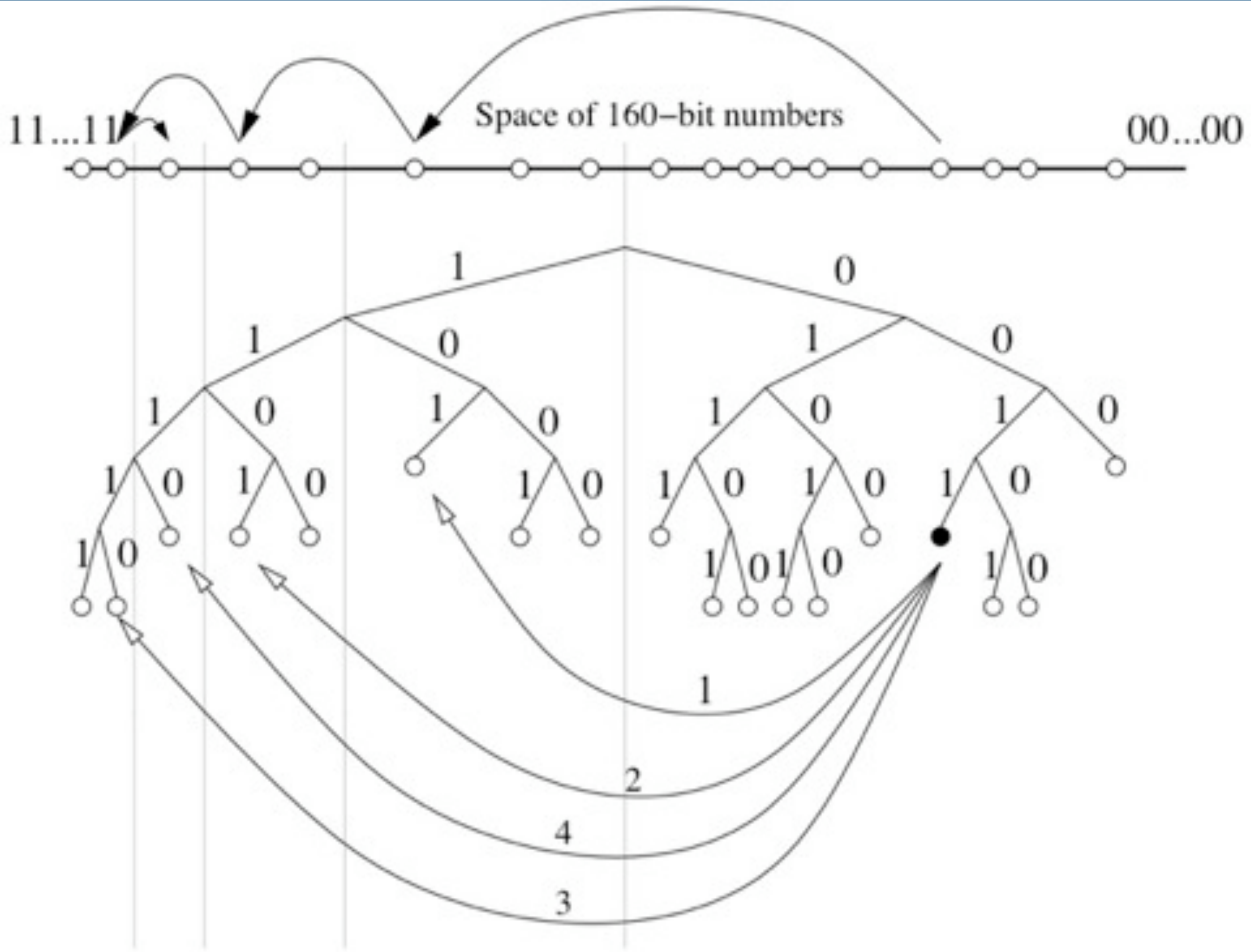


Fig. 1: Kademlia binary tree. The black dot shows the location of node 0011 · · · in the tree. Gray ovals show subtrees in which node 0011 · · · must have a contact.

Kademlia

- When routing to a key, a node sends messages to α (typically 3) nodes that are nearer to the destination key. α allows parallelizing the lookup so that it doesn't stop due to failed nodes
- The nodes are selected from the non empty k-bucket that is nearest to the key
- If there are less than α nodes in the k-bucket then the system takes nodes from a k-bucket that is farther
- The contacted nodes will reply with lists of the k nearest nodes to the key that they know
- The original node will use α of those k nodes for the next request until it founds the k nearest nodes to the key which store it

Kademlia



Kademlia

- Each time a node receives a message it updates its k-buckets as follows:
  - If the k-bucket that should contain a new found node has less than k nodes it is added
  - If the k-bucket is full then the oldest node in the k-bucket (they are sorted by last date of contact) is pinged. If it answers then the node will go in a sub-list that is used only when nodes in the k-bucket fail.

Kademlia

- When a new node joins the network it chooses its node id at random. It then must know one node in the network (given by the user). This node will go in a k-bucket (all others being empty)
- The node will then search for its own ID in order to add itself to its k nearest neighbors (while filling its own k-buckets)
- Then using random key searches it will continue to fill its k-buckets starting with the biggest ones

Kademlia

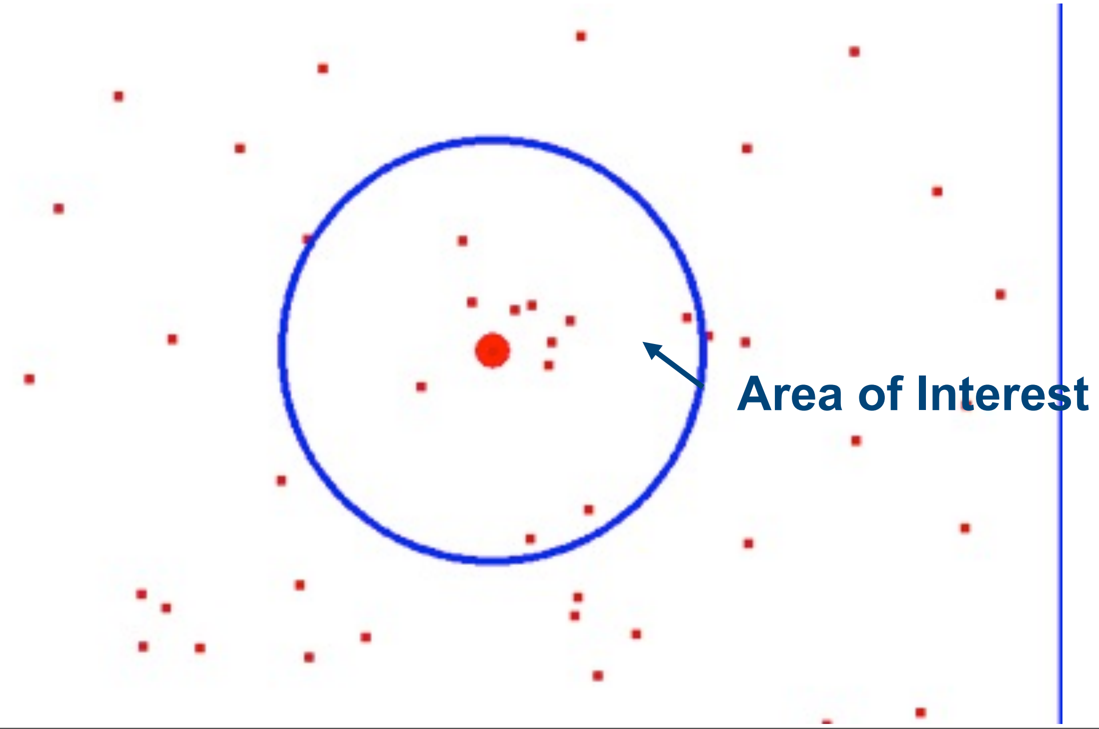
- Kademlia is the most used DHT protocol for file sharing
- It is used by eMule and Bittorrent clients
- Peer data are typically stored using a 160-bit hash of the file content
- As each of the k nearest nodes can store different peer data. One can find up to k peers to start downloading from.
- Keyword searches are made by storing a file link (containing at least each name and content hash) with keys consisting of 160-bit hashes of any of its title words

