BLOOM FILTERS:

- short binary array with of length m.
- k different 'collision-resistant' hash functions
- n number of elements mapped into the bloom filter

1) Operations:

- add element: hash the element with each hash function and place a 1 in the position indicated by the result
- membership: hash the element with each hash function and check that for each position there is a 1. If there is a 1 in each position then we say the element belongs to the set. If there is a zero in any of the positions then the element is not part of the set.

2) Benefits:
- time to add/check for element is indipendent of number of elements added (costant O(k)).
- computation can be easily parallelized between the hash functions
- space efficient --> with an optimal k we mantain a 1% false positive rate with 9.6 bits per element. Adding 4.8 bits to each element makes this rate drop about ten times.

3) False positive Probability (taken from wikipedia...):

question: how do we decrease the number of false positives?
answer: increase m, decrease n (doesn't really make much sense), make k optimal (not just increase!)

question: why does increasing k not always decrease the false positive rate?
answer: by increasing k we also increase the number of bits set per element. For some values of k this decreases the number of false posivites, after its optimal value the number of bits set starts to be a disadvantage. In general we want to keep half of our bloom filter empty (maybe need to investigate a bit more on this)

Now let's look at this a bit more mathematically:
- we assume a hash function chooses a position in the bloom filter with equal probability. The probability that a certain bit is NOT set to one by a hash function is: 1 - (1/m)
- the probability that a bit is still zero after k different hash functions is: (1 - 1/m)^k and after n elements have been inserted is (1 - 1/m)^nk.
- the probability that a bit is one after n inserts is 1 - (1- 1/m) ^nk.

- we have a false positive when the membership test returns 1s for all the hash functions even though the element does not belong to the set. The probability of returning all 1s from the hash function is: [1 - (1- 1/m)^nk]^k ~ (1 - e^(-kn/m))^k

- For a given m and n the value that minimizes k is ln2(m/n) ~ 0.7(m/n). With this value of k we have a false positive probability rate of 2^-k.

- given k and fixing a false positive probability p the relationship between m and n is m = -nln(p)/[(ln2)^2]. It is important to notive that m grows linearly with the number of items n.

URL SHORTENING:

question: how does it work?
answer:
- There are two main techniques to shrink long urls into short urls: hash functions and keeping a global 'counter'. The third option is to have the user choose an 'alias' for their url
- whenever a link to a short url is posted the user contacts the service (e.g. tinyurl.com or bit.ly) and the service redirects the user to the original url.

Purpose:
- space constraints on some services (Twitter and SMS)
- disguise underlying address --> privacy

Problems/ Criticism:
- users abused the system (redirects to unexpected websites that hosted inappropriate content / unpleasant aliases chosen for respectuful websites)
- linkrot: if the service goes down all the redirections are ineffective

- many url shortening services use foreign domain extensions (bit.ly) and therefore fall under the jurisdiction of that country . Nevertheless, most of the servers are NOT hosted within the actual country (now that Lybia is at war, bit.ly's CEO assures this will not slow the service down).

BUILDING A SCALABLE URL-SHORTENING SERVICE:

1) One server:

  - global counter
  - hash function: 6 characters long hash, each character can have 62

possible values (26 up+ 26 lw + 10 numbers) --> 62^6 combinations ~ 2^36. Birthday paradox: first collision at sqrt(2^32) ~ 2^ 18 = 210000 (too easy to have collisions)

Our server will get inundated by requests, we need to expand/ get a better server

2) One server, multicore / many servers (I think this scenario is very similar to having different servers?)

- global counter: problem is lock per counter -- contention between cores/servers to obtain lock on counter.
- hash function: we have to partition the address space to avoid conflicts between cores.

This is a partitioning problem. Possible solutions:

- Each core/server gets a lock on a range of numbers (say 100-1000) then locally gives them out (global counter scenario)
- Each server prepends his own server/core ID before the hash/ counter. ID: <server><core><hash_value>

Architecture:

- Two tiers, one is the front-end which assigns/request URL-tiny urls, the other is the storage tier. For every assignment we need to write to storage, for every request we need to read from storage.

question: how do we make the storage system scalable?

- we can partition the data blocks based on their initial character (or 2 characters -- 62^2) and each partition is owned by one server via consistent hashing (to allow for more nodes to join in/crash)

Fault tolerance:

- master db node is 1 server. Each master db node has secondary db nodes --> k more servers. Whenever there is a write to a master DB the master writes to its secondary DBs in case it fails.

SCRIBE:
publish subscribe system: you have topics that are created and all the objects can be published under a different topic and anybody who has subscribed to that topic has to get that object. (who creates the data: that is up to the system, in pastry it is only the creator of the topic)

Last system we are going to talk about is a system that combines both multicast tree and distributed hash table.

in a p2p system you do not want to fan out to everyone (flooding), you want to construct a tree-like structure (cycle free) to fan out the data in a distributed way.

joining the tree: you don't want to attach directly to the root of the tree you want to attach to an intermediate node or at least to any already subscribed node with the same probability to get a uniform load

how do you use a dht to help build the multi cast tree??

- the dht is composed of nodes that share the key space. When a topic is created the hash of the topic name is the key to that topic. That key is assigned to the appropriate node. That node becomes the root of the multicast tree for that topic.

- when another node wants to join the multicast tree, it looks up the key and it starts rounting the subscription message towards the root.

- Whenever this subscription message hits (is seen by) an already subscribed node, that node will the the new subscriber to become its child in the multicast tree and will store the name of that child for that topic.

