Distributed Systems - Enlightenment Guide Part 3 And 4

Operating System Support

Book: 10.2.1, Chp. 3
You can dig into the topic with a Paper on the development of User Interface Tools.
In the CORBA spezification in Chpt.11 (POA - Portable Object Adapter) you can find a particular implementation of the object adapter principle.
Fragen:

1. **Explain the difference between process and thread. What has the developer take care of when using multithreading? (p70f) Explain the importance of threads in distributed systems, in particular client/server systems. (p75-79)**

Process: Often defined as a program in execution, that is, a program that is currently being executed on one of the operation system's virtual processors (=OS has a number of them). The operation system takes great care to ensure that independent processes cannot maliciously or inadvertently affect the correctness of each other's behaviour → multiple processes share same CPU and other hardware are made transparent → high price (independent address space, if OS supports more processes than it can simultaneously hold → swap memory/disk)

Thread: Executes its own piece of code, independently from other threads. In contrast: no attempt to achieve a high degree of concurrency transparency → maintains only a minimum amount of information (CPU context + information for thread management). → No protection of data against inapropriate access by other threads → left to developer.

Multithreading:

- **Safety** - how to **synchronize** threats so that they do not interfere with one another (e.g. array read/write, deadlock)
- **Liveness** - how to **avoid deadlock** situations to ensure that all threats make progress (fairness?)
- **Performance** - Overhead (performance penalty) from context-switching and synchronization

Importance of threats in DS:

- convenient means of allowing **blocking systemcalls** without blocking the entire process in which the thread is running (but you have to ensure that a blocking threat does not block the entire process → LWP (lightweight processes))
- makes it much easier to express communication in form of maintaining **multiple logical** connections at the same time.
- conceal long interprocess message propagation times → initiate communication and immediately proceed with someting else (e.g. webbrowser).
- simplifies server code considerably
- develop servers that exploit parallelism to attain high performance

2. **Which aspects of distributed systems are important when designing the client? How are user interfraces integrated into the architecture of a distributed system? How can different types of transparencies be supported? (p82-88)**

- Interact with **human user and remote server in parallel** (non blocking towards user) → achieved through multithreading (e.g agenda)

- **Hide communication latencies**: Do something else in parallel (e.g. Web browser, webpage loads not as a hole but bit by bit)
- **Load balancing** (only useful if supported on server side): connections to different replicas, data transfer in parallel (e.g. google.com → every time another webserver fitting your needs)
- **Local data processing**: less data transfer (JS, form filling → database queries)
- **Compound documents**: seamless integration of various different kinds of documents (text, image, …) at user interface level. The UI hides that different application operate on different parts of the document (notifications are shared!). remote desktop!
    - **Drag-and-drop:** e.g. move file to trash → not only rearrange icons but pass filename too
    - **in-place-eiditing** (notifications): e.g. if an image is displayed in a wordprocessor and the user places the mouse above the image, the information is passed to a drawing program to allow the user to modify the image. The UI finds out the modification and passes the info to the wordprocessor

User interface integration:
Two ways:
- **Fat-client approach:** for each remote service the client will have a seperate counterpart that can contact the service over the network (e.g. agenda running on a PDA)
- **Thin-client approach**: Provide direct access to remote services by only offering a convenient user interface → Client is only used as a terminal without local storage. → ease task of system management
    - application logic and user interaction should not be thighly coupled, but are in reality → many requests to the server → long latencies in WANs → compress / only send differences when handling similar data (indicated through paket id)

Client side support for transparency:
Ideally the client should not know that data is replicated.
- Components for distribution transparency
    - **access** transparency: provide a stub (interface) → hides possible differences in machine architecture, as well as the actual communication
    - **location, migration and relocation** transparency: naming service and, cooperation with client-side software (inform) → (re-)binding to a new server → client middleware hides geographical location
    - **replication** transparency: proxy → forward request to each replica
    - **failure** transparency: retry, redirect to another server, cache (through client middleware)
    - **concurrency** transparency: transaction monitor (special intermediate server)

