prof. Marco Avvenuti
Dept. of Information Engineering
University of Pisa
marco.avvenuti@unipi.it

DISTRIBUTED HASH TABLE (DHT)

Outline

- The Publish-Subscribe pattern
- Peer-to-peer architecture fundamentals
- Location and routing: Distributed Hash Table
- Overlay and routing network example: Pastry
- Publish-Subscribe over Pastry: Scribe

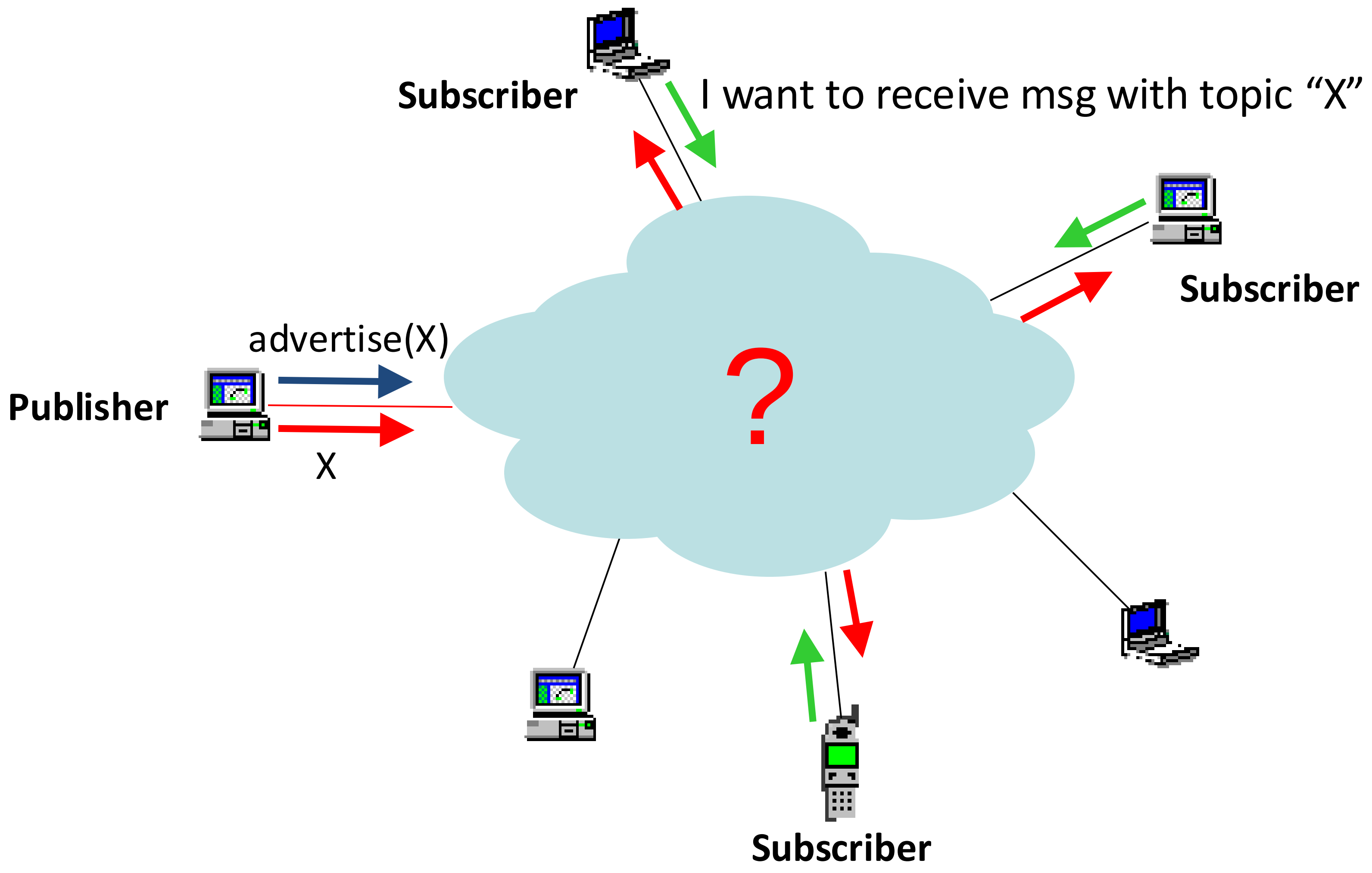
The Publish-Subscribe pattern

- A messaging pattern where senders of messages, called **publishers**, do not program the messages to be sent directly to specific receivers, called **subscribers**, but instead categorize published messages into classes without knowledge of which subscribers, if any, there may be
- Similarly, **subscribers** express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are

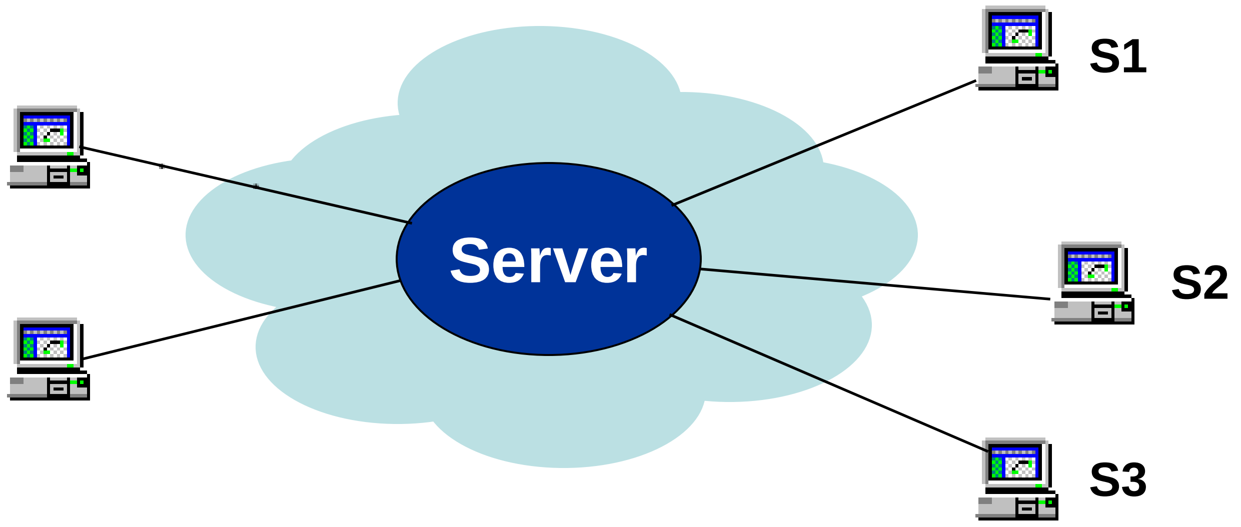
The Publish-Subscribe pattern

- Subscribers typically receive only a subset of the total messages published. The process of selecting messages for reception and processing is called **filtering**:
  - in a **topic-based** system, messages are published to "topics" or *named logical channels*. Subscribers will receive all messages published to the topics to which they subscribe. The publisher is responsible for defining the topics to which subscribers can subscribe
  - in a **content-based** system, messages are only delivered to a subscriber if the attributes or content of those messages matches constraints defined by the subscriber. The subscriber is responsible for classifying the messages

