University of Bologna

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Engineering Bologna Campus

Class of

Computer Networks M

Multicast, ONs, and MOMs

Antonio Corradi

Academic year 2015/2016

GROUP COMMUNICATION

Communication within a set of processes

Broadcast e Multicast

How to send general messages either to all currently present processes in the system or to a subset of processes (a group) in the system?

In a **single location** you can easily achieve it (in the same LAN)

On different **networks and locations**, you cannot easily achieve it

expressive incapacity, excess overhead, lack of QoS, ...

There are some semantic problems to solve in multicast and broadcast

How to cope **with the answers** (if any)?

- **no wait** – **asynchronous** operations
- wait for **one answer only**
- wait for **some answers only** (how many?, how long?)
- wait for **all answers** (how many? how long? **When to stop**?)

GROUP COMMUNICATION

IP Broadcast

Broadcast limited and directed (inside local network)

IP Multicast **heavier duty and protocol**

Multicast for class D addresses

Local Multicast support and ...

Internet uses **Internet Group Management IGMP** protocol since long ago (**RFC 1112** e **2236**) to **implement local multicast**

Often the **protocol could operate only on local subnetworks**, and it is implemented in different and not compatible forms

Multicast (more) global support

A multicast is realized by **flooding between networks**

a packet can traverse a node only once (node with state) and **is sent via any output queue apart from the one where it came in (how long to keep the state?)**

Traditional way of routing with simple and low cost (!) policies

IP GROUP COMMUNICATION

One can adopt some basic strategies with mechanisms

For example, we can use an a-priori dimensioning of time-to-live (TTL) of datagrams (to specify penetration and cost)

TTL=0 local send	TTL=1 local to connection
TTL<=32 local to area	TTL<=64 local to region
TTL<=128 local to continent	TTL>128 global

IP Multicast and the QoS?

How can we be sure that the message has been delivered (beyond best-effort semantics)?

There is a limited guarantee on IGMP implementations

that is

We do not know if messages were all delivered to all recipients and in which order

GROUP COMMUNICATION

IGMP as an example of local support to Multicast

(RFC 1112 e 2236)

IP multicast allows to send a unique packet to multiple receiver in the same locality, by using class D names to identify a group, not necessarily a local one but spanning a few local networks

The IGMP needs a **support from management router**

Every local network must hosts at least an IGMP router capable of managing local incoming and outgoing traffic and it controls the group with IGMP messages. It is possible to provide more multicast routers

IGMP v1 considers only **two simple messages** with C/S approach

IGMPQUERY	a **router** periodically verifies the existence of hosts that answer to a specific IP D address
IGMPREPORT	a **node** signals a state change to the router related to the group (only **join the group** and no **leave**)

IGMP VERSIONING

IGMP v1

Routers are in charge of group management

There is only a **join** message**,** but no **leave** message from the group in v1
Any router has always an **active** role that require to regularly emit queries: nodes reply to the query to signal their presence or do not reply (problem with nodes that **answer late to the first join query**)

this version requires group operations (*only one single report from a node for a single local network*)

IGMP v2 (support for join / leave)

The second version consider the capability of nodes to send a message of **explicit leave** (i.e., leave the address group)

Nodes that leave the group must notify the manager

More routers can be in charge of the management

Interference between router is settled with IP numbers order

ROUTING MULTICAST PRINCIPLES

Multicast must employ the least resources as possible during data transmission to receivers

Some assumptions tend to obtain an optimal use of resources and to avoid an excess of bandwidth

- **single sender** support
- **variable number of receivers** support (**up to n**), that can be added or removed dynamically

The main idea is to maximize **sharing,** so to **send only one copy**, instead on N ones, of the same **multicast** message (1 message cost) **instead of different unicast** (**N** message cost)

Derived from assumptions, protocols identify a central **tree** starting from the sender with **optimal shared paths from sender to current receivers**

The goal is to employ most shared hops as possible from root to leaves

the continuously changing tree must consider only currently active receivers and disregard the ones where there are no currently active receivers

ROUTING MULTICAST (STABLE)

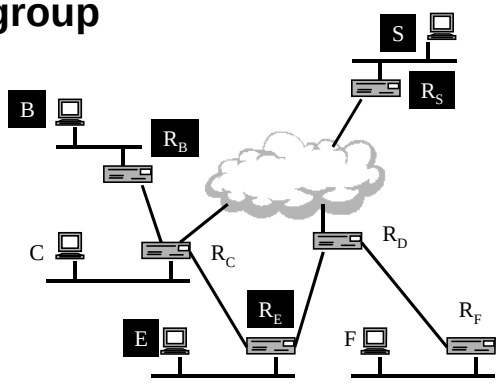
Multicast requires the **identification** of a (**dynamic**) **tree** from **sender to receivers for repeated forwarding**

the sender is the **root of the tree**, the intermediate routers are the **intermediate** nodes and identify subtrees**,** the receivers are the **leaf nodes** in the tree

- an open group of nodes with a **single sender**
- the group membership is **dynamic**
- Leaves are responsible for **joining the group**
- **shared paths optimize bandwidth**

The tree is **extremely dynamic**

Consider the case where an **host S** transmits and **B** and **E** are in the receiver group



MULTICAST: SPANNING TREE

We consider only routers as participants (no nodes) and **we want to build a tree from the interconnection graph**

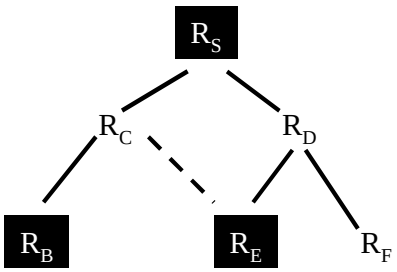
First step (request for leaf identification: root to leaves)

We want to build a tree (a **spanning tree**) that connects root to known leaf nodes, typically by using unicast routing protocol information and **organizing and aggregating paths**

We start sending a **flooding** message towards **every** possible recipient with the main objective of creating a **bone multicast**

The root identifies shortest paths by building it from replies from receivers

some receivers nodes are reached through multiple paths



MULTICAST: MULTIPLE PATHS

Second step (go back from leaves to root)

Every leaf signals direct **paths** (backwards) and can also **identify new paths** (even not shortest) for going from root to leaves

minimal path messages are sent backward from leaves to root

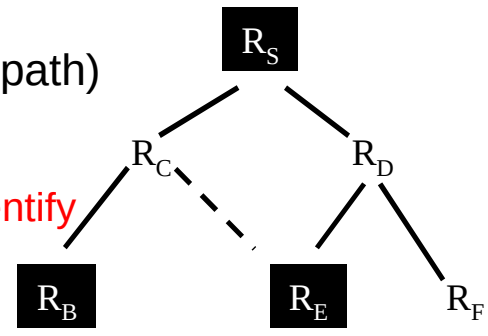
only some paths are selected, other are discarded

some *shortest path messages* from the source are sent back in a **larger scope**: they are forwarded from leaves on all exit links, except the one where it was coming (to identify other better paths not traversed from root to leaves)

Reverse path forwarding (backward path)

For every router reached from several path, the root can so select the best

Re is reached from Rd but it can try to identify other routes in order to determine new shared parts



MULTICAST in ACTION

Normal routing: normal routing operation must work continuously while **tree identification is ongoing…**

Distance Vector

Next hop information must be used (or use poisoned reverse) in order to block too long paths

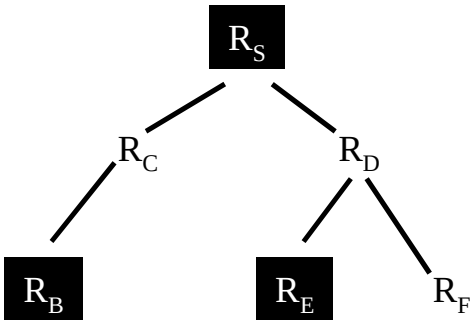
Link State

All shortest path trees must be built for every node and use “tie break” rules to settle conflicts

Reverse Path Broadcast (2 step) for **deleting Multiple Paths**

Leaves send a broadcast **towards the root** during normal routing operations

The root receive new paths and can reorganize the tree trying to aggregate several sub-paths and produce an optimal tree