3. **Explain some basic design considerations for servers. Describe the difference between stateful and stateless. Depict architecture and function of a multi-threaded server (e.g., a file or web server). (p88-92)**

Basic design considerations:

- **Iterative server:** the server itself handles the request and, if necessary, retunrs a response to the request client
- **Concurrent server:** does not handle the request itsef, but passes it to a seperate thread or another process, after which it immediately waits for the next incoming request. (e.g. multi threaded server or forking a new process for each request → thread/fork responsible for returning result)
- **server interrupt / out-of-band control**: Sometimes it is necessary to interrupt a server. (e.g. cancel FTP file transfer). One approach: kill the application and restart it like nothing happended → server tears down connection. A better approach is to send out-of-band data: data that is processd by the server before any other data from the client:
    - let the server **listen to a seperate control end point** to which the client sends out-of-band data, while at the same time listening to normal data (with lower prio)
    - Send out-of-band data throuth same connection (e.g. TCP urget flag)
- States
    - **stateless server**: does not keep information on the state of its clients, and can change its own state without having to inform any client (e.g webserver → uploading or mostly fetching data → forgets the client completely) (note: logs are nonetheless maintained)
    - **stateful server**: maintains persistent information on its clients → information needs to be explicitly deleted by the server. (e.g. file server where the server tracks the local copies of files). problem: if the server crashes the table has to be recovered → complex
    - **soft state:** particular form of stateless design, server maintains state on behalf of the client for a **limited time**. after time is over the server falls back to default behaviour → discarding any information (e.g. server informes client about updates for a limited time)
    - **session state:** often used in three-tired client-server architectures, where application server needs to access a database server through a series of queries before being able to respond → no harm done if session state is lost
    - **cookie**: client keeps server information and sends it there

- Finding the server
    - **End Point (port):** Client sends request to an endpoint, each server listens to a specific point
    - **Name or directory servers / deamon**: the deamon keeps track of the current end point of each serice implemented by a co-located server, the deamon listens to a well-known port
    - **Well known ports**: 0-1023, globally assigned by the IANA (Internet Assigned Numbers Authority) → only the network address has to be known by the client
    - **Superserver:** single superserver listening to each end point associated with a specific service, e.g. UNIX *inetd*. For each new request a new process is forked to take further care of the request.

Architecture / function of a multi-threaded server

The developer has to take care about concurrency control because each thread has access to the same set of data → can be inconsistant. the multi-threaded server consists of a dispatcher and a number of worker threads. the dispatcher is waiting for incoming requests and deligates them to the worker threads. Therefore is the dispatcher always available to hand over new requests.

4. **What are the particularities of object servers? How can the invocation of an object be realised on the server side (e.g., policies regarding thread, code sharing, and object creation/activation)? What is an object adapter? (p451-454)**

- Provides means to access objects remotely accoording to different activation policies
- Object creation on invocation vs. during server initialization
    - **On invocation**: create a transient object at the first invocation request and destroy it as soon as no clients are bound to it anymore. → server's resources are only as long needed as the object is needed but may take some time to complete
    - **Initialization**: create all objects when the server is initialized → many resources are consumed even when no client is making use of the object
- Seperate memory segments vs. sharing code (class definition)
    - **seperate memory segment**: neiter code nor data are shared → necessary when object implementation does not seperate code or for security reasions (own address space)
    - **sharing code**: provide a database containing objects that belong to the same class → efficiently implemented → implementation only loaded once
- Thread policies (single, one per object, one per request, …) and pool vs. on demand
    - **single thread**:

- several threads:
  - one for **each object:** when an invocation request comes in it is passed to the thread responsible for the object, but if the thread is already busy the request is temporarily queued → automatically protected against concurrent access → serialized
  - one per **request**
- pool vs on demand: independent from thread policy (performance, …)