The Publish-Subscribe problem



P-S: the Client-Server Solution



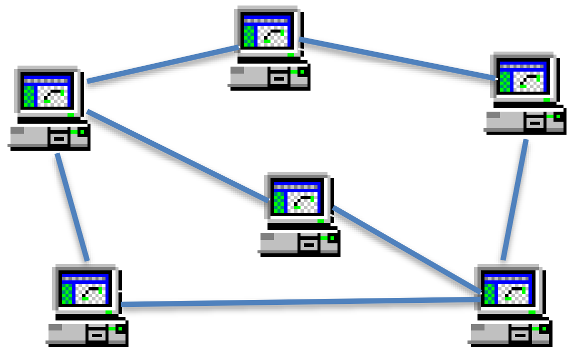
Simple

A lot of traffic and computation in the server

Not scalable

Single Point of Failure

Peer-to-peer Architecture (P2P)

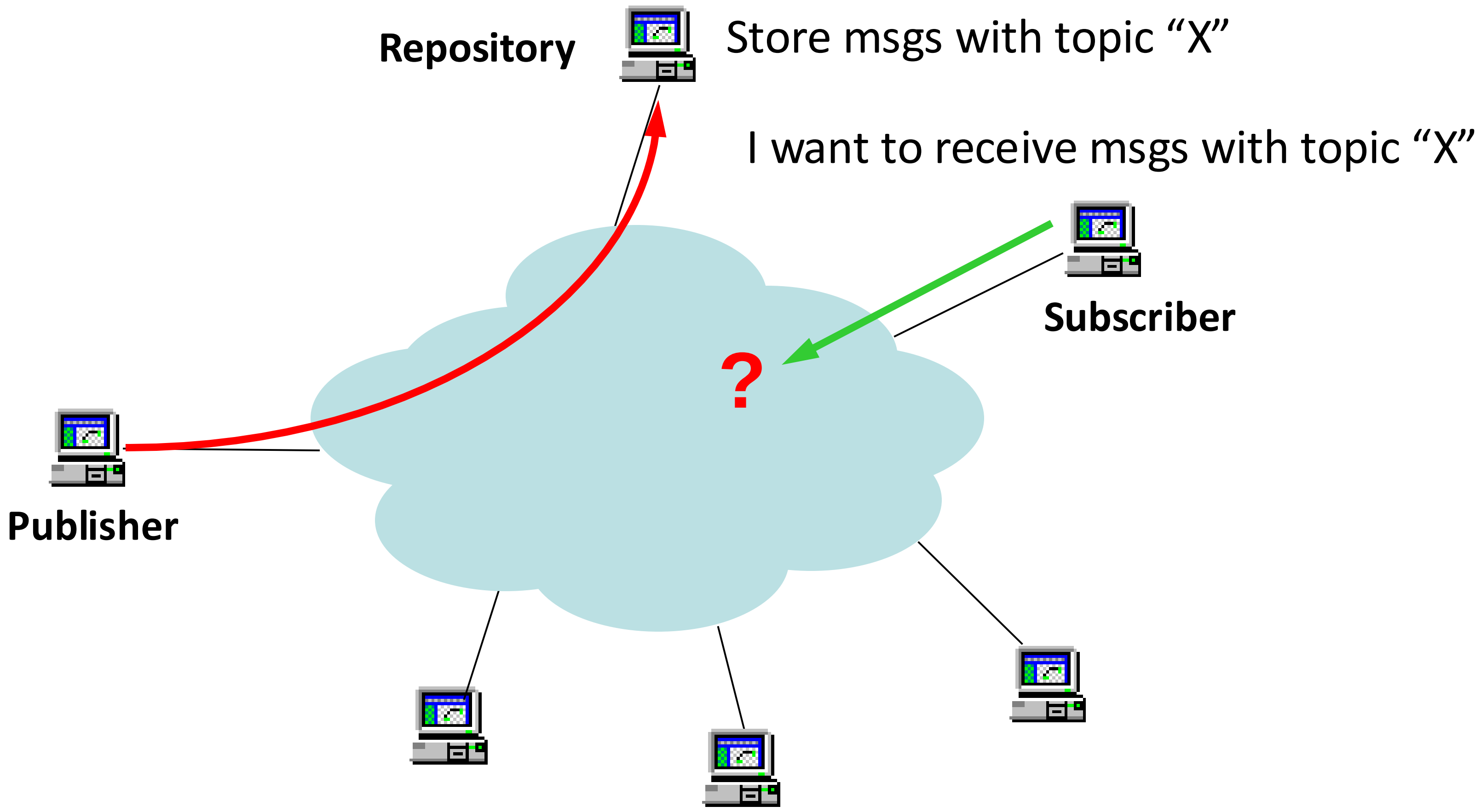


Distributed system architecture:

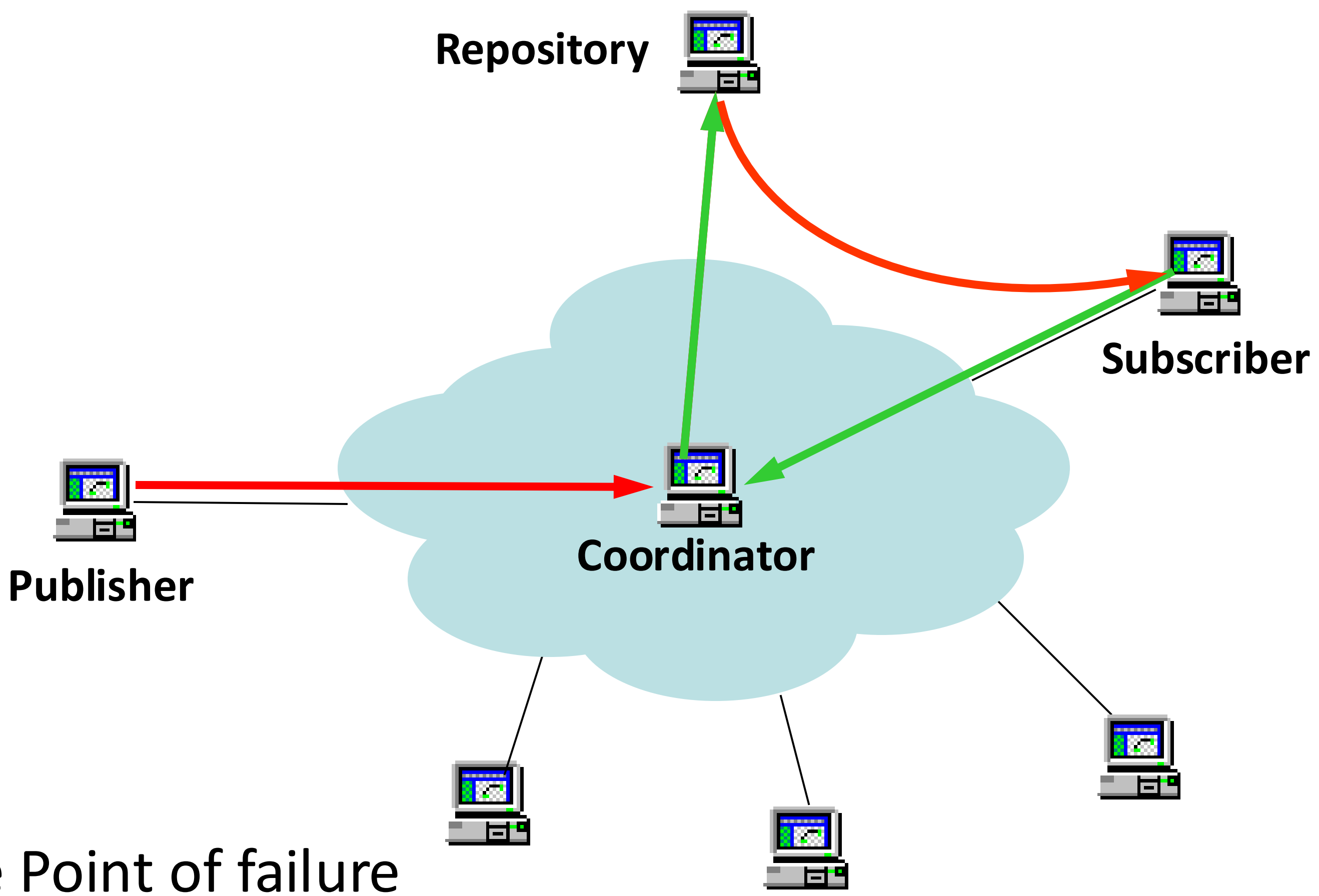
- Nodes are symmetric in function
- No centralized control

