Exercise 2

Monday 23.5 2011
Hand-in: Thursday 30.5. 2011 in lecture or per mail
Exercise: to be announced

Exercises Peer-to-Peer-Systems and Security (SS2011)

Dr. Heiko Niedermayer
Lehrstuhl für Netzarchitekturen und Netzdienste
Technische Universität München

Rules: There will be five exercise sheets. You have to hand-in 70 % of the assignments, attend atleast 3 exercise courses and present a solution in the exercise course to get the 0.3 bonus..

Task 1 CoolSpots Munich III
This time you should solve the task to organize the CoolSpots Munich network with a Distributed Hash Table. Use spot_ID = h(GPS coordinate of spot) to store the items. As item descriptions are rather short, the data is stored on the DHT nodes and not only on the node that contributed the item.

a) Briefly describe how a put and get operation works.
b) How can you find an item that is directly at your GPS coordinate?
c) Items at my exact GPS coordinates are not too useful, what do I have to do to look up items close to my GPS coordinate?

Solution:
a)
A user stores a spot in the DHT with a put command.
Spot s = new Spot(GPS Coordinate, SpotName, Author, Rating, Description, Date, …);
DHT.put(s.getKey(), s)

The put command operates on a KBR and routes the message to the node holding the key of the data item. It may be replicated by the DHT to k close neighbors to circumvent failures.
The DHT.get operates in simular fashion, but the node gets a return message with either the list corresponding spots or a failure.

The spots are stored at the hash of the coordinate. Since the lower bits of the coordinate are not relevant (due to noise and due to irrelevance of a high resolution that is smaller than even a small spot) Therefore, we apply an accurracy function that removes the bitlength to a reasonable accuracy. h(accurracyfunction(GPS.x,GPS.y))

b)
Use the get message of the DHT.
key = h(accuracyfunction(GPS.x,GPS.y));
Spot s = DHT.get(key)

c)
It is necessary to perform get request to all coordinates in the given proximity of the GPS coordinate.
➔ DHT.get for coordinate, for coordinate (x + 1 tick), for coordinate (x - 1 tick), for coordinate (y + 1 tick), for coordinate (y - 1 tick), for coordinate (x + 1 tick, y + 1 tick), … and many more coordinates that are within the given distance of the coordinate. ➔ Problem: depending on the resolution a large number of requests have to be made (even for low resolution, this may be 20 or even 100s of requests)

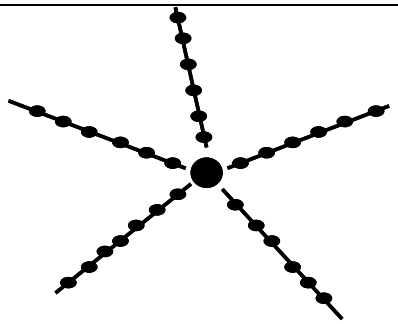
Options:

- exptected: basically adapt the resolution to a reasonably low value that is a good compromise, the requesting node can then filter the spots that are too far
- *an alternative that is not DHT-compatible as it would involve changes in the DHT implementation(!):* nodes not only reply with the requested coordinate, but with others they also have that are within the given proximity

Task 2 An unusual topology

The image on the right shows a solar-like topology. Peers are positioned on 5 rays that are connected via a central peer. The topology further imposes the constraint that the largest ray should not be more than twice as large (with respect to number of nodes) as the smallest ray.

a) Describe a topology-preserving join procedure.
b) What is the largest distance in the topology (let s be the number of nodes on the smallest ray without the central node)?
c) What happens when a node fails? What can you do to make the topology resistant to a failure of a single node?



Solution:

a)

Assume that the root knows the size of the rays. This could be checked regularily with couting packets for maintenance.

A join is done via a known node.
Step 1: Contact the root to ask if the node can join this ray.
→ If yes: add the node to the ray (either directly as neighbor of the node via which it joined, or at random position or at the end of the ray)
→ If no: route via the root to the smallest rays (known by the root) and join there (either directly as neighbor of the root, or at random position or at the end of the ray)

Btw, this corresponds to a complexity of O(n) as the root itself may not be known and the rays are O(n) long (due to constant number of rays). It could be reduced to O(1) for most cases if all nodes know the root (problem: the root can change).

b)

The longest distance is the distance from the endpoints of the longest rays. From the size rule they can at most be twice as long as the shortest rays. This case can be created here.
→
2 rays with twice the length ($2s = 2/7 * n$) – shortening any of the rays would reduce the distance.
3 rays with short length ($s = 1/7 * n$)
→ $D_{max} = 4s = 4/7 n$

c)

When a node fails, a ray can be split into two rays, one ray connected with the root, the other disconnected. If the root fails, all rays are disconnected.
How to resolve:

- not only link to predecessor and successor but to k predessors and successors. Nodes close to the root and the root itself will link to a corresponding predecessor in all rays.

Task 3 Consistent Hashing – Distribution of Interval sizes

In this task we want to compute the distribution of the interval size in systems on the basis of Consistent Hashing like Chord. Let us assume the ID space to be real-valued in the interval [0,1). Without loss of generality we can put our node on the position 0 in the ID space.

a) Nodes are positioned randomly on the basis of uniform random numbers. What the cumulative distribution function (CDF) L of the corresponding uniform distribution.
b) Now calculate the CDF for the minimum of n-1 independent experiments with the distribution from a). Hint: The CDF for the minimum von random variables with CDFs L_1, L_2, L_3, … is given by the forumla $L\_\min = 1 - \prod_i (1 - L_i)$.
c) Now differentiate L_min to get the probability density function l_min.
d) Plot the probability density function l_min.

Solution:

a)

Uniform distribution in interval [0,1).