Schedule part 2 MMVEs

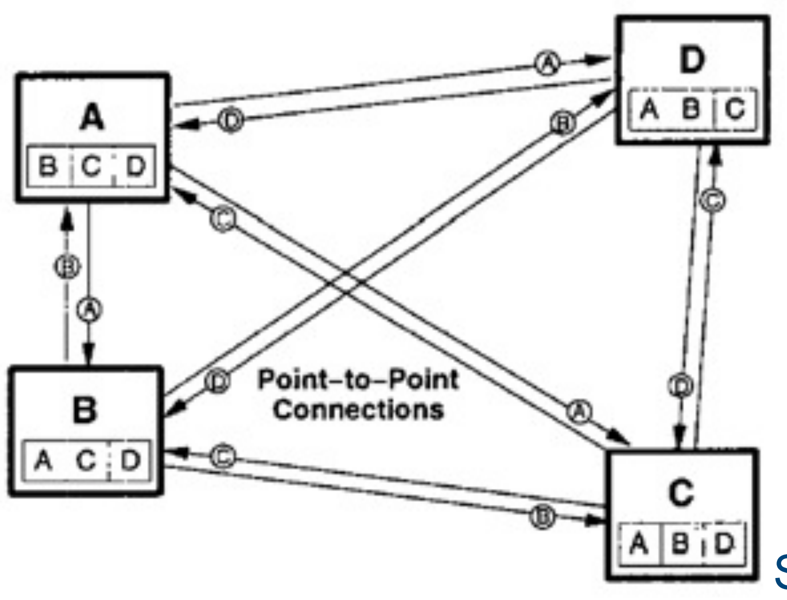
- The Scalability Problem
- Hybrid C/S and P2P approaches
- Zone-based coordinators
- Enhanced Point 2 Point
- Neighbor-list exchange
- Mutual Notification
- Mutual Notification with Overlay Multicast

The Scalability Problem

- Many (> millions) avatars scattered in the virtual world
- Message exchange with those within Area of Interest (AOI)
- How does each node receive the relevant messages?



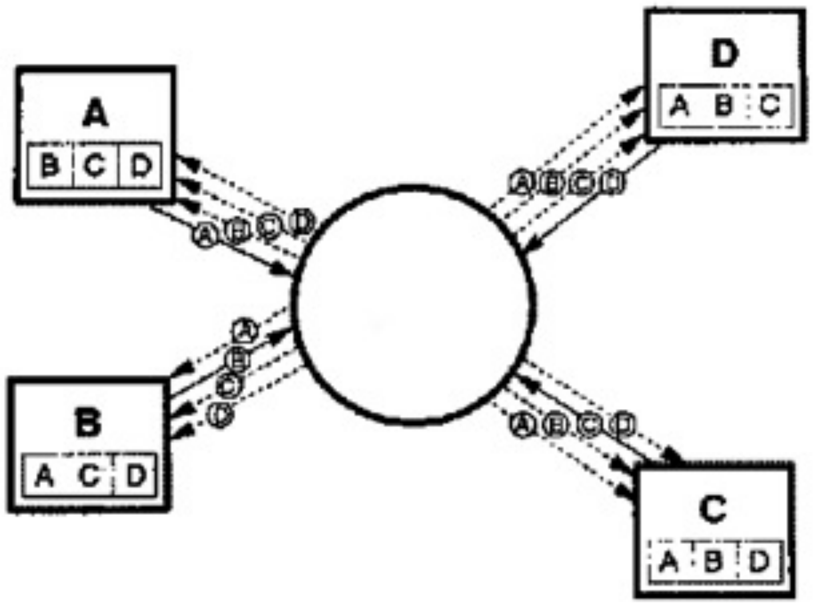
A simple solution (point-to-point)



Source: [Funkhouser95]

N * (N-1) connections ≈ O(N^2) → Not scalable!

A better solution (client-server)

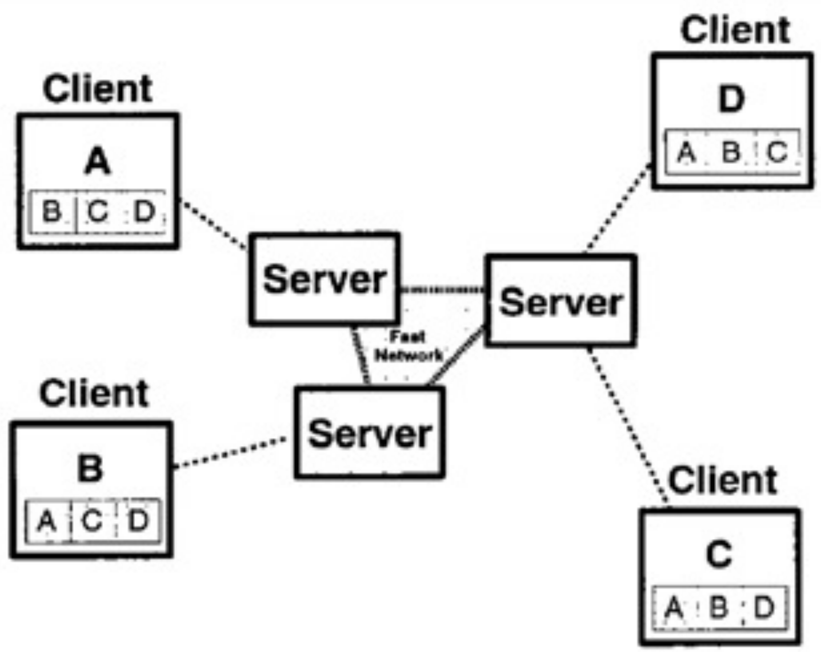


Source: [Funkhouser95]

Message filtering at server to reduce traffic

N connections = O(N) → server is bottleneck

Current solution (server-cluster)