Large number of heterogeneous and unreliable nodes

P2P: Lookup Problem


Repository
Store msgs with topic “X”
I want to receive msgs with topic “X”
Subscriber
?
Publisher

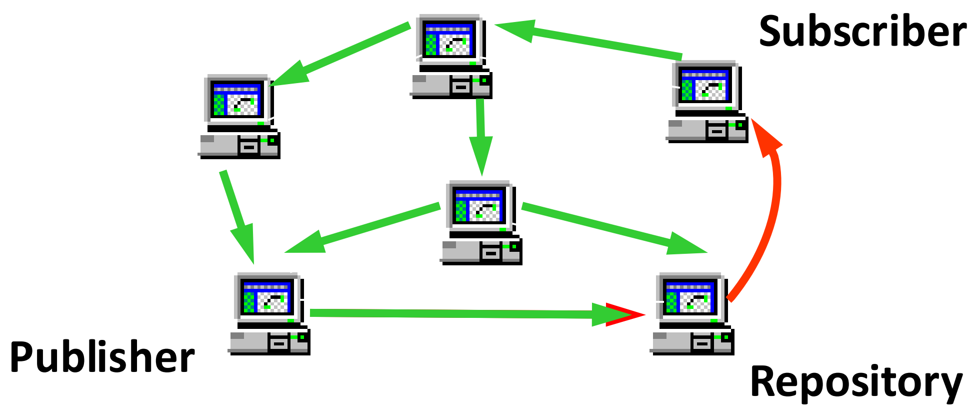
P-S: the Centralized Approach



Single Point of failure

O(N) status in the coordinator node

P-S: the Flooding Approach

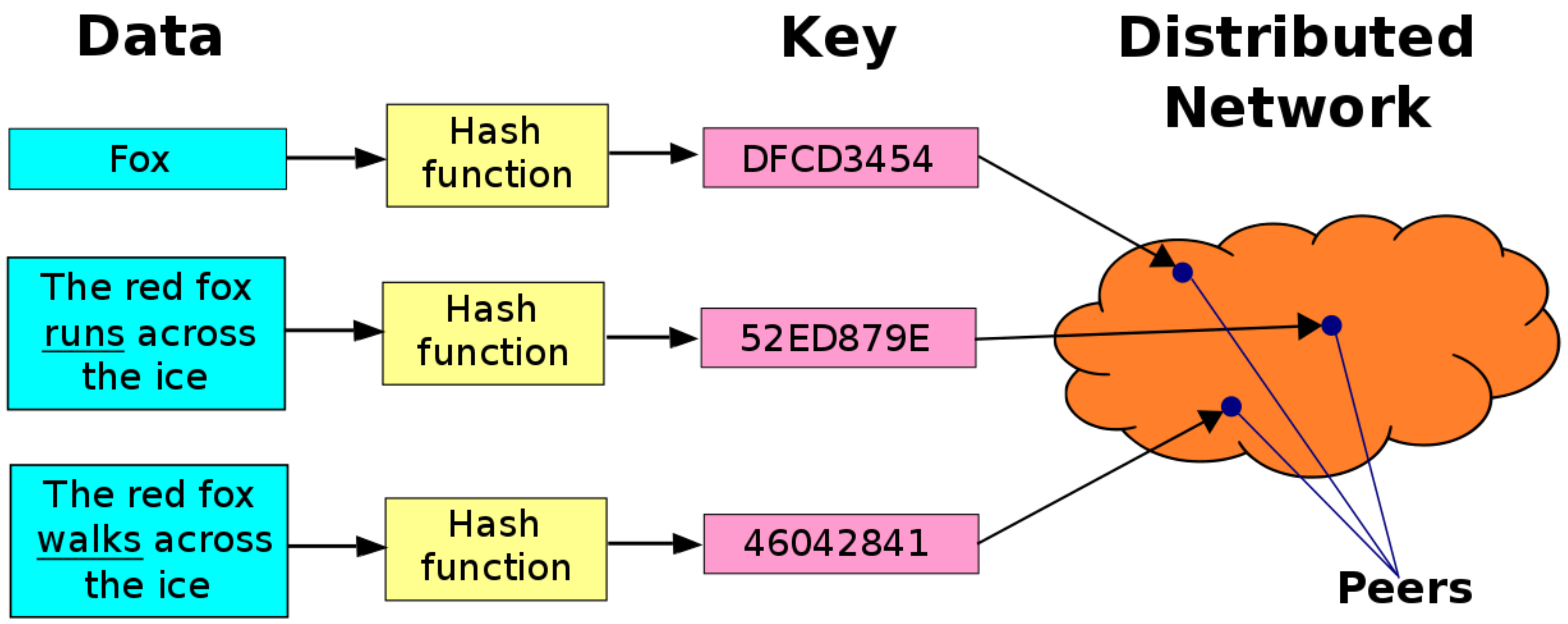


O(N) messages per lookup

DHT definition

- A distributed system that provides a **lookup** service similar to a hash table:
  - (key, value) pairs are stored in a DHT
  - any participating node can efficiently retrieve the value associated with a given key
- Keys are unique identifiers which map to particular values, which in turn can be anything from addresses, to documents, to arbitrary data
- Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption
- The main advantage of a DHT is that nodes can be added/removed with minimum work around re-distributing keys