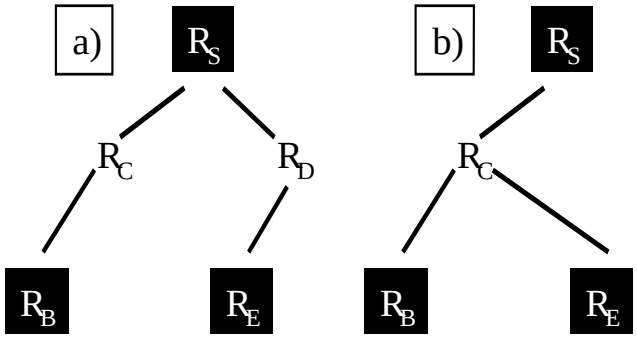
REVERSE PATH BROADCAST

Reverse Path Broadcast allows to choose between different paths to organize the optimal tree, while minimizing the number of sent messages and used bandwidth

With a broadcast from leaves (the **Reverse Path Multicast**) it is possible to **find paths, connecting leaves with the root, that have not been previously explored**

It is up to the root to choose the best tree organization

Reverse Path Multicasting (**RPM**) to reorganize the tree (even with a high cost)



MULTICAST: PRUNING and GRAFTING

PRUNING and GRAFT

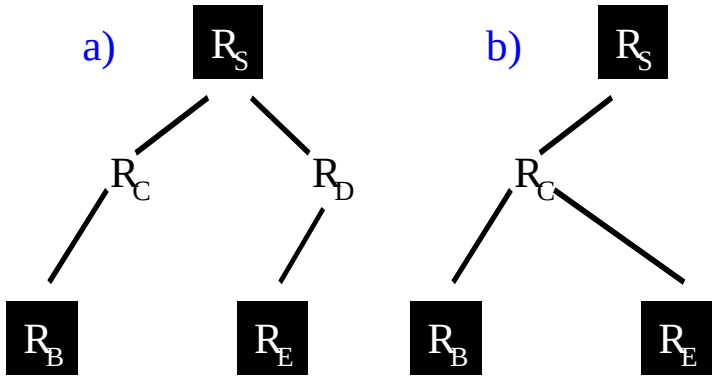
routers that have no receivers connected are excluded with '**cut**' messages that flows throughout the tree

The tree must be rebuilt in case of any modification

Reverse Path Multicasting (**RPM**) autonomously done by the leaves to consent

PRUNING - from a) to b) and reinserting parts of the tree

GRAFT - from b) to a)



REVERSE PATH MULTICAST

Reverse Path Multicasting from leaves to root (not a broadcast)

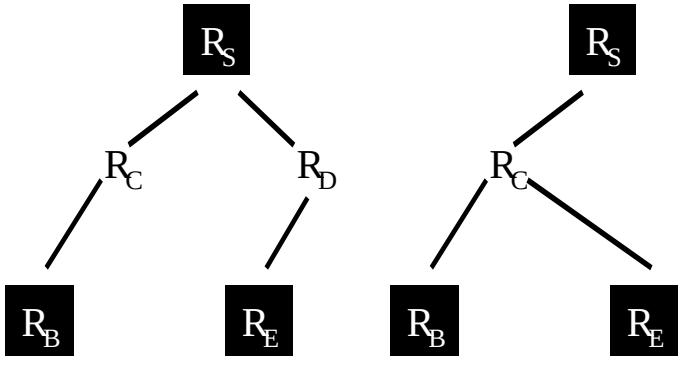
- used in a lot of multicast protocols
- keeps the state for communication **per-sender**, **per-group**

Networks with no members are **pruned out** from the tree and new ones can reenter the group (**explicit graft** from the bottom) without reorganizing the tree from scratch

The state (software) is kept for a limited and predetermined time

SOFT-STATE

The definition of the **RPM time interval** is **critical**



DIFFERENT MULTICAST PROTOCOLS

There are many different **routing multicast** protocols, **incompatible** with each other, even in competition between themselves and supported by different communities

DVMRP (**RFC 1075**) **D**istance **V**ector **M**ulticast **R**outing **P**rotocol

Employs RPM, based on a modified version of RIP and very used in MBONE (multicast backbone)

- Update messages are sent using special paths (tunnel) and using only some nodes

MOSPF (RFC 1584) **M**ulticast **O**pen **S**hortest **P**ath **F**irst Protocol

Extends link-state, suitable for big networks, based on RPM and soft-state

- It starts from networks map and uses them to calculate shortest path to every single destination
- It optimizes the trees and removes not used paths

MANY STANDARD MULTICAST

PIM (**RFC 2117**) **P**rotocol **I**ndependent **M**ulticast Protocol

Uses any unicast protocol in different ways so to suit different systems

Scattered intended when there is a low probability of multiple nodes on the same LAN and **Dense** where there are many neighbors routers

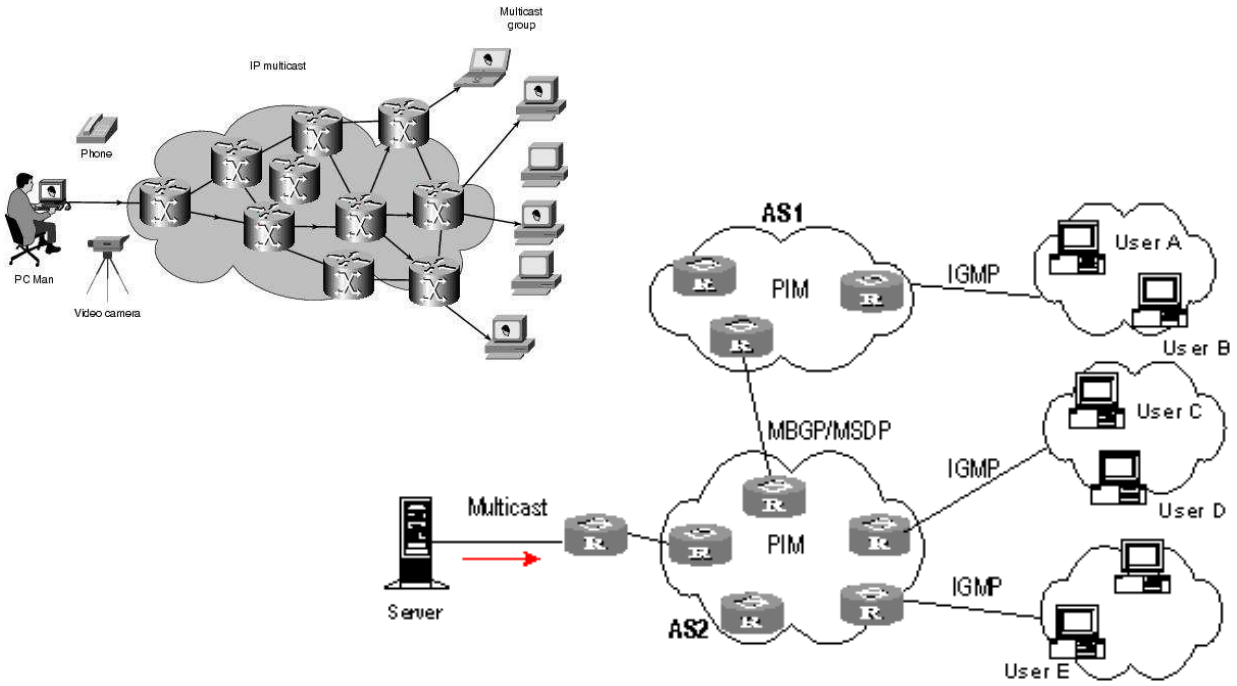
- **Scattered**: removing the most number of intermediate router to simplify the tree structure
- **Dense**: use of flooding and prune, simplified with regard to DVMRP

CBT (**RFC 2201**) **C**ore **B**ased **T**rees

suitable for an organization based on core routers to choose

- Some **nodes are fixed** (core) and **trees are unified** without defining a per-sender or per-group state
- It is possible to use sub-optimal tree organizations to avoid reorganizing connection for every multicast reconfiguration

MULTICAST PROTOCOLS



OVERLAY NETWORKS

There are many situations where you want to organize a **logical connection between different entities that reside in different locations and networks**