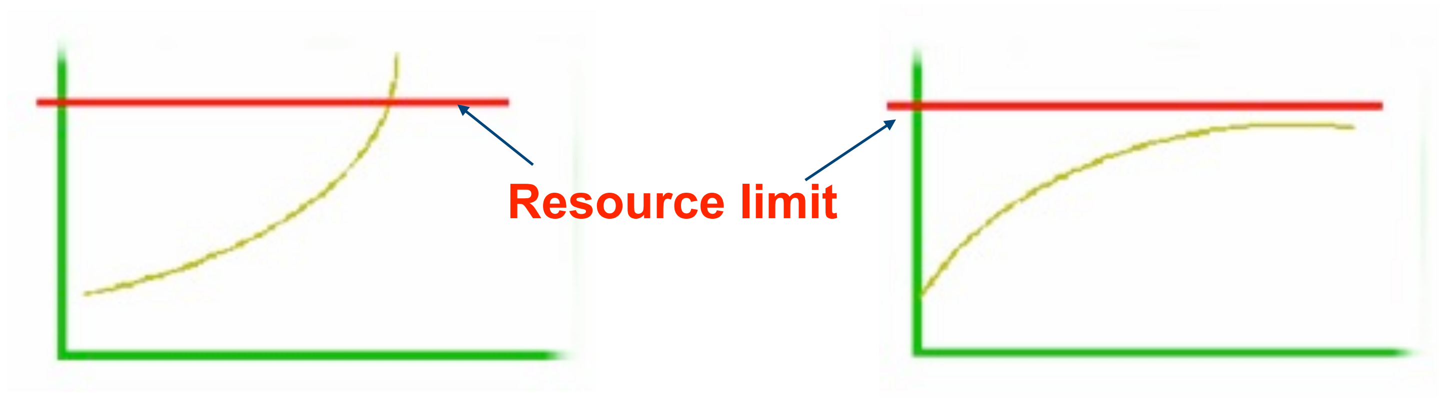
Source: [Funkhouser95]

Still limited by servers. Expansive to deploy & maintain.

Scalability Analysis

- Scalability constrains
  - Computing resource (CPU)
  - Network resource (Bandwidth)

Non-scalable system vs. Scalable system



x: number of entities

y: resource consumption at the limiting system component

Solution ?

- Strategies
  - Increase resource => More servers
  - Decrease consumption => Message filtering
- Architectures

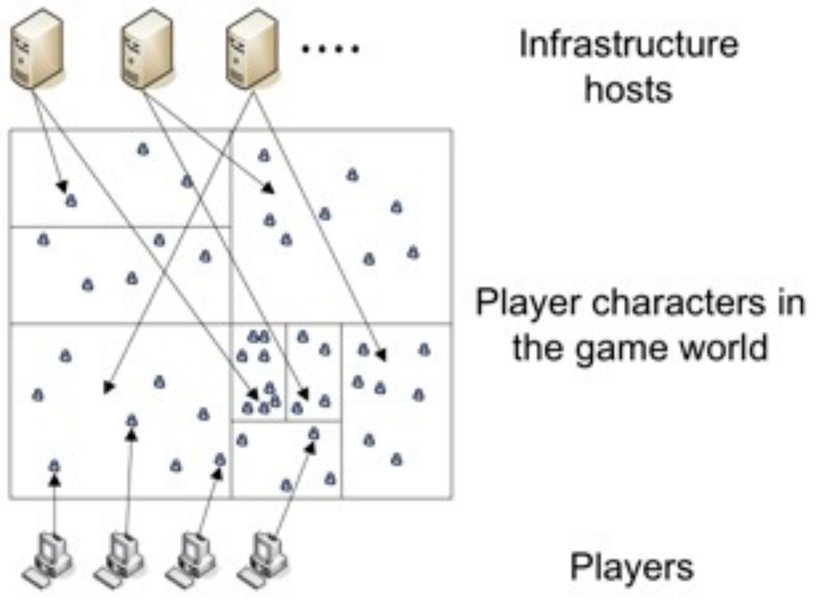
Architectures	Scale	
– Point-to-point	tens	10^1
– Client-server	hundreds	10^2
– Server-cluster	thousands	10^3
– Peer-to-Peer ?	millions	10^6
…		

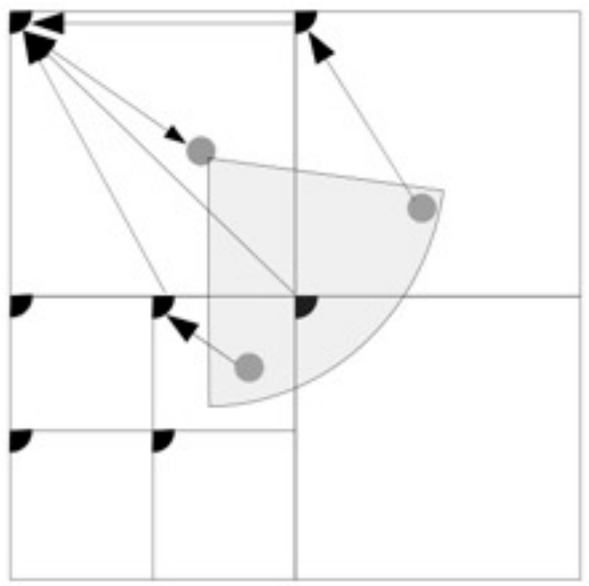
Hybrid C/S and P2P approaches

- Mix C/S approaches with P2P
  - P2P can be on the server-side
    - Servers communicate through a P2P overlay network
  - or on the client-side
    - Clients form a P2P overlay network
  - or both-side

Server-side P2P

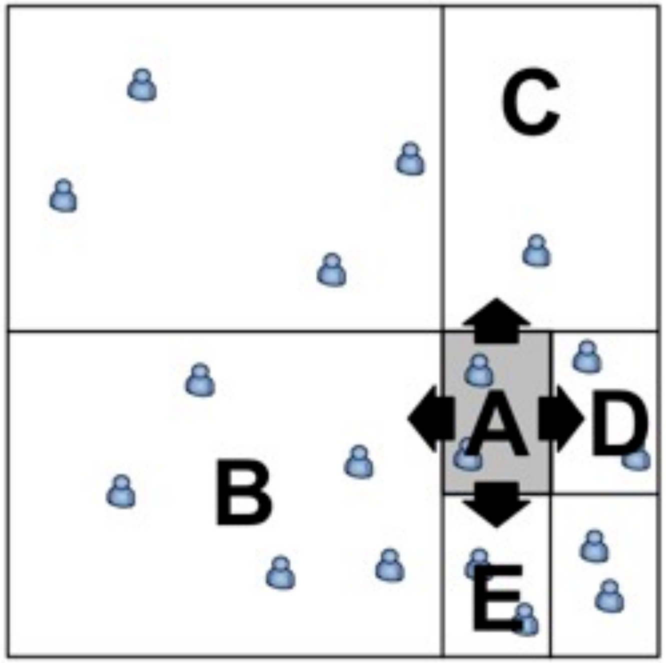
- Example: [Rieche et al. 2006] uses the CAN DHT
  - Can add servers dynamically when the server cluster is saturating