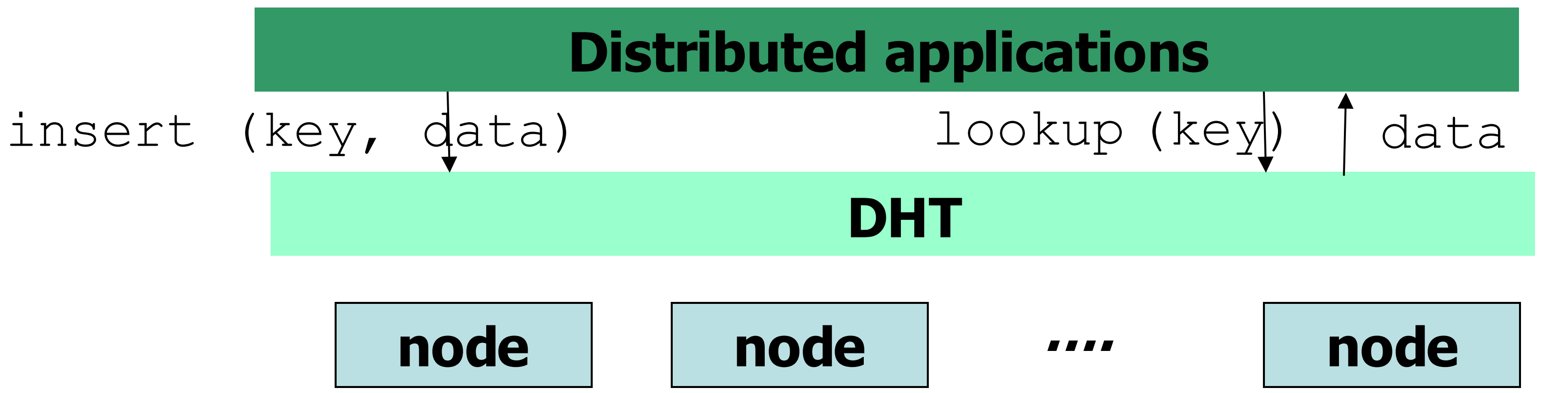
DHT approach



DHT Structure

- An abstract **keyspace**, such as the set of 160-bit strings
- A **keyspace partitioning** scheme splits ownership of this keyspace among the participating nodes
- An **overlay network** then connects the nodes, allowing them to find the owner of any given key in the keyspace

DHT API



Nodes are the hash buckets

Keys identify data uniquely (hash property #1)

DHT balances keys among nodes (hash property #2)

DHT operations

A typical use of the DHT for storage and retrieval might proceed as follows (e.g., to index a file with given filename and data in the DHT):

1. the SHA-1 hash of filename is generated, producing a 160-bit key *k*, and a message `insert(k,data)` is sent to **a** (any) node participating in the DHT
2. the message is forwarded from node to node through the overlay network until it reaches the node responsible for key *k* as specified by the keyspace partitioning. That node then stores the key and the data
3. any other client can then retrieve the contents of the file by again hashing filename to produce *k* and using a message `lookup(k)` to ask **any** DHT node to find the data associated with *k*
4. the message will again be routed through the overlay to the node responsible for *k*, which will reply with the stored data

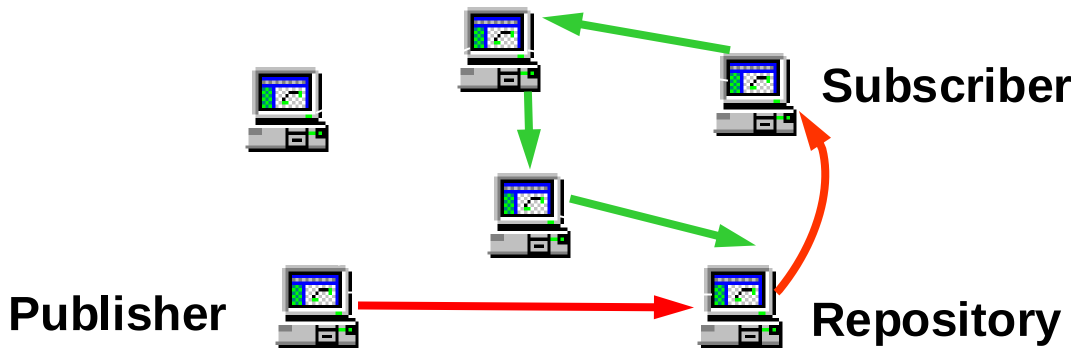
DHT Properties

- *Autonomy and decentralization*: the nodes collectively form the system <u>without any central coordination</u>
- *Scalability*: the system should function efficiently even with thousands or millions of nodes
- *Fault tolerance*: the system should be reliable (in some sense) even with nodes continuously joining, leaving, and failing
- *Load Balancing*: the hash function should <u>evenly distribute</u> keys to nodes
- *Anonimity*: can be achieved by special routing overlay networks that hide the physical location of each node from other participants

DHT application

- A DHT scales to extremely large numbers of nodes and can handle continual node arrivals, departures, and failures
- This allow DHTs form an infrastructure that can be used to build more complex services, such as instant messaging, multicast, peer-to-peer file sharing, content distribution systems
- In the Internet of Things, DHT-based peer-to-peer architectures can be used for Discovery Service and to manage the flow of data from sensors to applications

P-S: the DHT Approach



Usually *O*(log(N)) messages per lookup

Building and maintenance of routing tables is needed

PASTRY

A peer-to-peer overlay network for DHT

Pastry Overview

- The key-value pairs are stored in a redundant peer-to-peer network of connected Internet hosts
- Reduce the overall cost of routing a packet from one node to another by avoiding the need to flood packets
- The protocol is bootstrapped by supplying it with the IP address of a peer already in the network and from then on via the routing table which is dynamically built and repaired
- Because of its redundant and decentralized nature, there is no single point of failure and any single node can leave the network at any time without warning and with little or no data loss

Pastry Design

- The topology of the hash table's key-space is **circular**
- NodeIDs and Keys are 128-bit unsigned integer (in base 2^b) representing position in the circular space
- NodeIDs are chosen randomly and uniformly, *so peers who are adjacent in nodeID are geographically diverse (i.e., distant)*
- Unique nodeIDs are assigned when nodes join the system, based on a **secure hash** of the node’s IP address
- Keys are stored in the node with nodeID **numerically closest** to the key, among all live Pastry nodes

Pastry Routing

- Given a message and a key, Pastry reliably routes the message to the node with the nodeID that is numerically closest to the key
- Assuming a network consisting of *N* nodes, Pastry can route to any node in less than $\lceil log_{2^b}\ N \rceil$ steps *on average* (*b* = 4, typically)
- With concurrent node failures, eventual delivery is guaranteed, unless *l/2* or more nodes with adjacent nodeIDs fail simultaneously (*l* is an even integer parameter with typical value 16)

Prefix Routing

1. NodeIDs and keys are thought of as a sequence of digits with base 2^b
2. A node's **routing table** is organized into $\left\lceil log_{2^b}\ N \right\rceil$ rows and 2^b columns
3. The entries in row *n* each refer to a node whose nodeID matches the present node's nodeID in the first *n* digits, but whose *n+1*th digit has one of the $2^b - 1$ possible values other than the *n+1*th digit in the present node's ID
4. Each entry in the routing table refers to one of potentially many nodes whose nodeID have the appropriate prefix
5. Among such nodes, the one closest to the present node (according to a scalar proximity metric, such as the round trip time) is chosen