The solution is an **overlay network at the application level that connects all those entities to be considered together in an ON**

Overlay networks may be very different and also enforced in different ways, but their importance is paramount in many situations

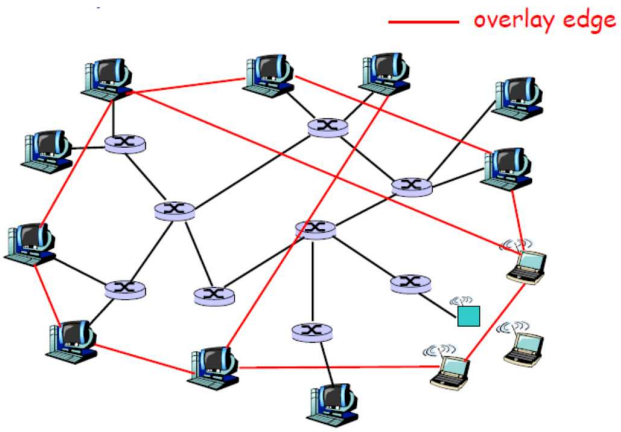
One main point very important is not only **organizing** it, but also to **grant QoS and respect an agreed SLA**

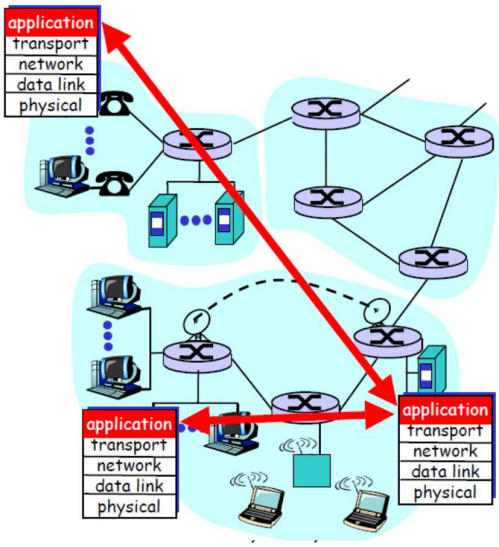
That is the reason why there are many different solutions for different cases, and also many different solutions and tools embodying these requirements

OVERLAY NETWORKS

The main point is to create a **new network at the application level** and to maintain it with **specified requirements**

All participants become part of it and can communicate freely (the same as if they were in a real network connection), by using an **application neighborhood**





OVERLAY NETWORKS PROPERTIES

Overlay network imply many challenges to cope with

- **Maintaining** the edge links (via IP pointer?)
- **Favoring** the insertion in the neighborhood
- **Checking** link liveness
- **Identifying** problems and faults
- **Recovering** edges
- **Overcoming** nodes going down and their unavailability
- **Re-organizing the overlay**, when some nodes leave the network and other nodes get in
- **Keeping the structure**, despite mobile nodes intermittent presence (and eventual crashes or leaving)
- **Creating a robust connection**, independently of omissions and crashes (QoS?)

CLASSIFICATION of OVERLAY NETWORKS

There are two main different kinds:

- **Unstructured overlays**
- **Structured overlays**

By focusing on new nodes arriving and entering the ON,

in *Unstructured overlays*, new **nodes choose randomly the neighbor to use to access** to the ON

in *Structured overlays,* there is **a precise strategy to let nodes in** and to *organize the architectures, maintained also to react to discontinuities and failures*

ONs propose solutions for P2P applications, but also for MOMs (even if **statically-oriented)**

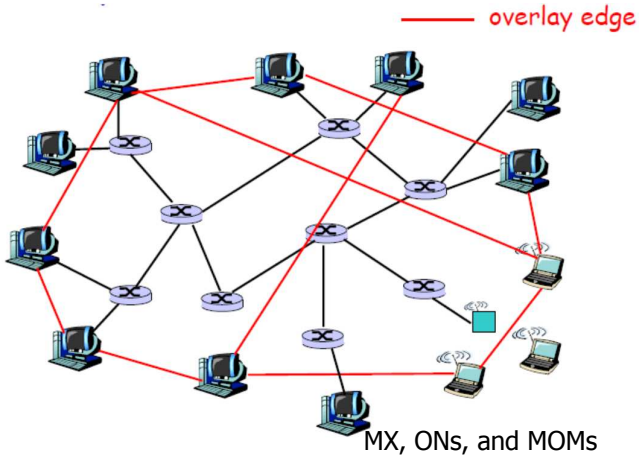
P2P	Napster, Gnutella, Kazaa, BitTorrent
Support	Chord, Pastry/Tapestry, CAN
Social Nets	MSN, Skype, Social Networking Apps

OVERLAY NETWORKS: USAGE

A good **overlay network** has the goal making **efficient the operations among the group of current participants** obeying the **specific requirements**

All participants in an overlay have **a common goal of exchanging information**, for instance…

They tend to **exchange data**: files in a P2P application, messages in social nets, specific application protocols in other environments, etc.



SYSTEM AND APPLICATION KEY ISSUES

ONs should organize the communication support and also enable the application level management

Throughput

- Applications over an ON need content distribution /dissemination
- How to replicate content … fast, efficiently, reliably

Lookup

- How to find out very fast the appropriate user information (content/resource) on the ON

Management

- How to maintain efficiently the ON under a high rate of connections/disconnections and intermittent failures in load balanced approach
- Both application reliability and availability is very difficult to guarantee: a self-organizing approach is typically followed

NAPSTER

A non structured approach for file retrieving

Centralized Lookup

Centralized directory services deal with nodes entering

Any node connects to a Napster server

Any node uploads list of files to server

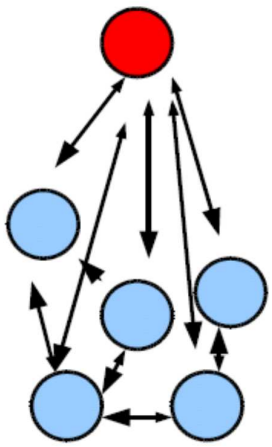
Any node gives servers keywords to search the full list with

File exchange peer to peer

Lookup is centralized from servers, but files copied P2P

Select “best” of correct answers (announce by ping messages)

Performance Bottleneck and low scalability



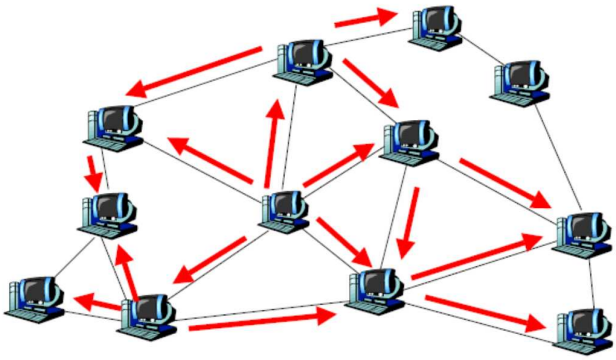
GNUTELLA

GNUTELLA is the main representative of **unstructured ORs,** by providing a **distributed approach** in **file retrieval**

Fully decentralized organization and **lookup** for files

- There are nodes with different degrees of connections and availability (from high-degree nodes to low-degree ones)
- High-degree nodes may receive even more links

Flooding based lookup, obviously **inefficient** in terms of scalability and bandwidth



GNUTELLA : SCENARIO

Step 0: Join the network

Step 1: Determining who is on the network

- **"Ping"** packet is used to announce your presence on the network.
- Other peers respond with a **"Pong"** packet and Ping connected peers
- A Pong packet also contains:
  - IP address, port number, amount of data that peer share
  - Pong packets come back via same route

Step 2: Searching

- Gnutella "**Query**" ask other peers (**N** usually **7**) for desired files
- A Query packet might ask, ***"Do you have any matching content with the string 'Volare'"?***
- Peers check to see if they have matches & respond (if they have any match) & send packet to connected peers if not (**N** usually **7**)
- It continues for **TTL** (**T** specifies the hops a packet can traverse before dying, typically **10**)

Step 3: Downloading

- Peers respond with a "**QueryHit**" (it contains contact info)
- File transfers via direct connection using **HTTP** protocol's **GET** method

