6. **Naming systems**. Three types:
- **Flat naming** identifiers are just random bit strings; name does not contain any information on how to locate the address of its associated entity. Name to address resolution: simple solutions for LANs (broadcasting, forwarding pointers), home-based approaches, Distributed Hash Tables.
- Structured naming (ex: www.cs.vu.nl):
  - Name spaces: labelled directed graph, given a path name, a name resolution retrieves the data in the node referenced by that name. Availability and performance are met by replication. It is normal that lower-level name servers have a lower TTL than higher ones.
  - Name resolution: each client accesses a local **name resolver**. **Closure mechanism** is needed: know how and where to start the resolution. Caching is more effective with recursive resolution because the name servers also learn information through the process and provides better geographical scalability but also puts higher performance demand. When you receive a reply you also get the TTL of the address you just received, but it is not explained on this example.
  - There can be hierarchy between client and resolvers: three name resolvers A->B->C, then B before querying the root server, it queries cache of resolver A, and C before querying the root server queries the cache of resolver B, and if it's not in B then it queries the cache of resolver A.
  - There can be cooperative caching, they can see each other's cache and if neither of them have the info then that's when they query the root server.
  - Iterative name resolution: name resolver queries each name server in an iterative fashion.
    Client->Client's name resolver (cnr): 1. queries [nl, vu, cs, ftp],6. receives #[nl, vu, cs, ftp]
    2. Cnr->root name server: queries [nl, vu, cs, ftp], receives #[nl], [vu, cs, ftp]
    3. cnr->name server nl node: queries [vu, cs, ftp], receives #[vu], [cs, ftp]
    4. cnr->name server vu node: queries [cs, ftp], receives #[cs], [ftp]
    5. cnr->name server cs node: queries [ftp], receives #[ftp]
  - Recursive name resolution: name resolver starts the process and each server queries the next one it finds until the resolution is satisfied; results are then returned to the client.
    Client->Client's name resolver (cnr): 1. queries [nl, vu, cs, ftp], 10. receives #[nl, vu, cs, ftp]
    Cnr->root name server: 2. queries [nl, vu, cs, ftp], 9. receives #[nl, vu, cs, ftp]
    Root name server->name server nl node: 3. queries [vu, cs, ftp], 8. Receives #[vu, cs, ftp]
    Name server nl node->name server vu node: 4. Queries [cs, ftp], 7. Receives #[cs, ftp]
    Name server vu node->name server cs node: 5. Queries [ftp], 6. Receives #[ftp]
- **Attribute-based naming**: {attribute, value} pairs. **Directory service**: special kind of naming service in which a client can look for an entity based on a description of its properties instead of its full name (ex: LDAP). Each directory entry is uniquely named using a sequence of naming attributes.

11. Mobile Computing: **Wireless Sensor Network**
- You're given **a table** with different points labelled with letters (A, B, C, etc) and the **distance of each square** in the table (it varies depending on the problem, be careful).
- Calculate distance between two nodes p and q: $d(p,q) = \sqrt{(q_x - p_x)^2 + (q_y - p_y)^2}$. With this information you can calculate the power needed, which depends on the model:
  - Broadcast Incremental Power:
    - The algorithm is explained in the exercise. "Inicialment, només el node S forma part de l'arbre, i la seva potència de transmissió és 0. Pas 1: Determinar quin nou node es pot afegir a l'arbre com a descendent de qualsevol node existent amb el mínim consum de potencia **addicional** i actualitzar la potència de transmissió del node existent escollit. Pas 2: Repetir el pas anterior fins que tots els nodes formin part de l'arbre."
    - Calculate all the distances and the power needed to transmit (depends on the formula you are given), and then choose and add the node that has the lowest power needed.
  - Network discovery services can be implemented in three ways:
    - **Directory server**:
      - Issue a multicast request to locate the server. It will respond with its unicast address. Then, point-to-point communication.
      - Directory server must deal with services that disappear spontaneously: it maintains a service's registration only if the service periodically renews its **lease** on the entry.
    - Serverless:
      - Participating devices collaborate to implement a distributed discovery service.
      - Push model: services multicast ('advertise') their descriptions regularly. Clients listen for the multicasts and run their queries against them.
        - Client sends directly to the service node, calculate this energy.
        - Service is sent to all nodes, calculate this energy.
      - Pull model: clients multicast queries. Devices respond with service descriptions that match.
        - Client sends to all nodes a query, calculate this energy.
        - Service nodes matching the query send a message to client, calculate this energy.
    - **Direct physical association** (no discovery service): human enables the carried device to learn the network address of a 'target' device.
  - **Adaptation**: there are two main techniques for dealing with those changes to resource levels at runtime (screen size heterogeneous, proxies to adapt to volatile components, etc):
    - **Middleware support**:
      a) Notify the user of reduced resource availability so can adapt to use less of that resource.
      b) Allow reservations that guarantee a certain level of a resource.
      c) Suggest a corrective action to the user to get an adequate resource supply.
    - Cyber foraging: a processing-limited device discovers a compute server in a smart space and offloads some of its processing to it.
      - Device should still run correctly (albeit more slowly or with reduced fidelity) if no server is available.
      - Offloading should incur low communication between the device and the server.
      - There's a base station BS and the other normal nodes (A, B, C, etc). The idea is to calculate if it's more expensive to calculate it in the BS or in the node itself.

9. Peer-to-Peer Systems: Chord concepts

- **Identifiers** are ordered on an identifier **circle** modulo 2^m (they go from 0 to $2^m - 1$). m is the number of bits.

- **A key is stored** on its **successor** (node with next higher ID, a key is identified by an ID). Each node has $O\left(\frac{K}{N}\right)$ keys.

- **Finger tables:** each node knows m other nodes and requires $O(\log N)$ hops to search.

  - **Creation:** Entry "i" in the finger table of node "n" points to node $n + 2^i$ (if any) **or its successor** if such node isn't present on the ring.

- **Joining the ring (basic),** three steps:

1. **Initialize fingers and predecessor of new node** "j". Locate any node n in the ring, ask n to lookup the peers at $j + 2^0, j + 2^1, j + 2^2$. Use the results to populate the finger table of "j".
2. **Update finger tables of existing nodes**. For each entry "i" in the finger table, new node "j" calls update function on existing nodes that must point to "j". Nodes in the ranges: $[pred(j) - 2^i + 1,\ j - 2^i]$.
3. **Transfer keys responsibility**. Connect to successor and transfer keys in the range $(pred\ ID, node\ ID]$ from successor to new node.

- **Joining the ring (stabilization),** three steps:

1. Join only initializes the finger to successor node.
2. All nodes run a stabilization procedure that periodically verifies successor and predecessor
3. All nodes also refresh periodically finger table entries.

- **Basic lookup,** the request is forwarded from a node to its successor. Requires $O(N)$ hops.

- **Finger tables lookup,** three step algorithm:

a) Is the key stored locally?
b) If key $ID\ \epsilon\ (node\ ID, succ\ ID)$ then forward query to successor. (Key between node and its successor).
c) Else, forward query to the largest node in finger table **not exceeding** key ID.