- → object adapter implements policies

Object Adapter

- An object adapter (OA) implements a particular **activation/invocation** policy (OA groups objects per policy)
    - Software implementing a specific activation policy
    - generic component to assist developers of distributed objects, only configured for a specific policy
    - policies can be configured at runtime
- Different OAs may co- exist in the same server dispatcher
    - each adapter has one or more objects under control
    - a request is dispatched to the appropriate OA
- OA is unaware of the specific object interface (--> generic!)
- OA extracts reference and invokes skeleton according to policy
    - request is not passed directly to the object, an adapter hands an invocation request to the server-side stub of that object.

Implementation details are hidden for the object adapter who communicates only with a skeleton!
servant = general term for a piece of code that forms the implementation

5. **Exlpain the most important aspects of code migration, including strong and weak mobility. Provide an example for weak mobility. (p103-107)**

process migration: the entire process is moved from one to another machine. (costly & intricate task)

- Parameters passed among client and server may refer not only to data but also to code

– perhaps even while being executed!

- Historical: **Balancing** of computational load
  - move process from heavily-loaded to lightly-loaded machine
- Distributed Systems: Moving code to where the data is can **lower communication overhead**:
  - move query processing to database machine
  - moving code to client can also improve scalability (e.g. form processing → database operations created on the client)
- **Exploit parallelism** (linear speed-up of e.g. Web search)
  - Implement a search query in the form of a small mobile programm, calles **mobile agent** that moves from site to site, sent to different sites
- **Flexibility**: Moving code can be used to customize (i.e. configure) the client dynamically and provide service interfaces on demand: Improved versioning and evolution
  - 1) server implements a standard interface (e.g. for accessing the file system)
  - 2) Server provides the client's implementation no sooner than necessary (when the client binds to the server) → client automatically downloads the implementation (requires: standardized protocol, code can be executed on the client's machine)
  - → clients need not have all the software preinstalled → implementation can be changed

- BUT **Security**?
  - not present, client blindly trusts the downloaded interface because it can access the hard drive unprotected
- BUT Code migration in heterogeneous systems is costly and intricate!
- Mobile code offers a different paradigm for structuring of distributed applications

To get a better understanding of the different code migration models:

- **code segment:** set of instructions that make the program being executed
- **resource segment:** contains references to external resources needed by the process, e.g. files
- **execution segment**: store the current execution state of a progress (private data, stack, program counter)

- **weak mobility**: minimum of code migration, it is only possilbe to transfer the code segment and some initialized data. the program always starts from the same predefined stating point (e.g. java applets → always starts at the beginning, simple)
    - **execution at target process**: the application is downloaded and executed in the processes address space (e.g. java applet in web browser address space)
        - no seperate process needs to be started → no communication at the target machine needed
        - target processes needs to be protected against malicious or inadvertent code executions
    - **execution in a seperate process**: simpler solution because there is no resource-access problem (just every single program you download - own process on execution).
- **strong mobility**: execution segment can be transferred as well. A running process can be stopped and moved to another machine where it resumes execution. → more general approach which is harder to implement.
    - **migrate process**: moving a running process.
    - **remote cloning**: yields an exact copy of the original process and executes it on a different machine (e.g. forking a child process on a UNIX system) → simple way to improve distribution transparency
- **sender initialed**: migration is initialized at the machine where the code currently reside, e.g. when uploading programs to a computer server
- **receiver initialed**: the initiative is taken at the target machine, e.g. Java Applets. Simpler.