GNUTELLA REACHABILITY

An analytical estimation of reachable users

T: TTL, *N*: Neighbors for Query

	T=1	*T=2*	*T=3*	*T=4*	*T=5*	*T=6*	*T=7*
N=2	2	4	6	8	10	12	14
N=3	3	9	21	45	93	189	381
N=4	4	16	52	160	484	1,456	4,372
N=5	5	25	105	425	1,705	6,825	**27,305**
N=6	6	36	186	936	4,686	**23,436**	**117,186**
N=7	7	49	301	1,813	**10,885**	**65,317**	**391,909**
N=8	8	64	456	3,200	**22,408**	**156,864**	**1,098,056**

GNUTELLA SEARCH

GNUTELLA Versions has adopted different scalability protocols

Flooding based search is extremely wasteful with bandwidth

- Enormous number of redundant messages (not efficient)
- A large (linear) part of the network is covered irrespective of hits found, without taking into account needs
- All users do searches in parallel: local load grows linearly with size

Taking advantage of the unstructured network some more efficient protocols appear

- **Controlling topology** for better search
  Random walk, Degree-biased Random Walk
- **Controlling placement of objects**
  Replication

GNUTELLA RANDOM WALK

Basic strategy based on high degree nodes

Scale-free graph is a graph whose degree distribution follows a **power law**

High degree nodes can store the index about a large portion of the network and are easy to find by (biased) **random walk** in a **scale-free graph** as a situation of random offer of files

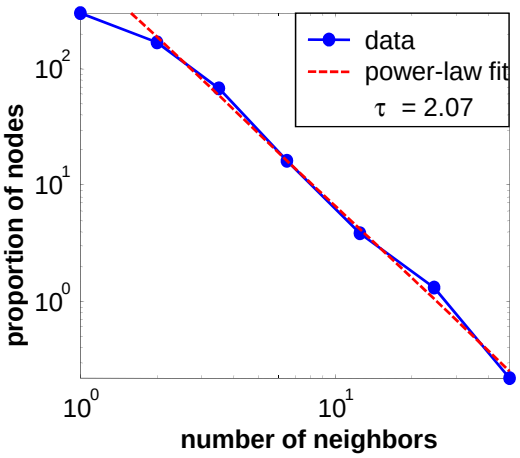
High degree nodes have a **neighborhood** of **low degree ones**

Random walk

Moves random to avoid to visit always last visited node

Degree-biased random walk

- Select highest degree node that has not been visited
- Walk first climbs to highest degree node, then climbs down on the degree sequence
- Optimal coverage can be formally proved



GNUTELLA REPLICATION

The main idea is to spread copies of objects to peers so that **more popular objects can be found easier and, to more likely find, also lunch more walks**

Replication is both in sense of **more copies of data** and also in terms of **more walkers**

Replication strategies

Replicate with i when q_i is the number of query for object i

Owner replication

- Produce replicas in proportion to q_i

Path replication

- Produce replicas over the path square root replication to q_i

Random replication

- Same as path replication to q_i, only using the given number of random nodes, not the path

But it is still difficult to find rare objects

UNSTRUCTURED VS STRUCTURED

To go deep into ON organization…

- **Unstructured P2P networks** allow resources to be placed at any node spontaneously
  The network topology is arbitrary and the growth is free
- **Structured P2P networks** simplify resource location and load balancing by defining a topology and defining rules for resource placement to obtain **efficient search for rare objects**

Which strategies and rules???

↓

Distributed Hash Table (DHT)

HASH TABLES

Distributed hash tables use Hash principles toward a better retrieval of data content and value

Store **arbitrary keys and connected data** (value)

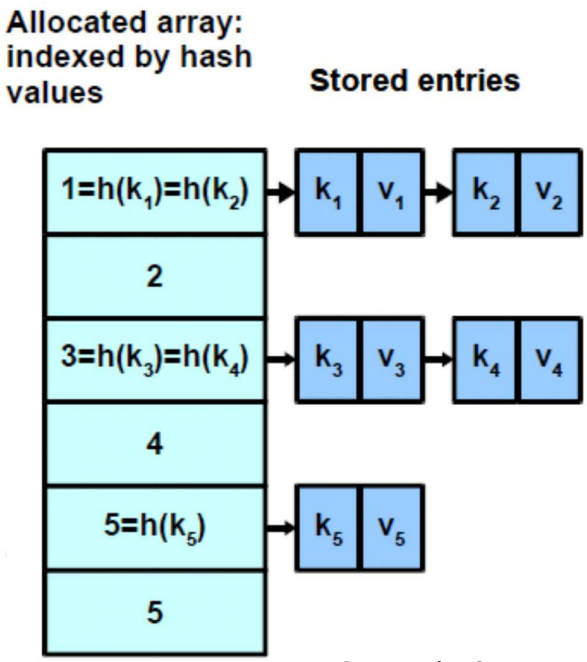
- put (key,value)
- value = get(key)

Lookup must be fast

- Calculate hash function h() on key that returns a storage cell

Chained hash table

- Store key in the chain (together with optional value)



DISTRIBUTED HASH TABLES

Hash table functionality in an ON is typically P2P:
lookup of data indexed by keys very efficient

Key-hash → node mapping

- Assign a unique live node to one key
- Find this node in the overlay network quickly and cheaply

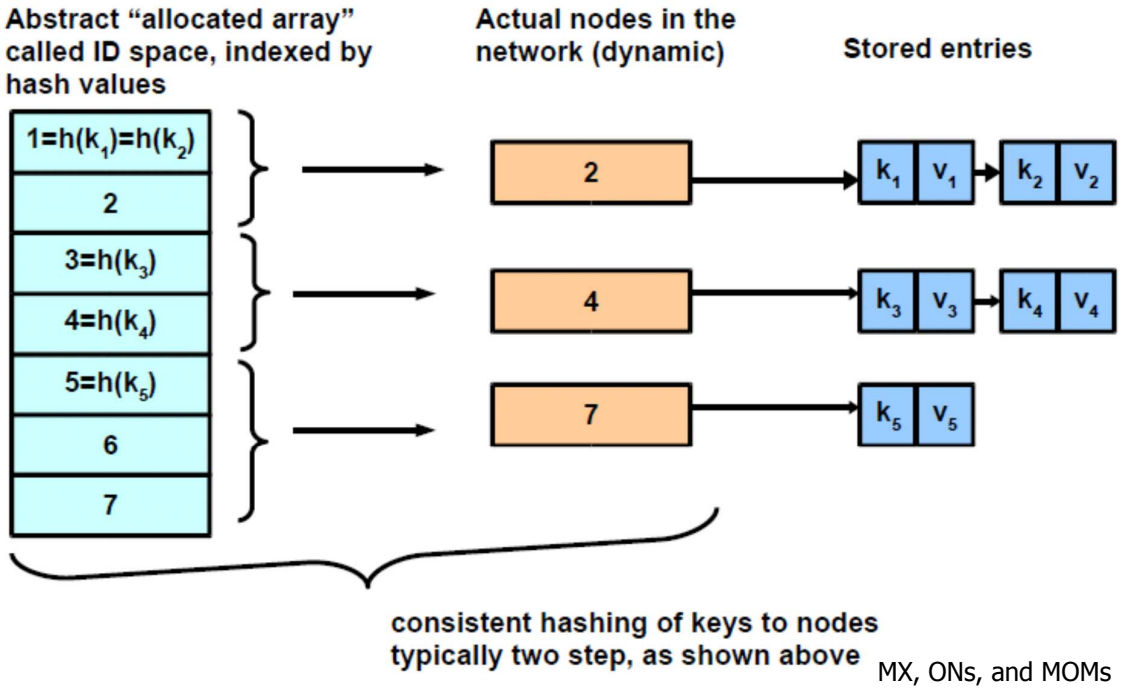
Support **maintenance of the ON** and **optimization of its current organization**

- **Load balancing**: maybe even change the key-hash → necessity of node mapping on the fly
- **Replicate entries on more nodes** to increase availability