U(x) = x für 0<=x<=1

b)

Apply formula for the minimum

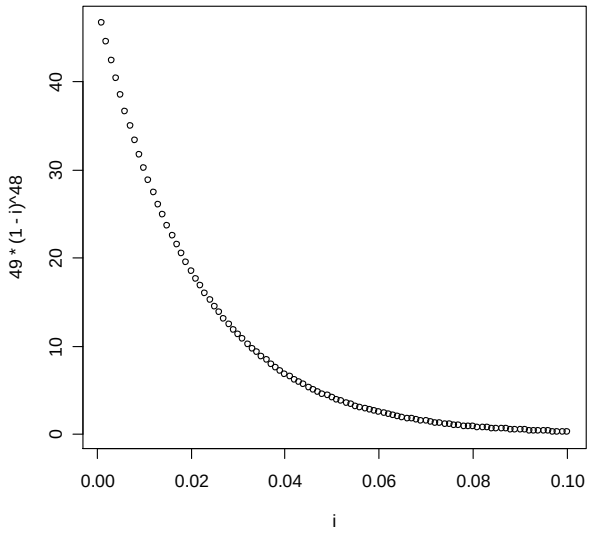
$L(x) = 1 - (1 - U(x))^{n-1} = 1 - (1 - x)^{n-1}$ für 0 <= x <= 1

c)

Differentiate L(x)

$$l(x) = \frac{dL}{dx} = -(1-x)^{n-2}(n-1)(-1) = (n-1)(1-x)^{n-2}$$

d) Plot for n=50



Task 4 A flexible Chord

The finger table entries in the classic Chord algorithm always point to the first node in the corresponding finger interval. This does not allow the freedom to select a finger among multiple peers. Yet, there are proposals to allow Chord to link to any node in the interval of the finger. If you remember the proof for the complexity of the lookup of O(logn) in Chord, we needed that the distance is halved per step.
Show that despite of that change, Chord still achieves O(logn) hops with high probability.

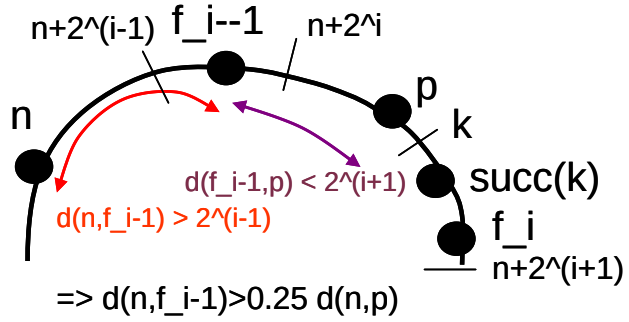
Solution:

Nodes position themselves randomly. The finger table follows the strategy above. The ID is arbirtrary.
The basic difference to Chord is not that not necessarily the first node in the interval i is taken. This means that you mean search for an ID with a responsible node in the interval that is smaller than the node in your finger table for the interval. This is new compard to the old case.

So, we may only utilize the link to the preceeding interval i-1. This may also happen in classic Chord when the first node is the successor(ID) and responsible for it. In that case, its predecessor is not in this interval. In Chord we always need to find the predecessor first.

Assumption: n is large and relevant intervals contain nodes.

d(n,f_i-1) > 2^(i-1)
d(f_i-1,p) < 2^i+2^(i-1)
=> 1/3 d(f_i-1,p) = 2^(i-1)
=> d(n,f_i-1) > 1/3 d(f_i-1,p) = 1/3 (d(n,p)-d(n,f_i-1)) da d(n,p) = d(n,f_i-1) + d(f_i-1,p)
=> 4/3 d(n,f_i-1) > 1/3 d(n,p)
=> d(n,f_i-1) > ¼ d(n,p)
=> per step we progress 0.25 of the way, so up to 0.75 remain.
=> in 3 steps with reach the case when only 0.42 of the way remain and 0.58 were progressed.
=> in O(1) steps we reach the case of progressing more than 50 % of the way which we needed in the proof for the classic Chord.



Task 5 Distance and links

The more links each node in a DHT has, the shorter the distance. Yet, one does not achieve the full benefit of having such links when the strategy is not good.

Assume first, that a node sits on a ring-like ID space which we simplify to the interval [0,1). Each node links to its successor and predecessor. Each node has 100 long distance links. As long-distance links each node links to nodes i=1..100 in the distance i/100.

a) Does this approach achieve logarithmic distance?
b) Does this approach needs further measures to ensure that the ring does not break? What is different in Chord?

Consider also the result of task 4 for the next subtask. Assume now that you add links in distances ¼, ½, ¾, 13/16, 14/16, 15/16, 61/64, 62/64, 63/64, ... The basic idea is to divide the first interval and then the most distant intervals into quarters.

c) How does this approach affect the distance? (Hint: What reduction is achieved within one hop.)

Solution:

a)

Considering the ID space to be of size 1.

No, because fingers only help to reach (for sure) a distance of 1/100 to the target. From then on, the message has to be handed from neighbor to neighbour. Thus, making it O(N)

b)

Here only the direct successor and predecessor are known. If one of them fails, only distant nodes are known, while in Chord even without a successor list, most successors will be directly in the finger table. So, yes, here we need a successor and predecessor list also for the normal maintenance.

c)

If we do routing in forward direction, we cannot efficiently route to close nodes, thus this would lead to an inefficient routing. However, reversing the direction, we will have a similar situation to Chord in backward direction. So, nodes in the preceeding ID space can be found efficiently.

In that case, however, we can not only half the ID space with one step, we can do bring it down to a fourth. Thus, O(log_4(n)) given the standard high probability argument in Chord (compare it to Pastry with b=2, => 4-ary tree).