6. **Describe the concept of virtualization and the meaning for code migration in distributed systems. What different kinds of virtual machines do you know? (p79-82)**

- Threads provide “illusion” of several processors → Virtualization extends this idea to the resources
    - extend or replacing an existing interface so as to mimic the behaviour of another system
- 70s: run legacy software (e.g. IMB 370 and succesors)
- 90s:
    - Application software is outliving system (operating system and hardware) → portabilty: It is now possible to port the legacy interfaces to new platforms and thus immediatley opening up the legacy system classes for existing programs
    - Administration of large and heterogeneous systems and applications → **flexibility + portablity**: The diversity of platforms and machines can be reduced by essentially letting each application run on its own virtual machine, possbiley including the relate libraries AND operation system, which, in return, run on a common platform
    - **Isolation**: → Reliablilty and security

(a) General organization between a program, interface and system
(b) General organization of virtualizing system A on top of system B.

Various interfaces offered by computer systems:

1. General Instructions: Interface between hardware and software
2. Privileged instructions: Can only be executed by privileged programs
3. Sytem calls: Offered by the OS
4. Library functions: Often known as application programming interface (API), mostly the library function is hidden by the API.

Process Virtual Machine with multiple instances of **(application, runtime)**-combinations, e.g. JVM

Here a runtime system provides an abstract instruction set that can be used for executing applications. The instructions can be interpreted (JVM) or emulated (the behavour of system calls have to be mimiced as well, e.g. Windows, UNIX)

Virtual Machine Monitor with multipe instances of **(application, OS)**-combinations, e.g. VMware, Xen

A system that is essentially implemented as a layer of completely shielding the original hardware, but offering the complete instruction set of that same as interface. Important is, that this interface can be offered simultaneously ot different programs → multiple and different operation systems run independent and concurrently on the same platform. Therefore the application and it's environment is completely **isolated** → protected against errors and security attacks. The VMM also provides a decoupling between hardware and software and can easily be moved to another machine.

Code migration:
In the beginning it was hard to migrate code between different platforms. With **virtualization** this process was now **easier** than before. One the one hand you can use a virtual operation system to make migration as easy as possible, or on the other hand use the way Java does it. Java code is not compiled to machine instructions but is translated into an "intermediate language", which is platform independent and is interpreted by the JVM. A **high degree** of **portability** and **flexibility** is achieved.

Naming and Discovery (chapter 5)

1. **Explain the terms "Name", "Identifier", and "Address" and relate them.**

Name: is a String of bits or characters to refer to an entity
Entity: can be anything (e.g. printer, disk, machine...) that can be excessed and operated on
Access Point: offers the access to an entity (just another entity in a DS). For instance a telephone can be seen as an excess point, whereas the telephone number would represent an **address**, which is the name of an access point. Entities can change their access points – e.g a router and a mobile computer, or seen from the other side an access point may be reassigned to a different entity - but how then to prevent the reference becoming invalid? Such a move illustrates that a name for an entity that is independent from its addresses is often much easier and more flexible to use. **location independency**
Identifier (uniquely): a name being an identifier would need it to have the following attributes:

- referring to one identity only
- each entity is referred to by only one identifier
- identifier always refer to the same entity

A telephone number as an identifier will not work – because it could be reassigned to another person (taking an company's cell phone for example). Identifiers are usually represented in

machine-readable form (32-64bit strings) – random and defined by the system.
Human-friendly names: are in contrary tailored to be used by humans and should be:

- easy to remember
- case insensitive character strings
- relatively simple
- have meaning to the user

How do we resolve names and identifiers to addresses?
There mostly is a relationship between name resolution in distributed systems and message routing. In General a naming system maintains a name-to-address binding, which is in its simplest form a table of names and addresses. That can lead to problems within a large scaled DS, instead a name is decomposed into several parts (ftp.cs.vu.nl) - and then a name resolution takes place through recursive lookup of those parts – starting from the last part of the name:

e.g: **NS(.)** -> NS(nl) -> NS(vu.nl) -> address of ftp.cs.vu.nl - **".**" is the root server

summed up relation between those three: names and identifiers are two ways to refer to an address.