DISTRIBUTED HASH TABLES

Find the **best node allocation** depending on **existing nodes**

Nodes can enter and leave the OR



STRUCTURED HASH TABLES

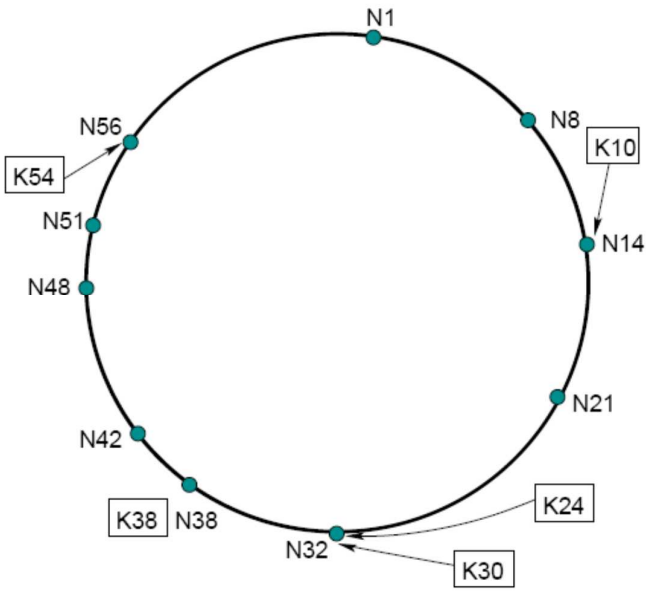
Many examples of tools for supporting DHT

- **Chord**
  Consistent hashing ring-based structure
- **Pastry**
  Uses an ID space concept similar to Chord
  But exploits the concept of a nested group
- **CAN**
  Nodes/objects are mapped into a d-dimensional Cartesian space

CHORD HASH TABLES

Hash is **applied over a dynamic ring**

- Consistent hashing based on an **ordered ring overlay**
- Both keys and nodes are hashed to 160 bit IDs (SHA-1)
- keys are assigned to nodes by using **consistent hashing**
  - **Successor in ID space**



CHORD CONSISTENT HASHING

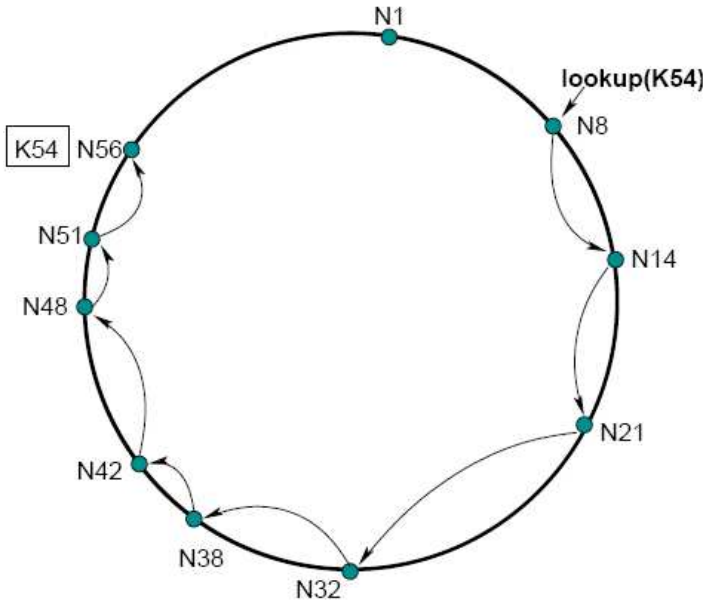
CHORD works on the idea of making **operations easier**

- **Consistent hashing**
  - **Randomized**
    - All nodes receive roughly equal share of load
  - **Local**
    - Adding or removing a node involves an O(1/N) fraction of the keys getting new locations
- **Actual lookup**
  - Chord needs to know only **O(log N)** nodes in addition to successor and predecessor to achieve **O(log N)** message complexity for lookup

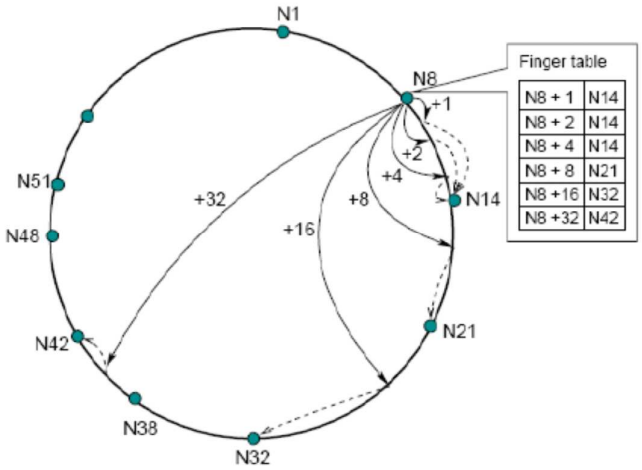
CHORD PRIMITIVE LOOKUP

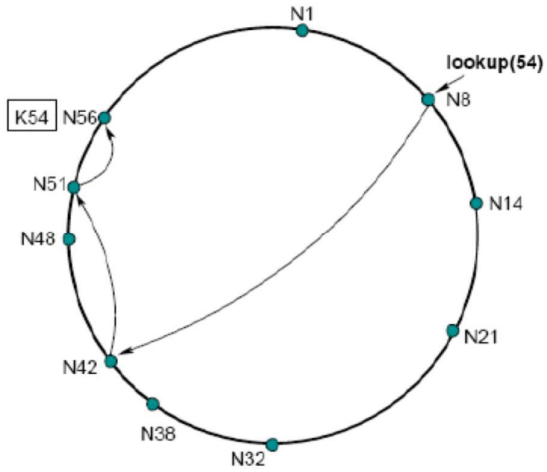
Lookup query is **forwarded to the successor in one direction (one way)**

- Forward the query **around the circle**
- In the worst case, O(N) forwarding is required
  - In two ways, O(N/2)
- CHORD may keep **finger table** to identify faster the node (finger tables as caches for the successors)



CHORD SCALABLE LOOKUP



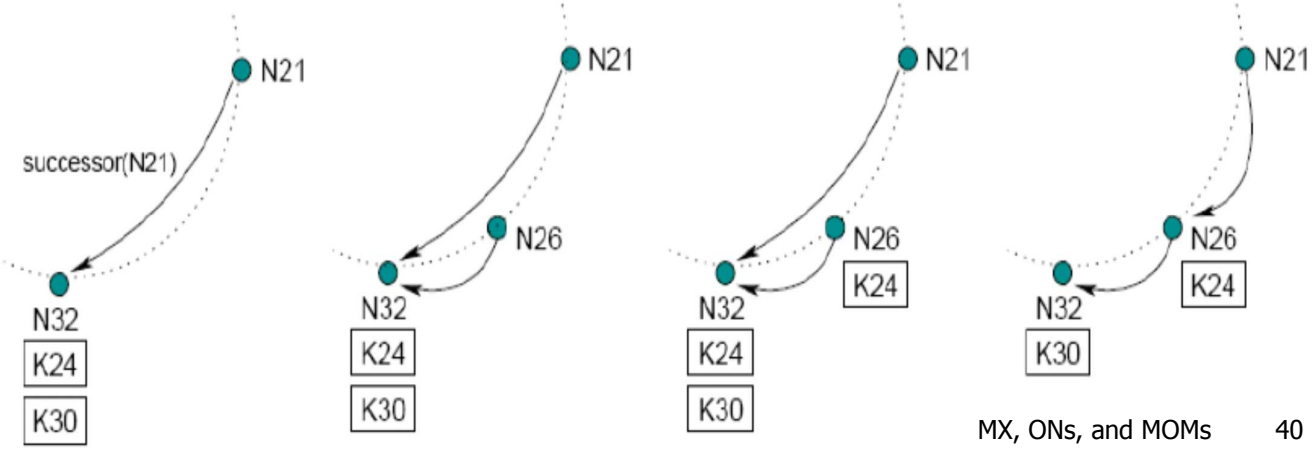


The i_{th} entry of a finger table points the successor of the key (nodeID + 2^i)

A finger table has O(log N) entries and the scalable lookup is bounded to O(log N)