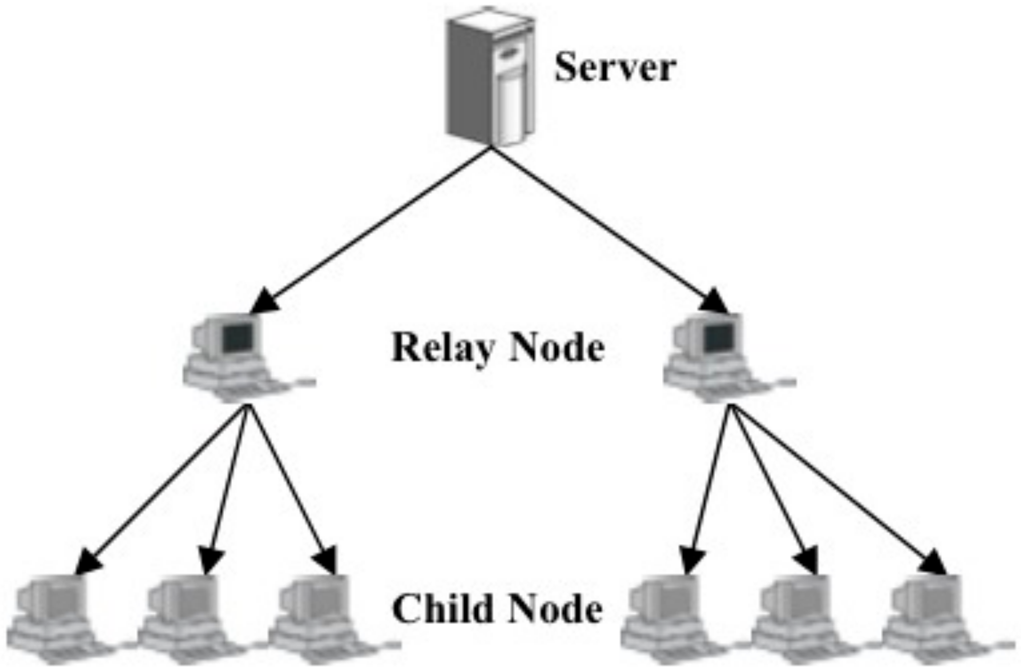
Server-side P2P

- Example: [Rieche et al. 2006]
  - Moreover if a server fails, replication on neighboring servers allows for graceful recovery



Client-side P2P

- Example: [Ito et al. 2006]
  - Some clients act as relay for other clients

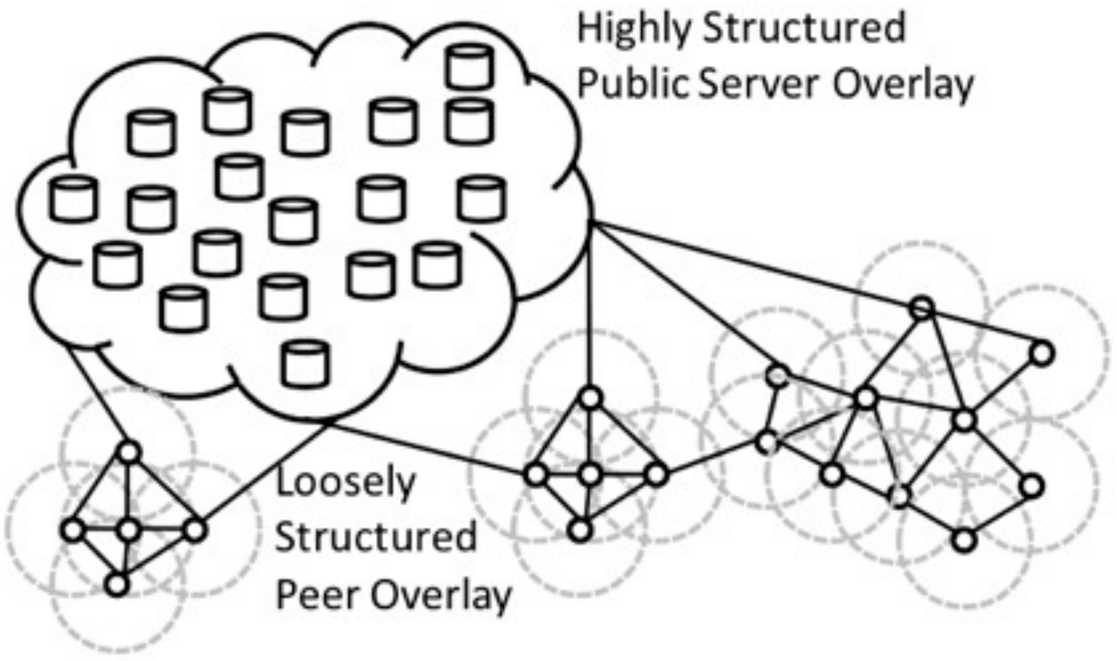


Client-side P2P

- Example: Badumna
  - Clients communicate mainly with each other using a P2P overlay (typically for object attribute updates and chat)
  - They use servers only when there is a need for arbitration, authentication or intrusion detection

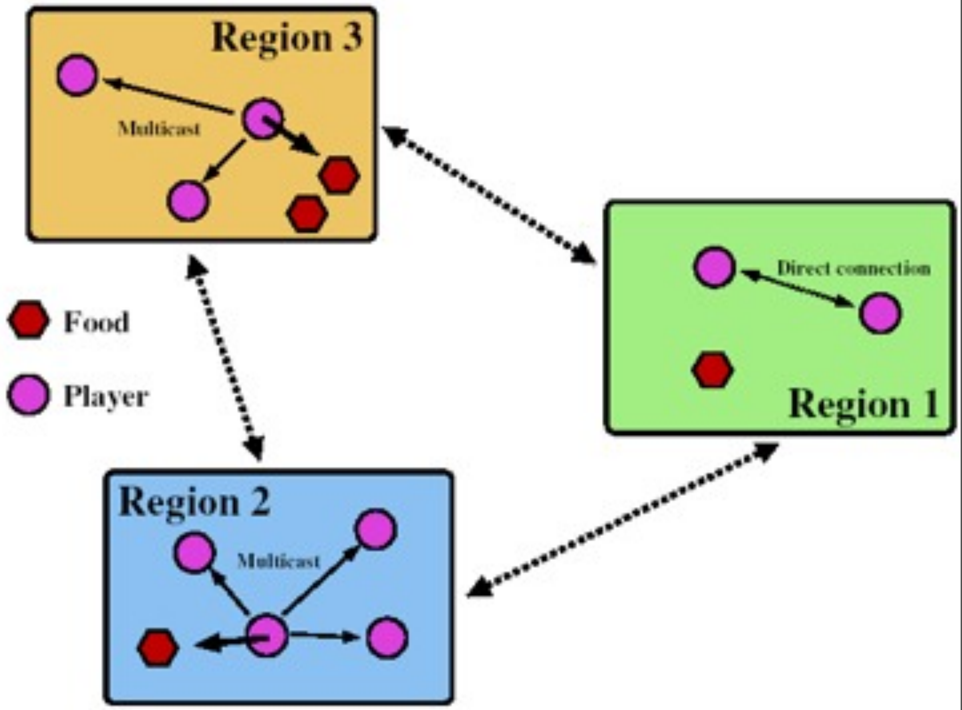
Both-side P2P

- Example: HyperVerse/GP3 [Esch et al. 2008]
  - Uses 2 overlay networks with different characteristics