Routing Table of a Pastry node with:

nodeID = 65a1*x*

b = 4

- digits are in base 16
- *x* represents an arbitrary suffix
- the IP address associated with each entry is not shown

0*x*	1*x*	2*x*	3*x*	4*x*	5*x*		7*x*	8*x*	9*x*	a*x*	b*x*	c*x*	d*x*	e*x*	f*x*
60*x*	61*x*	62*x*	63*x*	64*x*		66*x*	67*x*	68*x*	69*x*	6a*x*	6b*x*	6c*x*	6d*x*	6e*x*	6f*x*
650*x*	651*x*	652*x*	653*x*	654*x*	655*x*	656*x*	657*x*	658*x*	659*x*		65b*x*	65c*x*	65d*x*	65e*x*	65f*x*
65a0*x*		65a2*x*	65a3*x*	65a4*x*	65a5*x*	65a6*x*	65a7*x*	65a8*x*	65a9*x*	65aa*x*	65ab*x*	65ac*x*	65ad*x*	65ae*x*	65af*x*

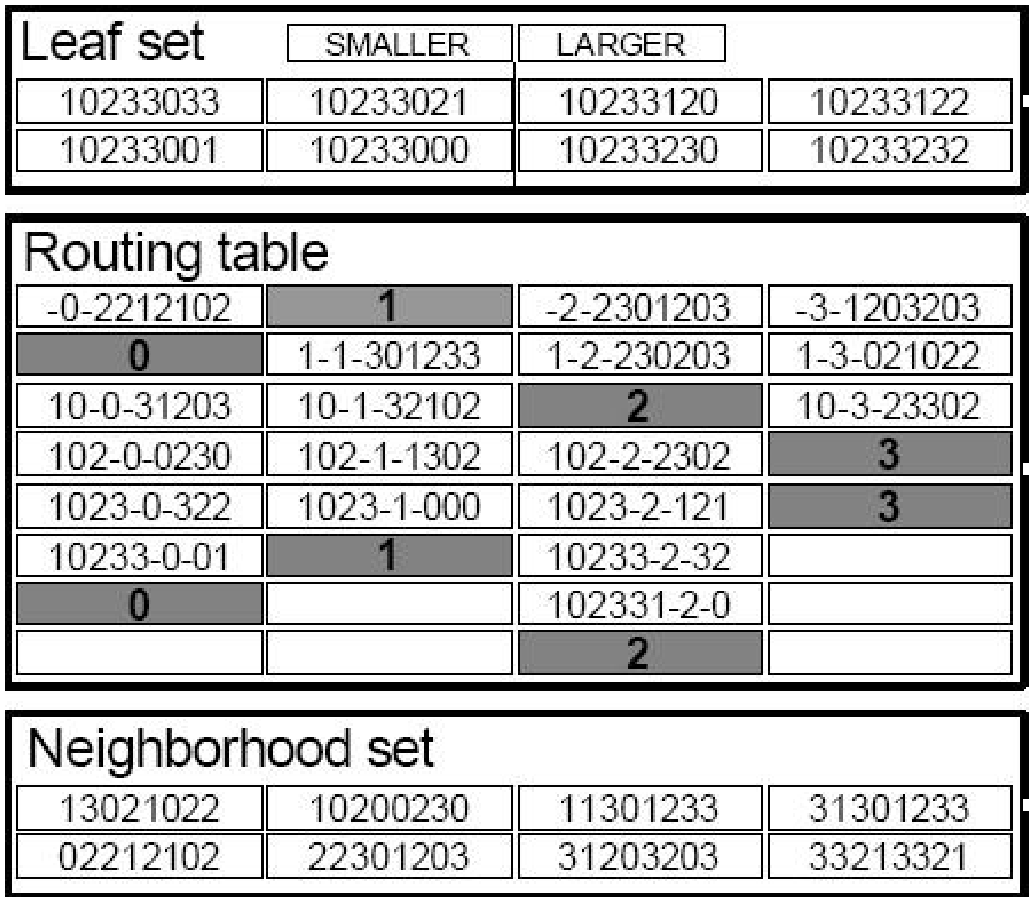
Leaf set

In addition to the routing table, each node maintains IP addresses for the nodes in its leaf set, i.e., the set of:

- *l/2* nodes with numerically closest larger nodeIDs
- *l/2* nodes with numerically closest smaller nodeIDs

relative to the present node’s nodeID

Example



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	

Neighborhood set

13021022	10200230	11301233	31301233
02212102	22301203	31203203	33213321

Nodes numerically closer to the present Node

Prefix-based routing entries: common prefix with 10233102-next digit-rest of nodeID

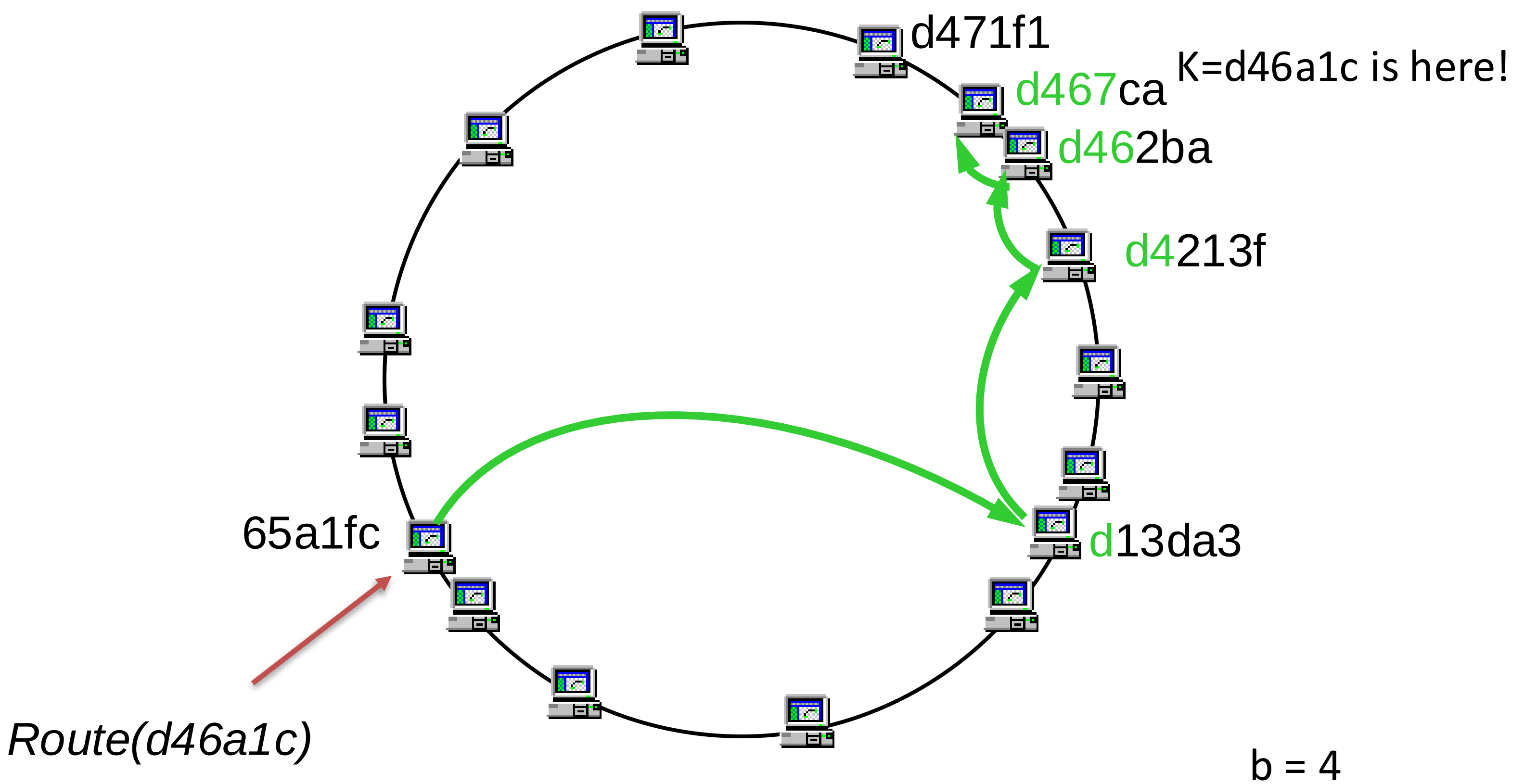
Nodes “physically” closest to the present node