CHORD NODE JOIN

- A new node has to
  - Fill its own **successor, predecessor and fingers**
  - Notify other nodes for which it can be a successor, **predecessor of finger**
- Simple way: **find its successor, then stabilize**
  - Join immediately the ring (lookup works), then modify the structure organization



CHORD STABILIZATION

If the ring is correct, then routing is correct, **fingers are needed for the speed only**

- **Stabilization**

The support monitors the structure and organizes itself by controlling the OR freshness

  - Each node periodically runs the stabilization routine
  - Each node refreshes all fingers by periodically calling find_successor($n+2^i-1$) for a random i
  - Periodic cost is O(logN) per node due to finger refresh

CHORD FAILURE HANDLING

Failed nodes are handled by

- **Replication**: instead of one successor, we keep R successors
  - **More robust to node failure** (we can find our new successor if the old one failed)
- **Alternate paths while routing**
  - If a finger does not respond, take the previous finger, or the replicas, if close enough

At the DHT level, we can replicate keys on the **r successor nodes**

- The stored data becomes equally more robust

PASTRY

PASTRY is a DHT similar to CHORD in a more organized way

- Applies a sorted ring in ID space (as in Chord)
  - Nodes and objects are assigned a 128-bit identifier
- NodeID interpreted as a sequence of digit in **base 2^b**
  - In practice, the identifier is viewed in Hex (base 16)
  - **Nested groups as replication entities**
- The node that is responsible for a key is **numerically closest** (not the successor)
  - Bidirectional sequencing and by using numerical distance
- Applies **Finger-like shortcuts** to speed up lookup

PASTRY

PASTRY keeps

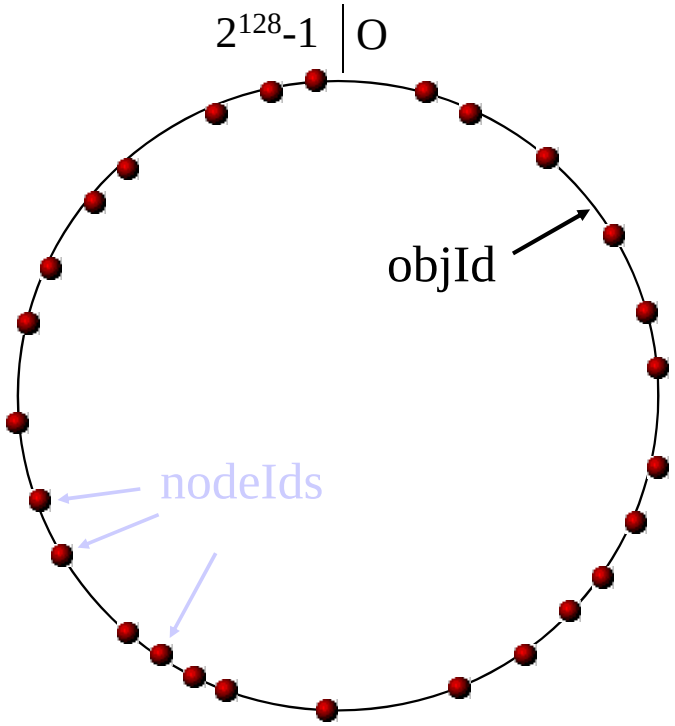
Routing tables to explore **proximity** and find **close neighbors**

Leaf sets to maintain IP addresses of **nodes with numerically closest larger and smaller nodeIds**

Generic P2P location and routing substrate

- Self-organizing overlay network
- Lookup/insert object in < $log_{16} N$ routing steps (expected)
- *O(log N)* per-node state
- Network proximity routing

PASTRY: OBJECT DISTRIBUTION



Consistent hashing

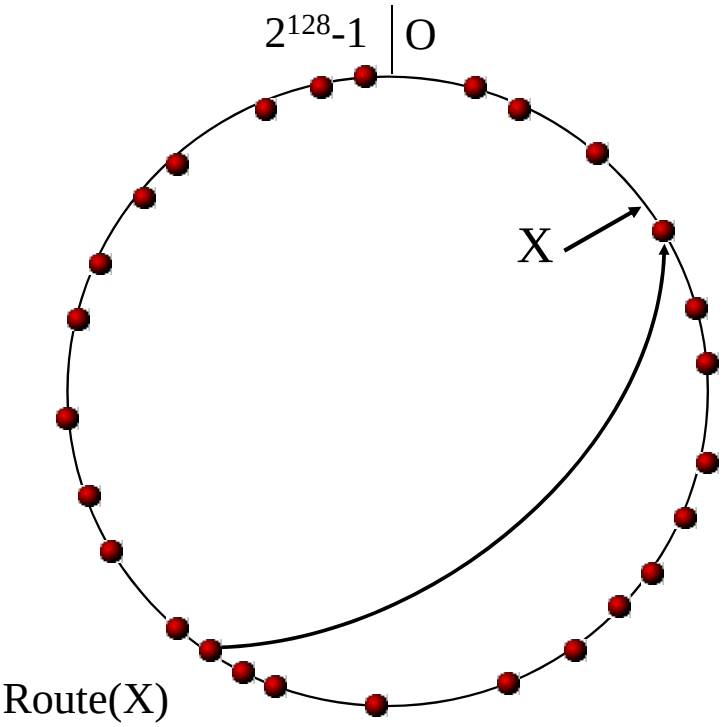
128 bit circular id space

nodeIds (uniform random)

objIds (uniform random)

Invariant: nodes with numerically closest nodeId maintain objects

PASTRY INSERT / LOOKUP



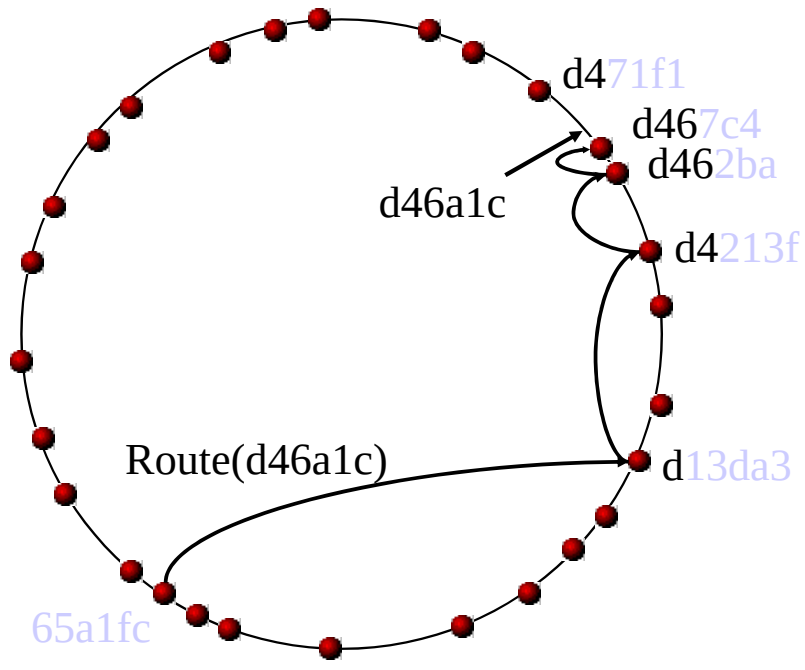
Msg with key X is routed to live node with nodeId closest to X

Problem: complete routing table not feasible

PASTRY ROUTING TABLE (# 65A1FC)

Row 0	0x	1x	2x	3x	4x	5x		7x	8x	9x	ax	bx	cx	dx	ex	fx
Row 1	60x	61x	62x	63x	64x		66x	67x	68x	69x	6ax	6bx	6cx	6dx	6ex	6fx
Row 2	650x	651x	652x	653x	654x	655x	656x	657x	658x	659x		65bx	65cx	65dx	65ex	65fx
Row 3	65a0x		65a2x	65a3x	65a4x	65a5x	65a6x	65a7x	65a8x	65a9x	65aax	65abx	65acx	65adx	65aex	65afx
$\log_{16}$ N rows																