Zone-based coordinators

- Fixed number of zones
- In each zone, one super-peer coordinates connectivity
- Example : SimMud [Knutsson et al. 2004]
    - coordinators manage multicast trees using Pastry and Scribe (multicast tree based on Pastry)

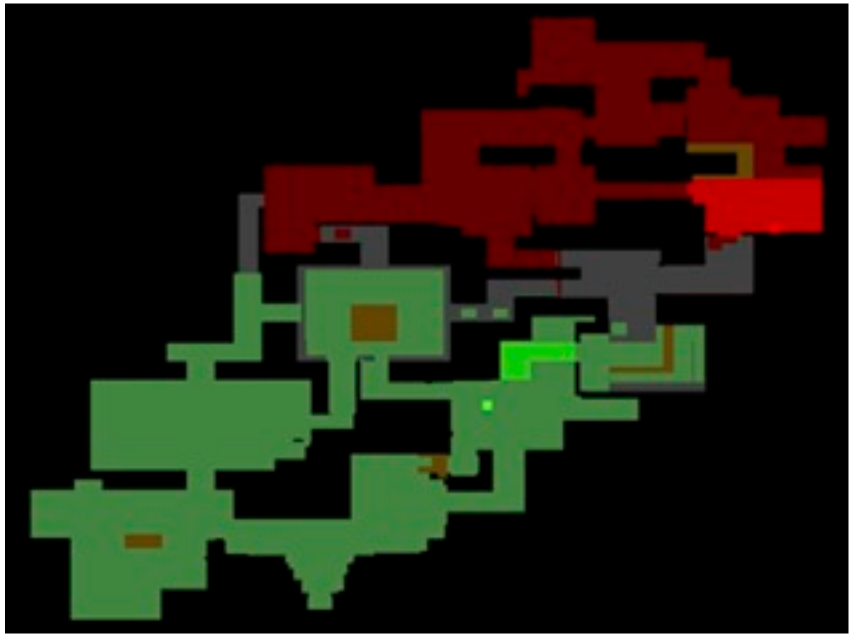


Enhanced Point 2 Point

- N*(N-1) Point 2 Point connections
- Plus filtering
- Example:
  Update Free Regions (UFR) [Makbily et al. 1999]
  - defines pairs of mutually invisible regions in the VE
  - avatars in different UFRs don't need to communicate
  - Problems :
    - all nodes need to communicate regularly to negotiate their UFRs
    - what happens when crowding occurs ?

Enhanced Point 2 Point

- Another example: Frontier Sets [Steed et al. 2004]
  - As long as the green avatar stays in the green part of the map and the red avatar stays in the red part, no mutual update is required

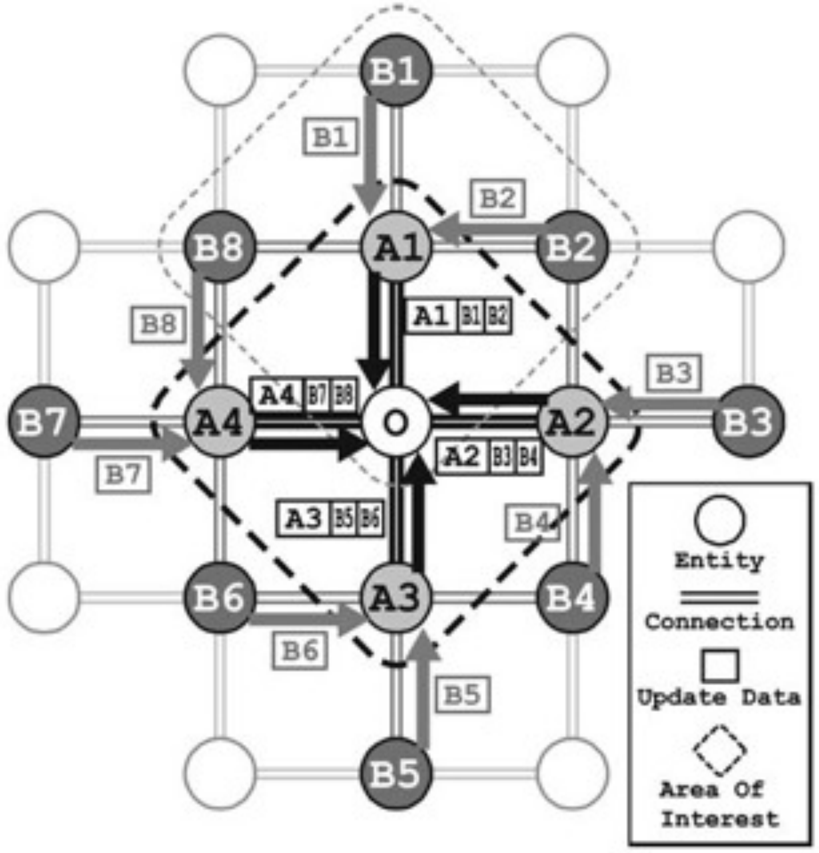


Neighbor-list exchange

- Each node connects with a fixed number of nearest neighbors
- Nodes constantly exchange neighbor-lists so that they can discover new neighbors
- To avoid partitions there is a minimum number of connections with some neighbors even if they are very far

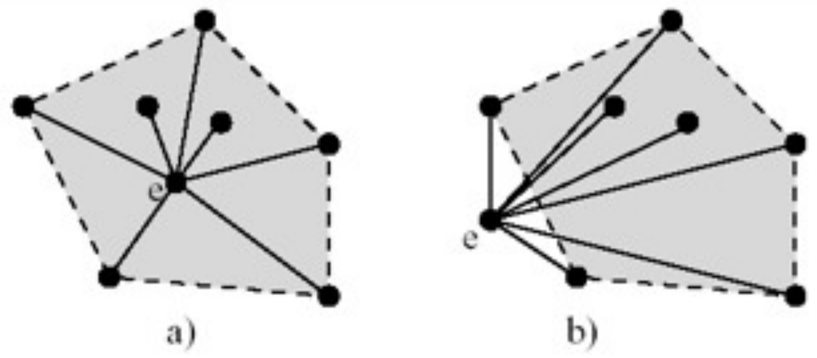
Neighbor-list exchange

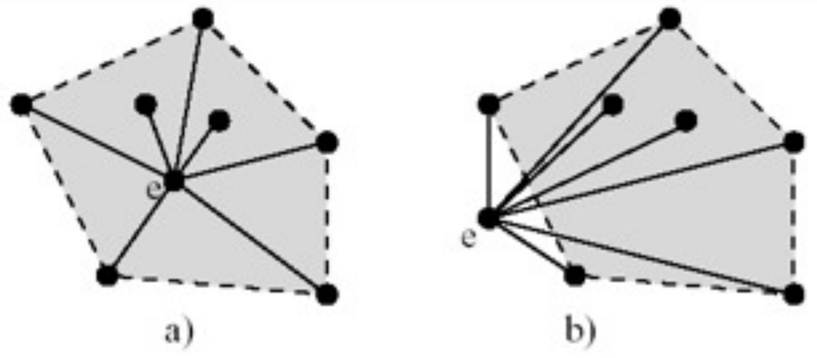
- Example: [Kawahara et al. 2004]