nodeId=10233102, b = 2, l = 8

Routing step

1. Seek the routing table:
   - forward the message to a node whose nodeID shares with the key a prefix that is at least one digit longer than the prefix that the key shares with the current nodeID
2. If no such node is found in the routing table, check the leaf set:
   - forward the message to a node whose nodeID shares a prefix with the key as long as the current node, but is numerically closer to the key than the current nodeID
3. If such node does not exist in the leaf set, the key is stored in the current node (unless the l/2 adjacent nodes in the leaf set have failed concurrently)

Routing Example



b = 4

N = $2^{24} = 16^6$

Locality

- Describe properties of Pastry’s routes with respect to the **proximity metric**
- The proximity metric is a scalar value that reflects the “distance” between any pair of nodes. For example: <u>the round trip time</u>
- It is assumed that a function exists that allows each Pastry node to determine the “distance” between itself and a node with a given IP address

Short routes property

The total distance, in terms of the proximity metric, that messages travel along Pastry routes:

1. each entry in the node routing tables is chosen to refer to the nearest node, according to the proximity metric, with the appropriate nodeID prefix
2. as a result, in each step a message is routed to the nearest node with a longer prefix match

Simulations show that the average distance traveled by a message is between 1.59 and 2.2 times the distance between the source and destination in the underlying Internet

Route convergence property

The distance traveled by two messages sent to the same key before their routes converge

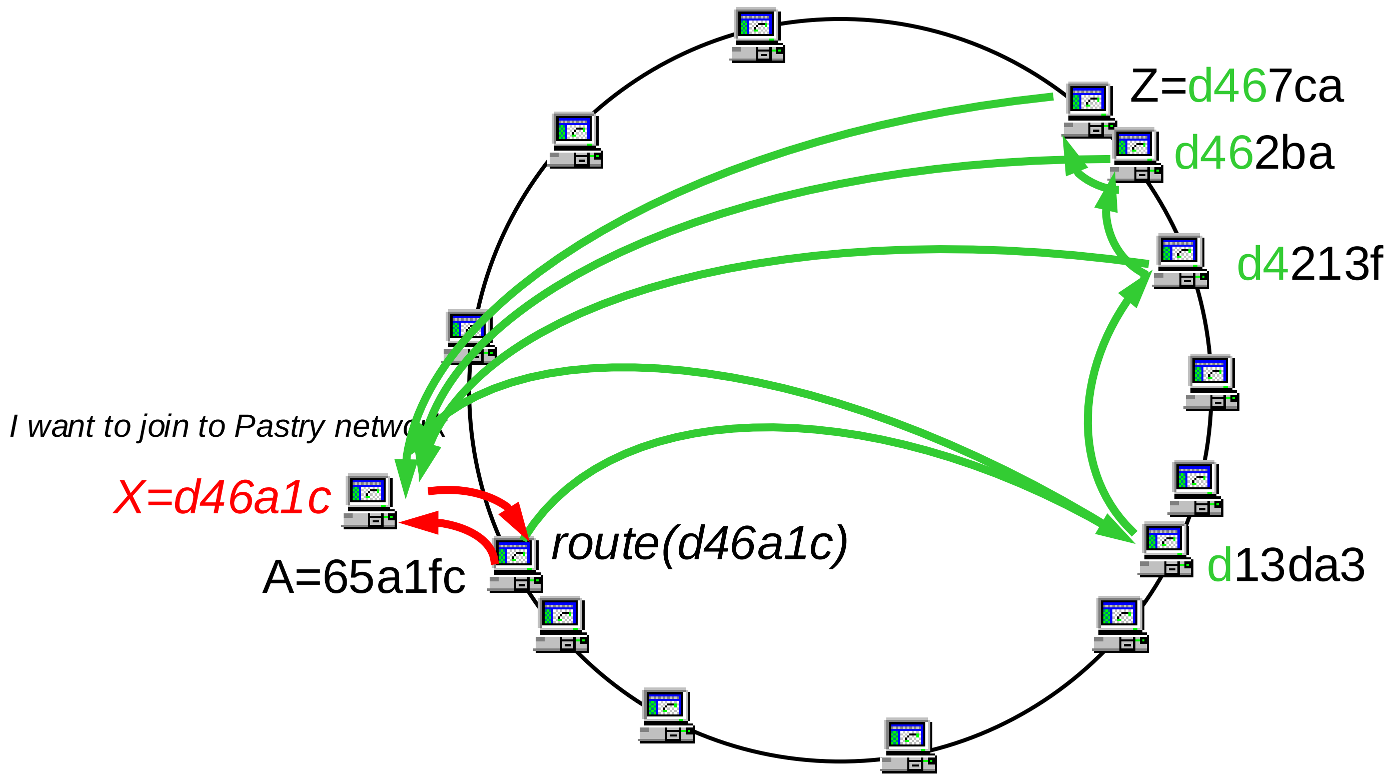
- Simulations show that, given our network topology model, the average distance traveled by each of the two messages before their routes converge is approximately equal to the distance between their respective source nodes

Node Arrival

An arriving node with nodeID = *X*

1. contacts a **known** nearby Pastry node *A* (according to the proximity metric) and asks to route a “join” message using *X* as the key
2. this message is routed to the existing node *Z* with nodeID numerically closest to *X*
3. *X* then obtains the leaf set from Z, and the *i-th* row of the routing table from the *i-th* node encountered along the route from *A* to *Z*
4. *X* notifies nodes to update their states

Node Arrival Example



Node Departure or Failure

- Neighboring nodes in the nodeID space (which are aware of each other by virtue of being in each other's leaf set) periodically exchange keep-alive messages
- If a node is unresponsive for a period T, it is presumed failed
- All members of the failed node's leaf set are then notified and they update their leaf sets
- Since the leaf sets of nodes with adjacent nodeIds overlap, this update is trivial