PASTRY ROUTING



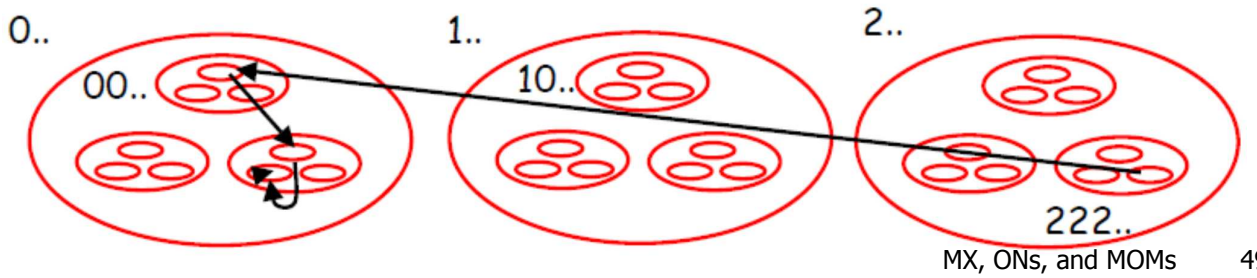
Properties

- $\log_{16}$ N steps
- O(*log N*) state

PASTRY NESTED GROUPS

Simple example: nodes & keys have n-digit **base-3** ids, e.g., 02112100101022

- There are **3** nested groups for each group
- Each node knows IP address of one delegate node in some of the other groups
- Suppose node in group 222… wants to lookup key k= 02112100210
  - Forward query to a node in 0…, then to a node in 02…, then to a node in 021…, then so on.



PASTRY ROUTING TABLE AND LEAFSET

Routing table

- Provides **delegate nodes** in **nested groups**
- **Self-delegate** for the nested group where the node belongs to
- O(log N) rows
  → O(log N) lookup

Leaf set

- Set of nodes that are numerically closest to the node, the same as Successors in Chord
  - L/2 smaller & L/2 higher
- Replication boundary
- Stop condition for lookup
- **Support reliability and consistency**

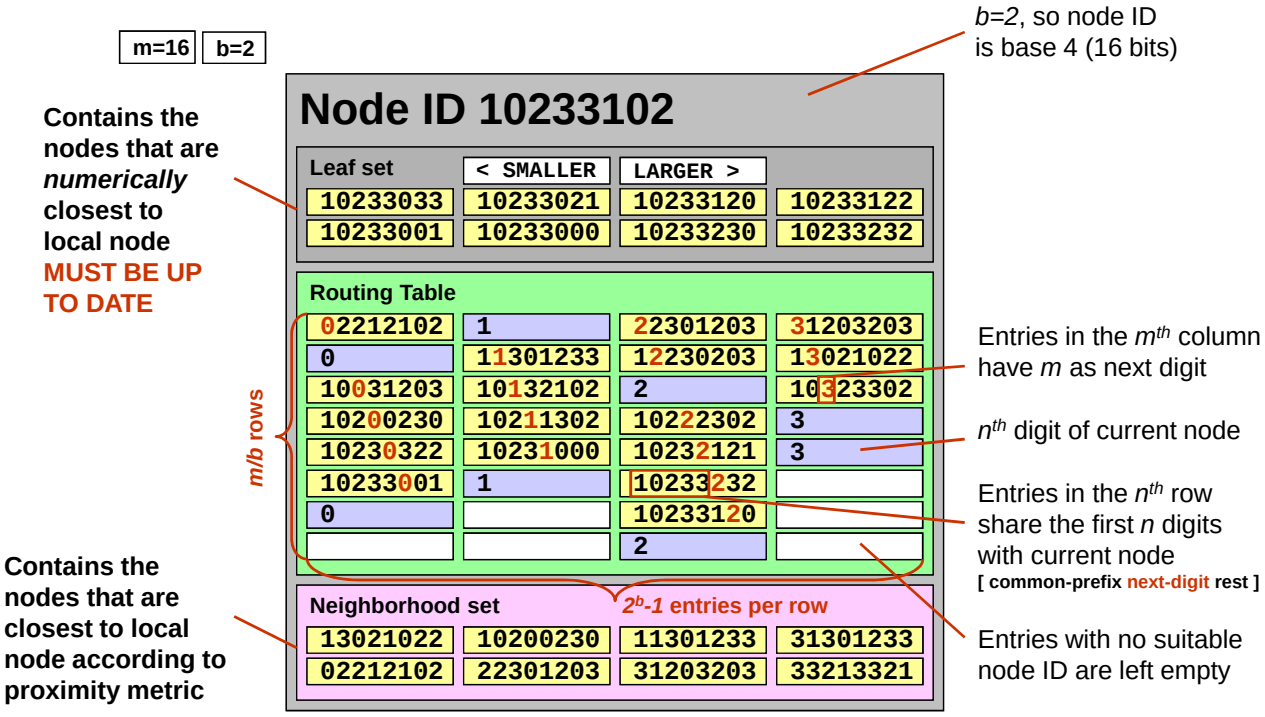
Base−4 routing table

NodeId 10233102

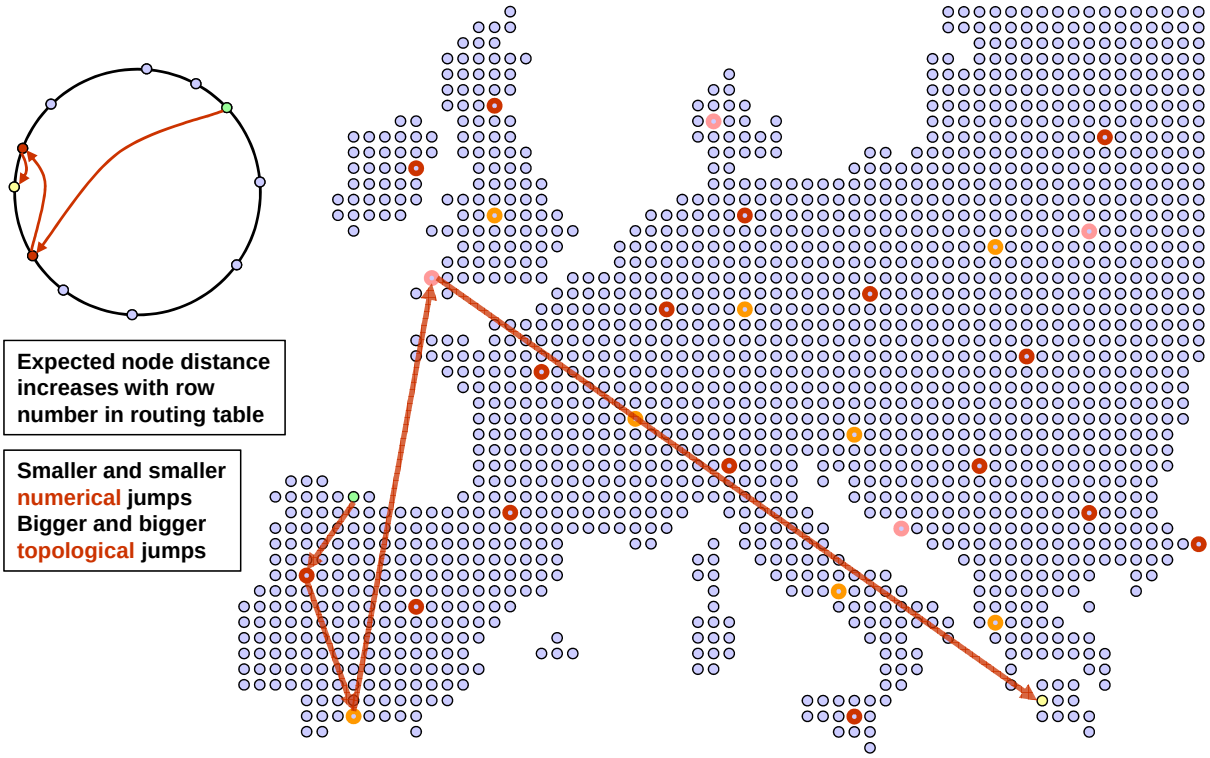
Leaf set	SMALLER	LARGER	
10233033	10233021	10233120	10233122
10233001	10233000	10233230	10233232