Mutual notification

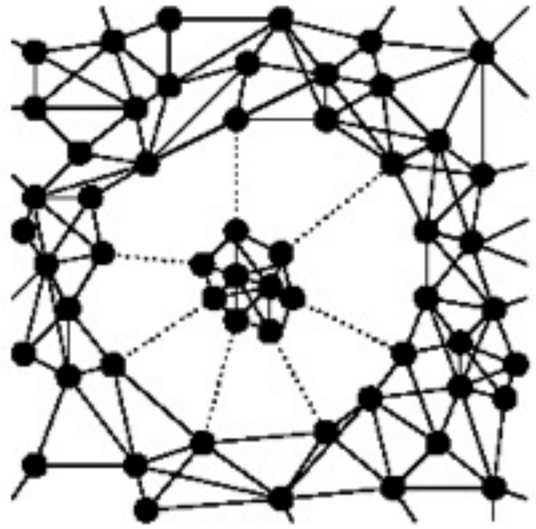
- Each node is only linked with AOI neighbors
  - ie nodes which avatars are in the AOI of the local avatar
- when nodes move they notify each other of the new AOI neighbors
- Example : Solipsis 1 [Keller and Simon 2003]
  - uses the minimum convex hull





Mutual notification

- Example : Solipsis 1
  - if e is inside the convex hull of its external neighbors => no problem
  - if it isn't then it needs to connect to other neighbors
  - additional connections are also maintained so that there is no partition



Mutual notification

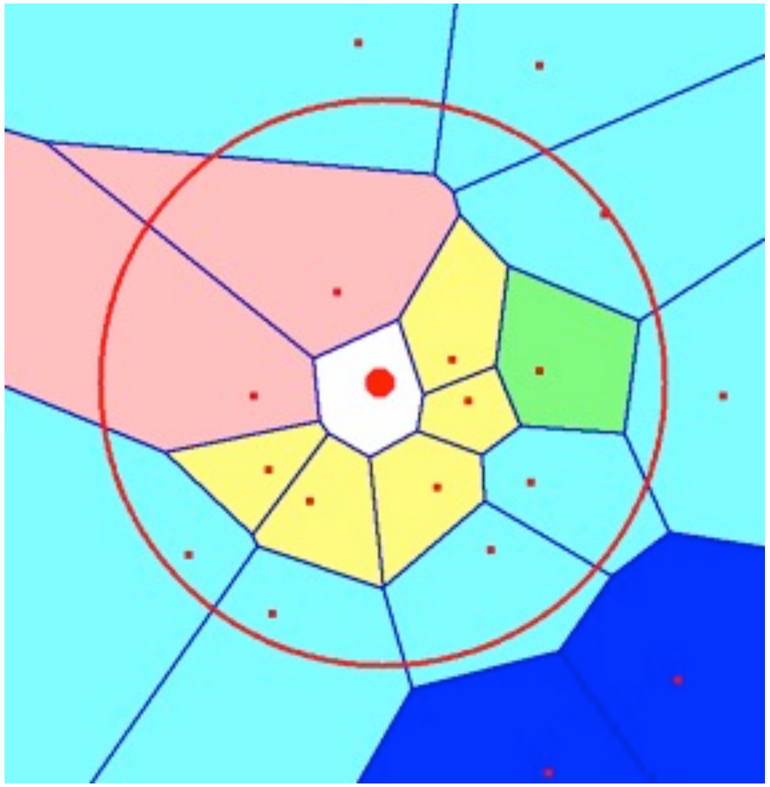
- Example : VON [Hu et al. 2004]
  - Uses Voronoi diagrams to solve the neighbor discovery problem
  - Each node constructs a Voronoi of its neighbors
  - Mutual collaboration in neighbor discovery

Mutual notification

- Example : VON

Mutual notification

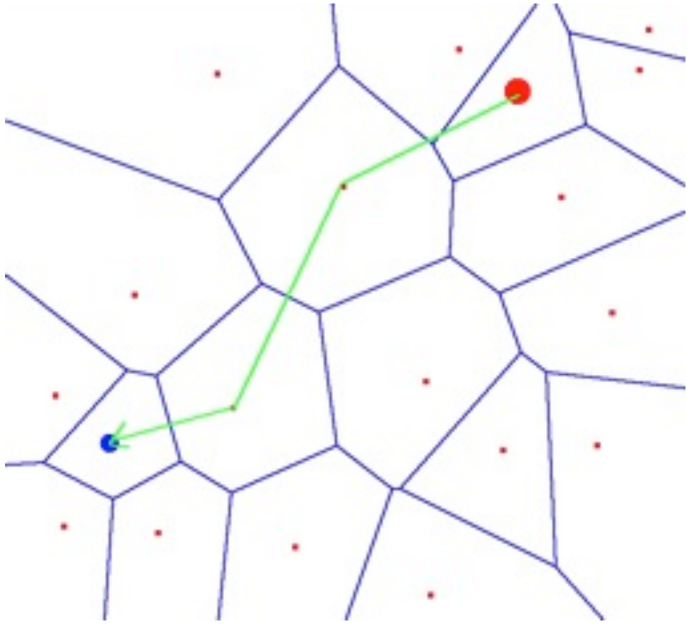
- Example : VON



Circle	Area of Interest (AOI)
White	self
Yellow	enclosing neighbor (E.N.)
L. Blue	boundary neighbor (B.N.)
Pink	E.N. & B.N.
Green	AOI neighbor
D. Blue	unknown neighbor