Node Departure or Failure

- A recovering node contacts the nodes in its last known leaf set, obtains their current leaf sets, updates its own leaf set and then notifies the members of its new leaf set of its presence
- Routing table entries that refer to failed nodes are repaired lazily

Pastry API

nodeId = pastryInit(Credentials)

a. causes the local node to join an existing Pastry network (or start a new one) and initialize all relevant state

b. returns the local node's nodeID

The credentials are provided by the application and contain information needed to authenticate the local node and to securely join the Pastry network

route(msg,key)

- causes Pastry to route the given message to the node with nodeID numerically closest to key, among all live Pastry nodes

send(msg,IP-addr)

- causes Pastry to send the given message to the node with the specified IP address, if that node is alive. The message is received by that node through the deliver method

Pastry API

Applications layered on top of Pastry must export the following operations:

deliver(msg,key)

- called by Pastry when a message is received and the local node's nodeID is numerically closest to key among all live nodes, or when a message is received that was transmitted via *send*, using the IP address of the local node

forward(msg,key,nextId)

- called by Pastry just before a message is forwarded to the node with nodeId = nextId. The application may change the contents of the message or the value of nextId. Setting the nextId to NULL will terminate the message at the local node

newLeafs(leafSet)

- called by Pastry whenever there is a change in the leaf set. This provides the application with an opportunity to adjust application-specific invariants based on the leaf set

Scribe

A publish-subscribe system built on top of Pastry

Scribe

- Scalable application-level multicast infrastructure
- Any Scribe node may create a *group*
- Other nodes can then join the group, or multicast messages to all members of the group
- Best-effort dissemination of messages, no particular delivery order

Scribe

- Uses a peer-to-peer network of Pastry nodes to manage group creation, group joining and to build a per-group multicast tree used to disseminate the messages multicast in the group
- Fully decentralized: all decisions are based on local information, and each node has identical capabilities
- Can support simultaneously a large numbers of groups with a wide range of group sizes, and a high rate of membership turnover

Scribe API

- **create(credentials, groupId)**
  creates a group with groupId. Throughout, the credentials are used for access control
- **join(credentials, groupId, messageHandler)**
  causes the local node to join the group with groupId. All subsequently received multicast messages for that group are passed to the specified message handler
- **leave(credentials, groupId)**
  causes the local node to leave the group with groupId
- **multicast(credentials, groupId, message)**
  causes the message to be multicast within the group with groupId

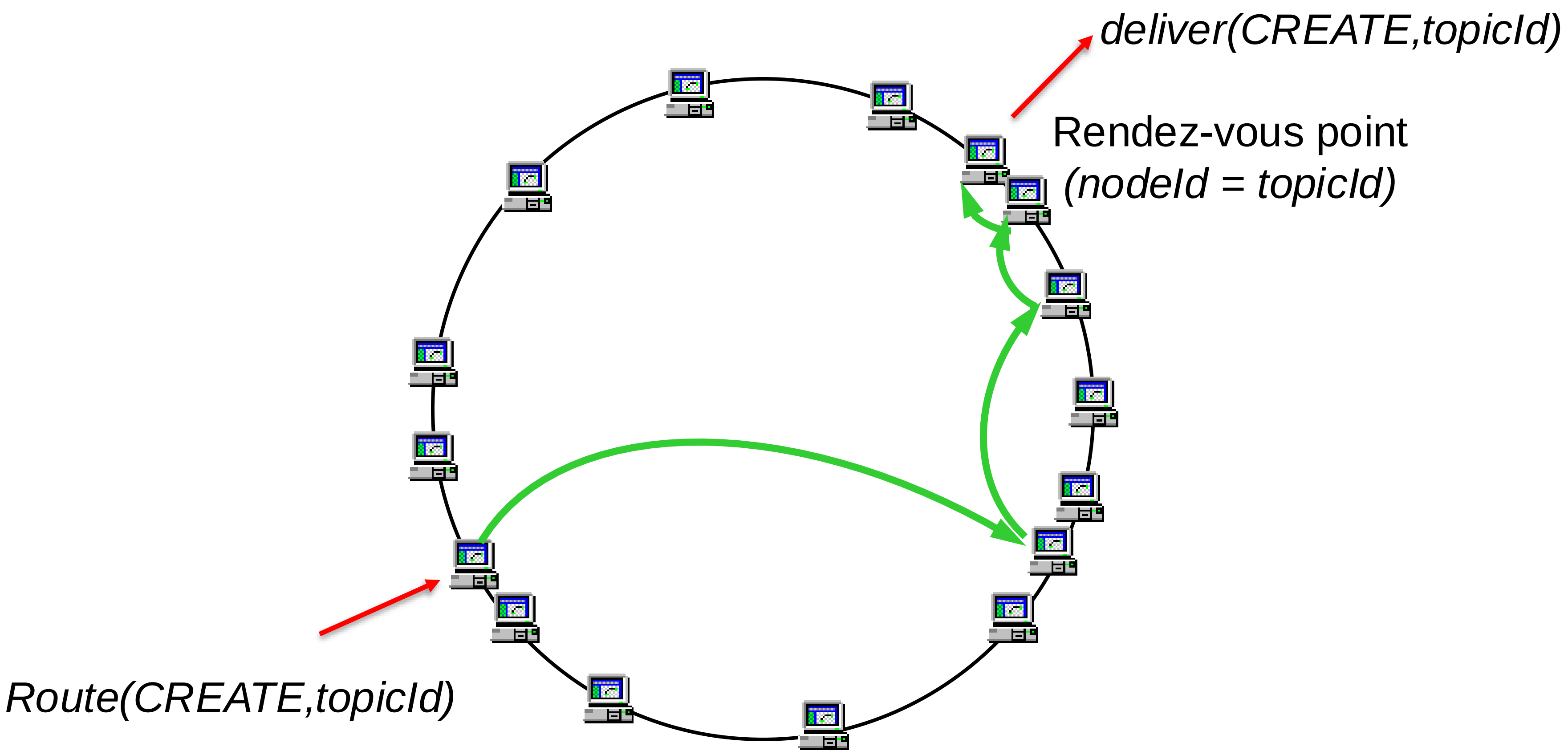
Scribe P-S Implementation

- Each topic (group) has a unique **topicId**
- Rendezvous point
  - the Scribe node with a *nodeId* numerically closest to the *topicId*
  - root of multicast tree created for the group
- The Scribe software on each node provides the *forward* and *deliver* methods, which are invoked by Pastry whenever a Scribe message arrives

Scribe P-S Implementation

- The topicId is the hash of the topic's textual name concatenated with its creator's name
- The hash is computed using a collision resistant hash function (e.g. SHA-1), which ensures a uniform distribution of topicIds
- Since Pastry nodeIDs are also uniformly distributed, this ensures an even distribution of groups across Pastry nodes