Routing table			
-0-2212102	1	-2-2301203	-3-1203203
0	1-1-301233	1-2-230203	1-3-021022
10-0-31203	10-1-32102	2	10-3-23302
102-0-0230	102-1-1302	102-2-2302	3
1023-0-322	1023-1-000	1023-2-121	3
10233-0-01	1	10233-2-32	
0		102331-2-0	
		2	

PASTRY ROUTING TABLE …



PASTRY ROUTING & TOPOLOGY



PASTRY JOIN AND FAILURES

Join

- Uses routing to find numerically closest node already in the network
- Asks state from all nodes on the route and initializes its own state

Error correction

- **Failed leaf node**: contact a leaf node on the side of the failed node and add an appropriate new neighbor
- **Failed table entry**: contact a live entry with same prefix as failed entry until new live entry found,
  if none found, keep trying with longer prefix table entries

MOM MIDDLEWARE

Message Oriented Middleware (MOM)

Data and code distribution via **message exchange** between **logically separated entities**

Typed & **un-typed message exchange** with **ad-hoc tools** both **synchronous** and **asynchronous**

- **wide autonomy** between components
- **asynchronous** and **persistency** actions
- **handler (broker)** with different strategies and QoS
- easy in **multicast, broadcast, publish / subscribe**

Example: Middleware based on messages and queues
MQSeries IBM, **MSMQ Microsoft**, **JMS** SUN

MOM DEPLOYMENT

The **specific deployment** and the **interconnection graph (OR)** is **always static (without the need of a name system)**

Network overlay model between different applications with specific support in distributed environment

- **Necessity of high-level Routing (as in ONs, but static)**
- **Data treatment** while communicating between **different environments**
- **Predefined and static** participating **entities**

Centralized model

MOM with a central node as hub-and-spoke that is responsible of support and pass messages between **different clients**

Distributed model

MOM is located on any client node to form a static ON network, that operate through P2P communication messages between nodes in need of communication

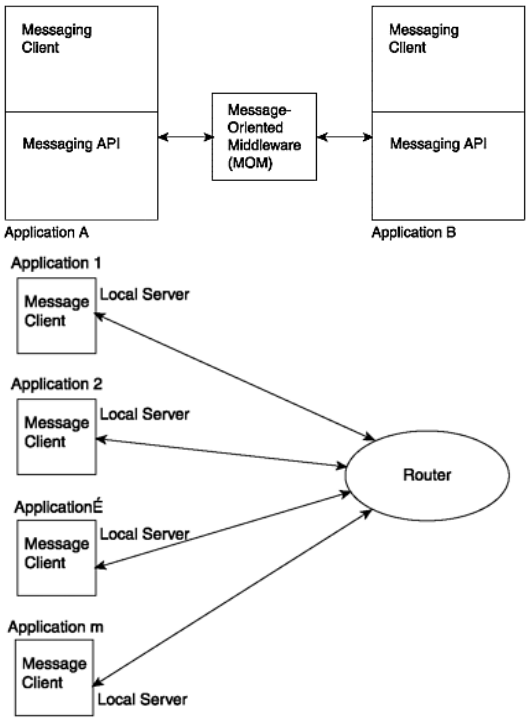
MIDDLEWARE MOM

MOMs provide simple and efficient services

Communication operations available via local ad-hoc API

MOMs put together different nodes and provide services on different **fruition nodes** arranging **queues for the support of every communication**

MOMs as **integrators use of routers, their interconnection and format conversion**



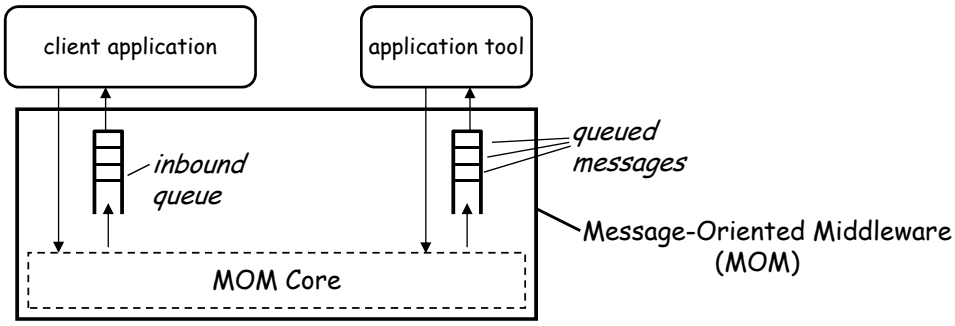
MESSAGE-ORIENTED MIDDLEWARE

MOMs use **queues local to interested nodes**

Inbound and outbound queues on interested **different machines** (connected in an univocal way)

Queue managers guarantee the expected operation level and message forwarding

Routing system to connect different queues
(as an **overlay network** for application level routing)



MIDDLEWARE MOM or GLUE

By following a 'Glue' model

MOMs keep together **different autonomous systems** and organize their specific **interconnection**

Relay are **intermediate** entities that allow the implementation to scale and to organize high level **routing**

Message Broker are entities able to support message content transfer between **environments** with **different representations**

The **MOM** operations use **not** only **asynchronous point-to-point** messages, but also **many-to-many communications**

The **realization cost** must be **limited** and **reduced**:
the main objective is to fast integrate **existent legacy systems**

MOM: MQSeries IBM

MOM proposal very popular and supported

Typically, the **interconnection graph** (**routing**) is controlled by an **always static** and **inflexible** system management **(no name servers and no dynamicity)**

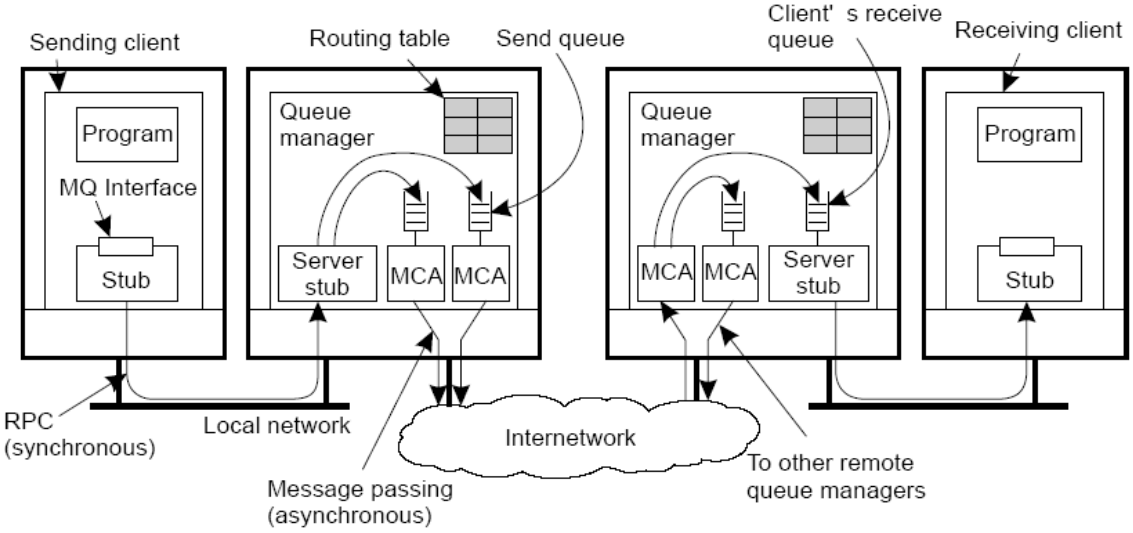
Application level messages are managed by a **queue manager**
Processes interact through *API RPC* to put/extract messages from local queues

Transfers are enabled by unidirectional **channels** managed by **Message Channel Agents** that deal with all details (different delivery politics, message type, etc.)

MCA coordination is offered via **primitives** that should enable coordination (different activation policies, duration, maximum allowed cost, state persistence, etc.)

MQSeries IBM – Websphere part

For the deployment, **the system administrator** defines the **appropriate interconnections** by using **routing tables**, at the **configuration time**



MQSeries IBM: Broker

To achieve the best integration, an **MQ Broker** can operate on the messages by:

- modifying **formats**
- organizing **routing** based on **contained information**;
- working on **application information**, to **specify action sequence**