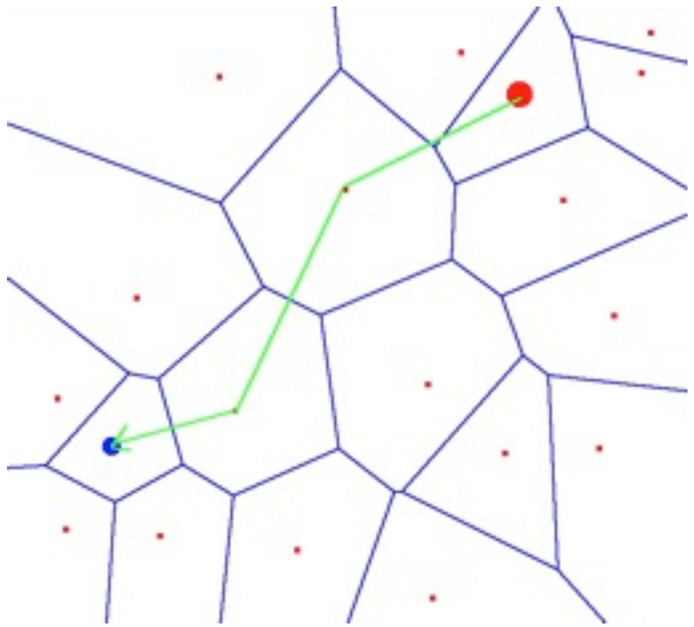
Mutual notification

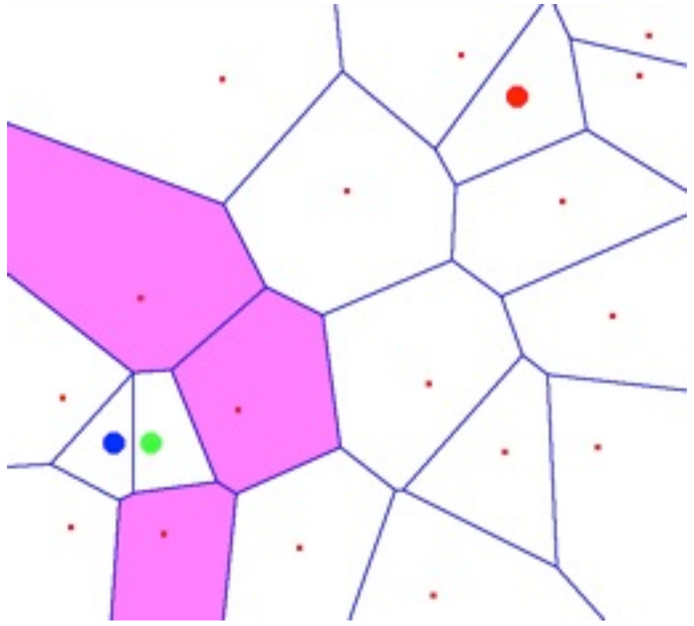
- Example : VON
  - when a new node appears: a JOIN request is forwarded from any node to the closest one



Mutual notification

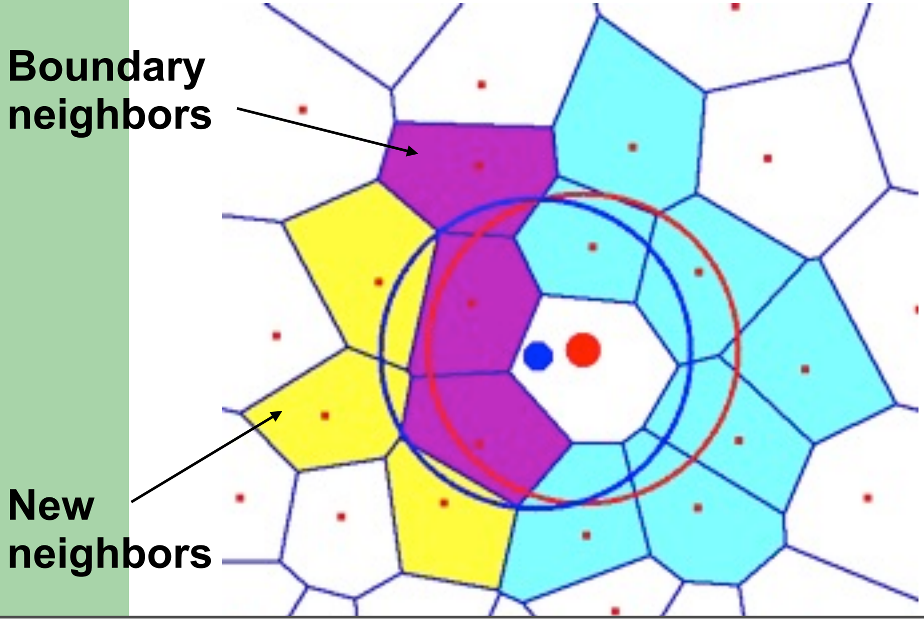
- Example : VON
  - when a new node appears: a JOIN request is forwarded from any node to the closest one





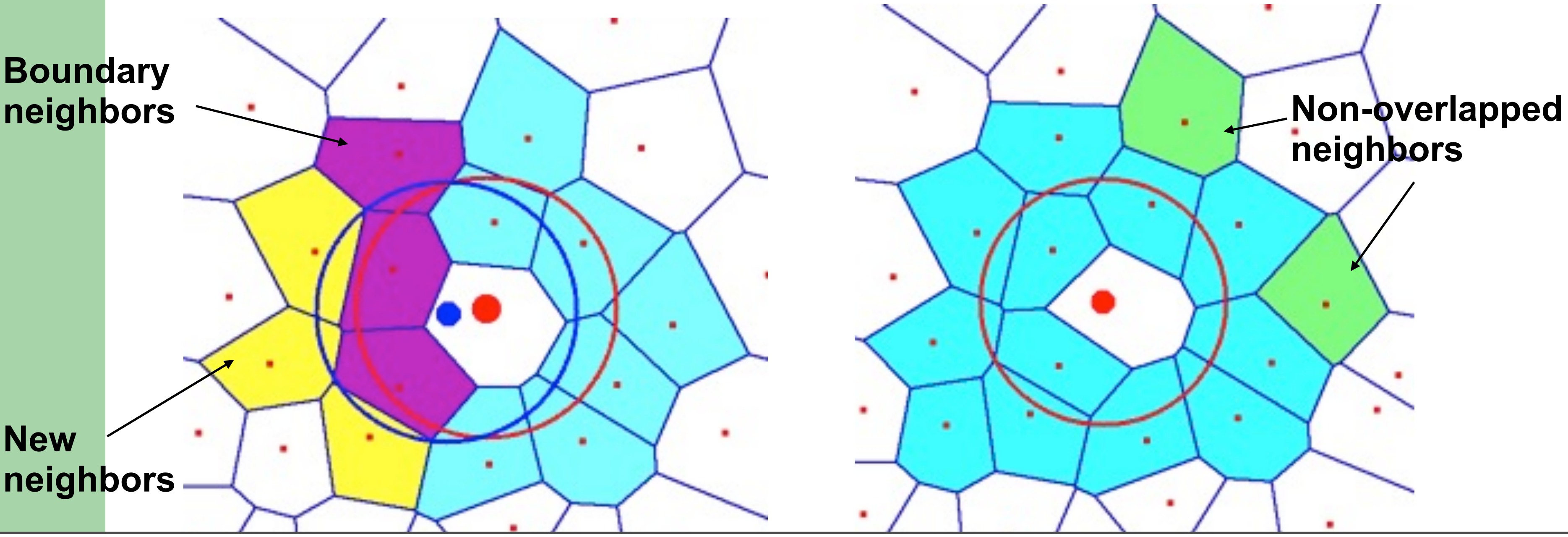
Mutual notification

- Example : VON
  - when a node moves, it sends its position to all its neighbors. B.N. check for overlaps of the AOI with new neighbors. B.N. notify the moving node.



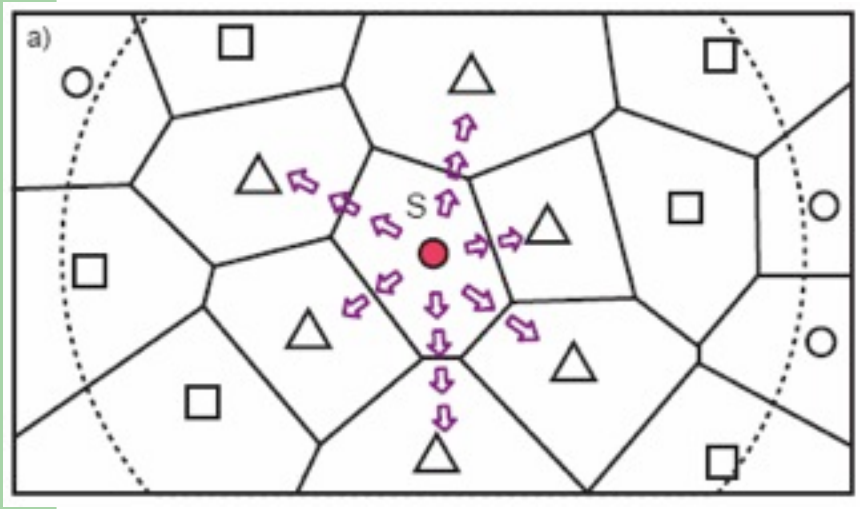
Mutual notification

- Example : VON
  - when a node moves, it sends its position to all its neighbors. B.N. check for overlaps of the AOI with new neighbors. B.N. notify the moving node.