- to keep the structure consistent we have keep alive messages. When a node goes missing (crashes?) its children simply re subscribe to the topic (hence find a new parent).

MORE ON SCRIBE
==========

SUMMARY: SCALABLE APP-LEVEL MULTICAST

- used to create and manage groups
- build efficient multicast trees for dissemination of messages to each group
- balances load on nodes while achieving acceptable delay and link stress

- built on Pastry
  - leverages robustness, self-org., localilty, reliability

- Pastry can route to any node in less than CEIL (log_{2^b} N)
  - b is typically 4

- With concurrent node failures, eventual delivery is guaranteed
  unless l/2 or more nodes with adjacent nodeIds fail simultaneously
  - l is typically 16

REQUIREMENTS

- 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 delivery

- Simple API to its applications

  create(groupId)   - create a group with groupId
  join(groupId, messageHandler) - join group; use call-back for recv
messages
  leave(groupId)   - leave the group groupId
  multicast(groupId, message)   - message must be multicast to within
group

NOTE:

- fully decentralized
- each node can act as
  - ROOT
  - MULTICAST SOURCE
  - GROUP MEMBER
  - NODE WITHIN MULTICAST TREE -> forwarder

Node:

forward() : message is routed through a node
deliver() : when message arrives at node with nodeId numerically
          closest to message's key
          - or message addressed to local node

invoked by Chord/Pastry whenever a message arrives

Possible message types:

- JOIN
- CREATE
- LEAVE
- MULTICAST

groups -> set of groups that the local node is aware of
msg.src -> nodeId of the message's src node
msg.group -> groupId of the group

msg.type -> message type

route (msg, key) -> route given message to the node with nodeId closest to key
send (msg, IP-addr) -> send directly to IP address

RENDEZVOUS POINT:
- The scribe node with nodeId numerically closest to the groupId acts as the rendezvous point for the group

- The rendezvous is the root of the multicast tree

GROUPID

- The groupId is the collision-resistant

  SHA-1 (group's textual name || creator name)

  uniform distribution of the groupIds

- Since Pastry nodeIds are also uniformly distributed this ensures an even distribution of groupIds

- Alternatively

  - we can make creator of group be rendezvous point for group

  nodeId can be hash (textual name of the node)
  groupId can be hash (nodeId of creator || textual name of the group)

MULTICAST TREE

- Scribe creates a multicast tree rooted at the rendezvous point

  - tree formed by joining Pastry routes from each group member to the rendezvous point

- Scribe nodess that are part of a group's multicast tree are called forwarders with respect to the group. They may or may not be members of the group.

- Each forwarder maintains a children table for the group containing

  IP address, nodeId

  for each of its children in the multicast tree

CREATE A GROUP:

- To create a group, a Scribe node asks Pastry to route a CREATE message using the groupId as the key.

```
  route (CREATE, groupId)
```

- Pastry delivers this message to the node wit the nodeId numerically close to groupId. The Scribe deliver method adds the group to the list of groups it already knows about.

- The scribe node becomes the rendezvous point (ROOT)

```
  forward (msg, key, nextId)
       switch (msg.type) is
            JOIN:          if !(msg.group E groups)
                             groups = groups U msg.groups
                             route (msg, msg.group)
                           groups[msg.group].children U msg.source
                           nextId = null // stop routing


  deliver (msg, key)
       switch (msg.type) is
            CREATE:       groups = groups U msg.group
            JOIN:         groups [msg.group].children U msg.source
            MULTICAST:    forall node in groups [msg.group].children
                             send (msg, node)
                          if (memberOf (msg.group))
                            invokeMessageHandler (msg.group, msg)
            LEAVE:
                          groups[msg.group].children =
                                groups[msg.groups].children - msg.source
                          if (NUM(groups[msg.group].children) == 0)
                             send (msg, groups[msg.group].parent)
```

FORWARDER

- may or may not be a member of the group
- each forwarder maintains a children table for the group containing
  - IP address, nodeId for each of its children in the multicast tree

JOIN A GROUP:

```
  route (JOIN, groupId)
```

- When a scribe node wishes to join a group it asks Pastry to route a JOIN message with the groups groupId as the key
  - JOIN.groupID

- This message is routed towardes the group's rendezvous point

- checks its list of groups to see if it is currently a forwarder; if so, accepts the node as a child (add to children table)

- if it i not a forwarder, creates an entry for group, adds src node as child in the associated children table; passes on the JOIN to become a forwarder

- forward checks its list of groups to see if it is currently a forwarder

LEAVE A GROUP:

send (LEAVE, group.parent)

It records locally that it left the group

- if no other entries in children table, it sends a LEAVE message to its parent in the multicast tree

1. CAN THERE BE ANY LOOPS ?
2. Does the rendezvous point handle all JOIN requests ?
3. Is the tree balanced ? (Randomization properties)
   - forwarding load is evenly distributed

MULTICAST

route (MULTICAST, groupId)

1. and ask for it's IP address
2. Cache IP addresses, and use it in subsequent multicasts to avoid repeated routign through Pastry

FAILURE DETECTION

- non-leaf node can send a heartbeat message to its children
- if a child suspects parent is fault (does not receive hearbeat)
  - calls route (JOIN, groupId) again
- will route to a new parent, thus repairing the multicast tree

WHAT ABOUT RENDEZVOUS POINT FAILURE ?

- again, children can detect and one of k closest nodes can take over

SOFT STATE IN CHILDREN TABLE ENTRIES

- periodically discarded unless refresh message from child

Fault detection and recovery is LOCAL. Only a small number of nodes are involved (O(log N)).