Create Topic

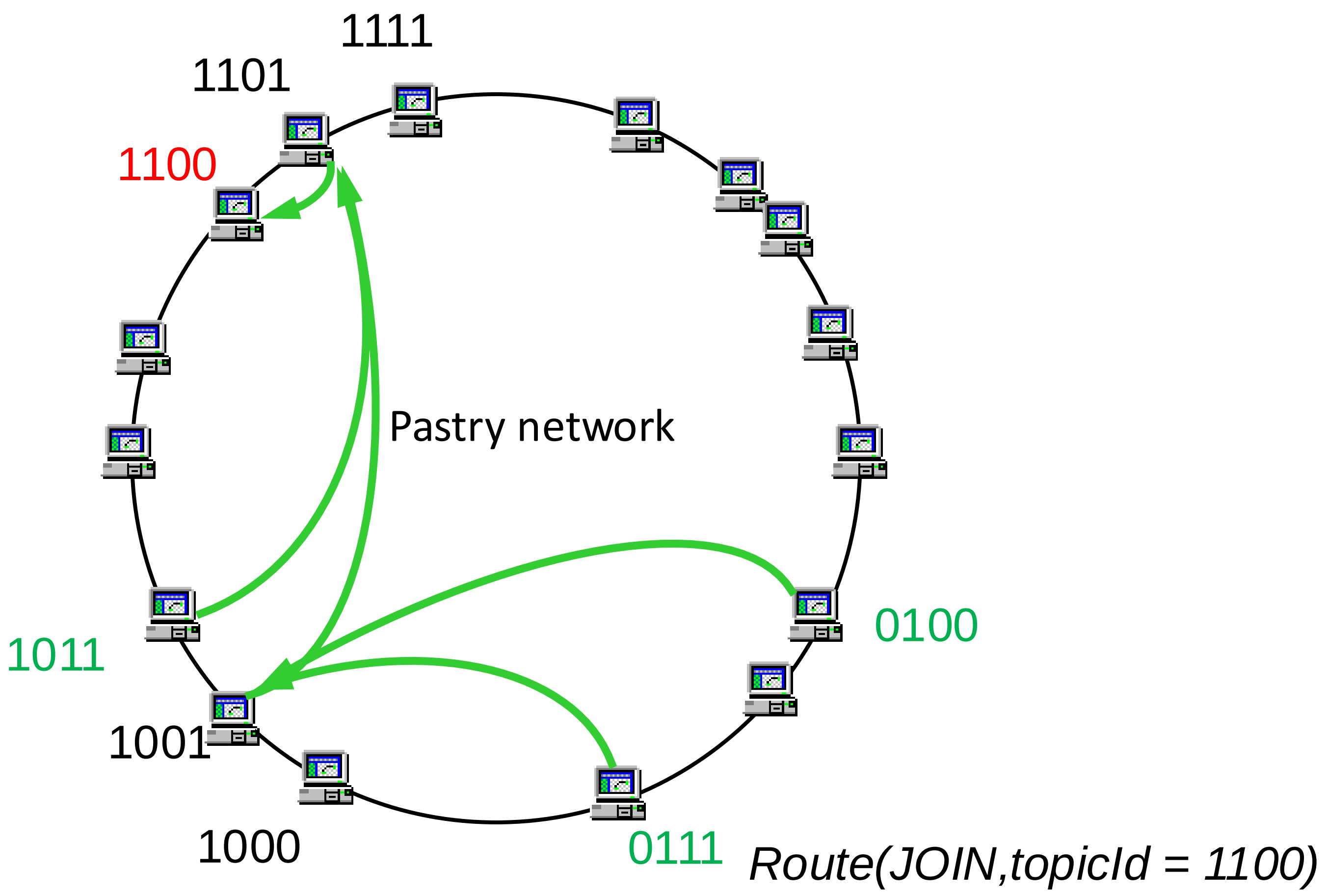


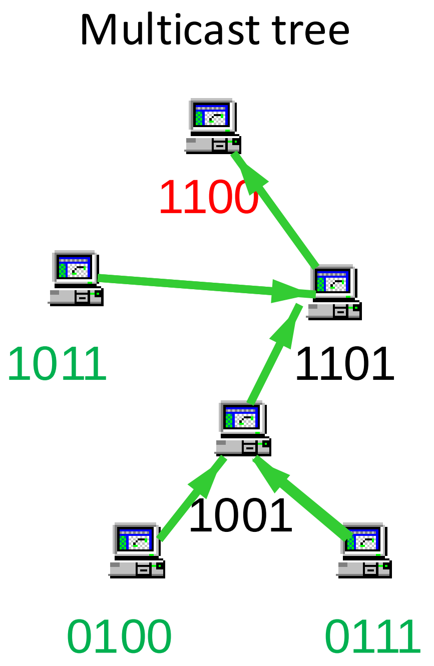
create(credentials,topicId)

Scribe P-S Implementation

- Scribe creates a multicast tree, rooted at the rendez-vous point, to disseminate the multicast messages in the group.
- The multicast tree is created using a scheme similar to reverse path forwarding.
- The tree is formed by joining the Pastry routes from each group member to the rendez-vous point.

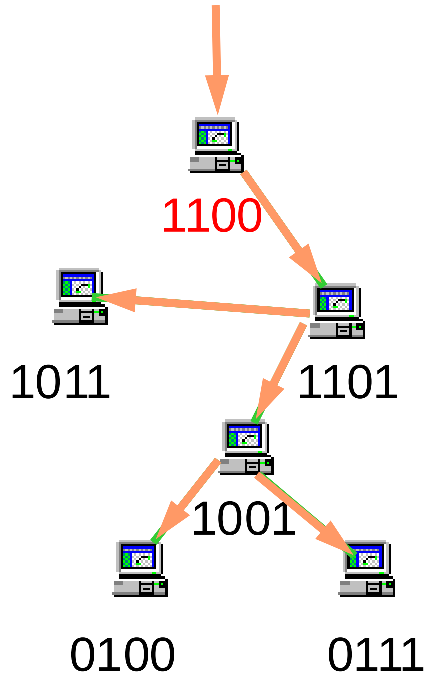
Subscription



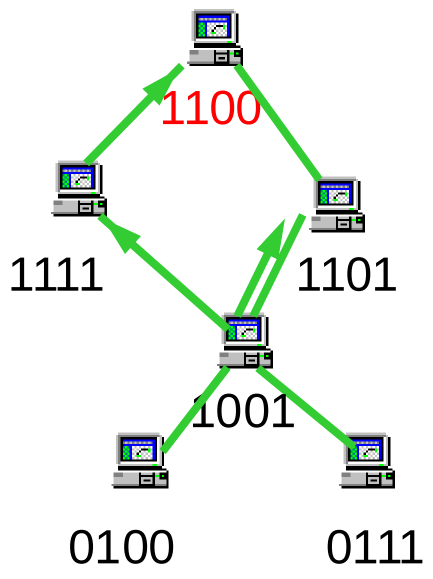


Message Dissemination

- Route through the Pastry network using the topicId as the destination
- Dissemination along the multicast tree starting from the rendez-vous



Repairing the multicast tree



References

Distributed Hash Tables (DHTs): Design and Performance. Brad Karp. (ppt file)

A Survey on Distributed Hash Tables. Nitin Gupta. (html page)

Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. Antony Rowstron and Peter Druschel. (pdf file)

Scribe: A large-scale and decentralized application-level multicast infrastructure. Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. (pdf file)