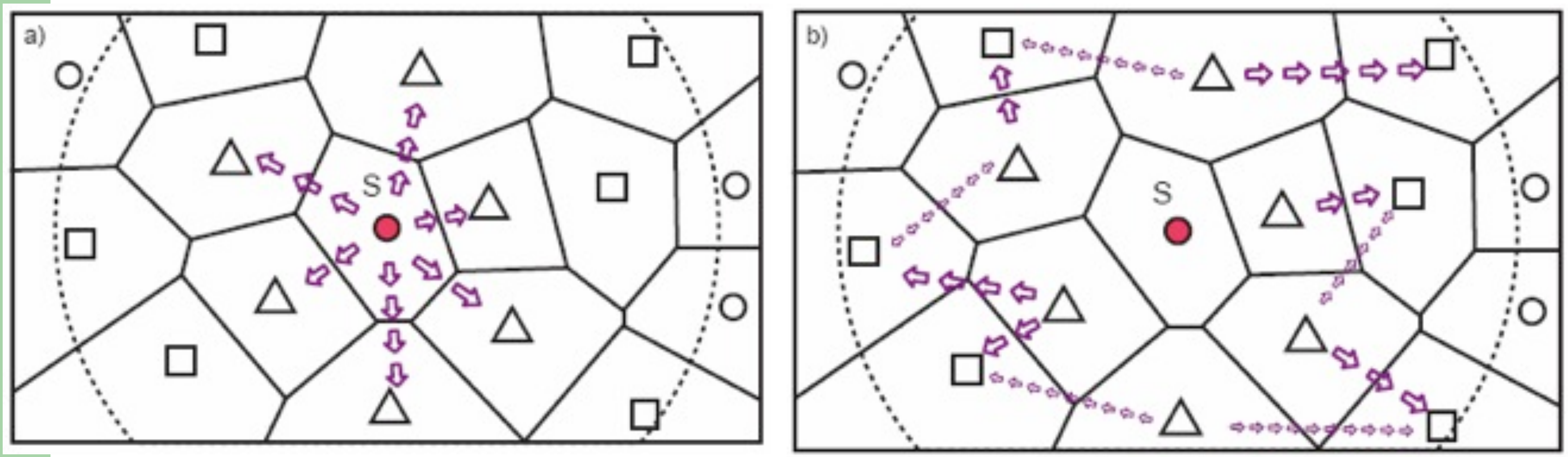
Mutual notification with Overlay Mcast

- Instead of maintaining direct connections with all neighbors => multicast can be used
- Example: VON forwarding [Chen et al. 2007]
  - connect with 1-hop neighbor, forward to others in AOI



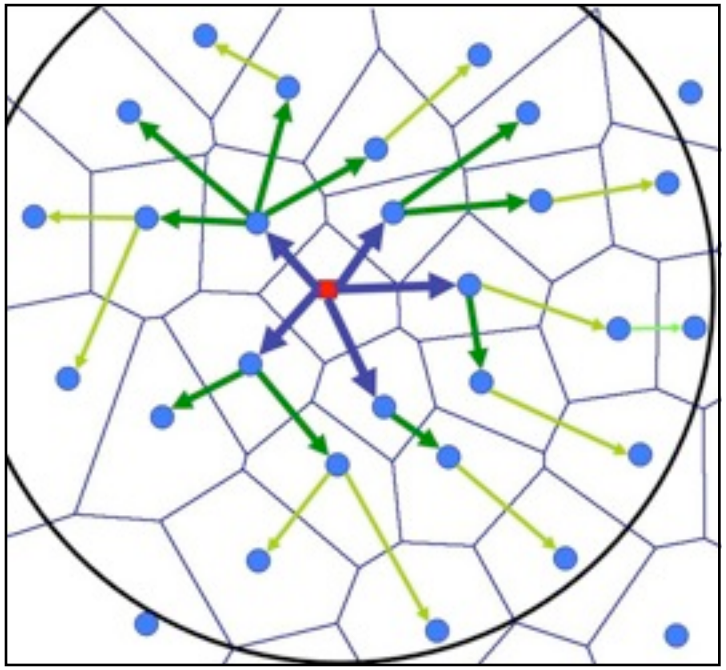
Mutual notification with Overlay Mcast

- Instead of maintaining direct connections with all neighbors => multicast can be used
- Example: VON forwarding [Chen et al. 2007]
  - connect with 1-hop neighbor, forward to others in AOI



Mutual notification with Overlay Mcast

- Example: VoroCast [Jiang et al. 2008]
  - Constructs a spanning tree dynamically



Mutual notification with Overlay Mcast

- Example: FiboCast [Jiang et al. 2008]
  - Adjusts message transmission frequency to distant nodes

	Max_count	*Fibonacci*
1	2	0
2	3	1
3	3	1
4	4	2
5	5	3
6	7	5
7	10 > 8	8
8	∞	∞

hops	1	2	3	4	5	6	7	8	
	✓	✓	X	X	X	X	X	X	-6
	✓	✓	✓	X	X	X	X	X	-5
	✓	✓	✓	X	X	X	X	X	-5
	✓	✓	✓	✓	X	X	X	X	-4
	✓	✓	✓	✓	✓	X	X	X	-3
	✓	✓	✓	✓	✓	✓	✓	X	-1
	✓	✓	✓	✓	✓	✓	✓	✓	0
	✓	✓	✓	✓	✓	✓	✓	✓	0

Interesting Middlewares

- Project JXTA (juxtapose): offers several java libraries that can be used to program P2P overlays http://jxta.dev.java.net/
- The VAST Project: offers an implementation of VON and VSM in C++ (older version in Java) http://vast.sourceforge.net/index.php
- The Badumna Network suite: offers a complete P2P overlay for MMVEs in C# (but can be used through a proxy) http://www.badumna.com/

Thanks

- This keynote is based on:
  - Eng Keong Lua; Crowcroft, J.; Pias, M.; Sharma, R.; Lim, S., "A survey and comparison of peer-to-peer overlay network schemes," Communications Surveys & Tutorials, IEEE , vol.7, no.2, pp. 72-93, Second Quarter 2005
  - A P2P tutorial presented at PDCAT07 by Aaron Harwood http://p2p.csse.unimelb.edu.au/docs/PDCAT07-Tutorial.pdf
  - The VAST project "publications" and "related work" pages (managed by Shun-Yun Hu) http://vast.sourceforge.net/index.php
  - Wikipedia's Distributed Data Sharing category articles http://en.wikipedia.org/wiki/Category:Distributed_data_sharing