2. **What is a "Name Space"? Explain the basic principle of the "Closure Mechanism" and provide an example (e.g., Unix file system or DNS).**

Because names are good for machines, but not convenient for humans we need some way to provide structured naming, that is exactly what so called name spaces does, they offer us a way to organize names in a structured manner that are composed from simple, human-readable names. "names are organized into name spaces!". That structure is represented as a labeled directed **graph** with two types of nodes:

- **Leaf node:** 1 incoming and no outgoing edge
- **Directory node:** 1 ingoing – 0 to many outgoing edges

A special node is the **root node,** that has no incoming edge, subsequently is the starting point when it comes to a name resolution. In terms of simplicity naming systems have only one.
Each path in a naming graph can be referred to by the sequence of labels corresponding to the egdes in that path – N: <label1, label2, labelN> - N refers to the first node in the path.
There are different types of paths and two perspectives.

- **relative**: /calendar – is relative to its root server
- **absolute**: e.g http://www.google.com/calendar

- **global**: denotes the same entity, no matter where it is used in a system.
- **local**: a name whose interpretation depends on where that is being used / a name is essentially a relative name whose directory in which it is contained is implicitly known.

In a case of a name space the name is always defined relative only to a directory path.

Different ways to **organize a name space**:

- Most have only a single root node
- In many cases strictly hierarchical, in the sense that the naming graph is organized as a tree, then each node, except the root, has exactly one incoming edge, consequently each node also has exactly one associated absolute path name.

Closure mechanisms: means knowing how and where to start in terms of looking up a name, that is called name resolution (returns the identifier of a node from where the name resolution process continues. They have to be partly implicit and may be very different when comparing them to each other.
Unix as an example: The OS knows that the inode oft he root directory is the first, then it apperantly can always starting or continue from there.

Example: **/home/dropbox/university**

The system maps the home folder to /users/username/ which then represents your root folder in terms of having access to the directory table of this string/name -> systems has to know where to start for a resolution look up!

3. **Describe the different layers of distributed name spaces and how they may benefit from replication and caching. Explain the difference between iterative and recursive name resolution.**

Is organized hierarchically, so to say the name space is partitioned into three logical layers.

Global layer:

- formed by highest level nodes (root node + children nodes)
- characterized by Stability (directory tables are rarely changed)
- Represent organizations/groups of organizations which names are stored in a name space

Administrational Layer:

- directory nodes within a single organization
- represent groups of entities belonging all to the same organization/administration
- e.g directory node for every department
- relatively stable, changes occur more frequently than in the global layer

Managerial layer:

- nodes (representing hosts) change regularly
- maintained by individual end users of a DS

Different layers have to meet different requirements (availability and performance).
High availability for the *Global Layer*, because if it fails a large part of the name space will be unavailable! Low rate of change with client use caching after first request instead of asking again! Many people send requests to the global level **throughput** is important! replicating + client side caching

Availability for A*dministrational Layer* is primary important for clients in the same organization as the name server – less important that resources in an organization are temporarily unreachable for users outside that organization. **Performance** is also important – quicker than with the global layer! It is unacceptable that an account for a new user takes hours to become effective! high perf. Machines + client side caching and replication.

Managerial Layer – **Performance** is crucial!!! availability is less demanding, use single dedicated NS at the risk of temp. unavailability. Client side caching is less effective.

iterative name resolution: name resolver hands over the complete path to the root name server. The Resolver iterates then from name server to server till the name is completely resolved.

recursive name resolution: each server requests the next (lower level) server and the result will be propagated back up till it reaches the client. So the client's reuqest to the root name server (any name server it can find) will be then handed over tot he next server until the whole path is resolved and propagated back. ***Drawback:*** takes more performance on every single name server because it not only has to resolve the address but also hand it over to the next level, subsequently a global layer usually only supports iterative resolution.
Advantages: Caching results is more effective (each name server will learn the address of each name server responsible implementing lower level nodes). Also communication costs may be reduced. Then the next client asking fort he same address gets the result immediately.

4. **Explain the Domain Name System DNS in principle, and the process of name resolution based on the structure of the DNS Database (Resource Records). What is reverse lookup? What is a zone-transfer?**

Domain Name System (DNS) = largest DS naming service today, for looking up IP addresses of hosts and mail servers. Is a hierarchically organized rooted tree. Complete pathname (max 255 chars) separated by a dot Root = dot at the end www.google.at**.** (usually left out for readability)
Each node has one incoming edge.
Subtree = domain.

Path name = domain name (abs/rel)

Contents of a node is formed by resource records (different types of resource recs.)
SOA (Start of Authority): contains the email address of an admin responsible for the represented zone.
A(ddress) record represents a host in the internet (contains IP). Multiple IP → muliple A records
MX (mail eXchange): for mail servers
SRV: contain name of servers for specific services – client does not need to know DNS of a host but rather the standardized service name.
NS (Name Server): for hosts representing a zone
PTR (pointer): with such an record DNS allows for reverse lookups IP > Name > IP
DNS distinguish aliases from what are called canonical (primary) names, by using an **CNAME** (containing the canonical name of the host) record.
HINFO (additional host info)
TXT (any other kind of data)

Each Zone is implemented by a name server, that is virtually always replicated for availability. Updates for a zone are handled by the primary name server!
Zone transfer: secondary DNS Server do not access the database directly but requests the primary server to transfer its content.

DNS Database = Small collection of files containing the descript records of all single hosts.

5. **What is a Directory Service resp. "Attribute-based naming"? Describe the basic structure of the X.500 name space and its LDAP implementation.**

Besides location independency and human friendliness there is other criteria such as **search effectiveness**. There are many ways in which description can be provided for search; a popular in DS is to describe an entity in terms of pairs (attribute, value), referred to as **attribute-based-naming**. By defining the attributes values within a search you refine the search results.
Those attribute-based-naming systems are also known as **directory services.**
The critical part with directory services is **designing** the appropriate **set of attributes** that work for most entries (usually a manual process) –imagine how hard it is to define a unique way of categorizing music in the internet. In LDAP we have got the resource description framework (**RDF**) for that, described as driblets – **e.g (person, name, Alice)** describing a Person named Alice. In RDF each subject, predicate or object can be a resource itself, whereas resources can be anything. References in RDF are URLs.
Searching (through all descriptions) in an attribute naming system with only one machine is not much of a problem but if the data is distributed **performance becomes** an big **issue**.

LDAP (Lightweight Directory Access Protocol)

An approach is to **combine** directory services with structured naming. LDAP is **derived** from **OSI's X.500** directory service, whereas some simplifications had to be made in order to use it.
LDAP consists of a number of **records** (collection of **attribute/value pairs**). Each attribute has an associated type and single or multiple (array) values are allowed.
The collection of all directory entries is called directory information base or short **DIB**.
In order to get an unique name for each record there is the relative distinguished name **RDN,** which is just a defined set of attributes which together form the unique name of a single record.

An example for an RDN:

/C=NL/O=Vrije University/OU=Comp. Sc. – analogous to the DNS name **nl.vu.cs**

Listening RDNs in a sequence leads to a hierarchy of the collection of directory entries – a **directory information tree (DIT)**, that forms a naming graph such as with DNS.
A node in LDAP can simultaneously represent both, a **directory and** a LDAP **record**.
The distinction is supported by two lookup operations:

- **Read**: read a single record given its path name in the DIT
- **List**: list the names of all outgoing edges of a give node in the DIT

The **difference** of implementing an **LDAP**, compared to **DNS**, is that LDAP supports more **lookup operations**. In large scaled directories the DIT is usually partitioned and distributed across serveral servers **(directory service agents)** – corresponding to a zone in DNS.
Client are represented as directory user agents **(DUA).** A DUA is similar to a name resolver in structured naming services. (communicating with the DSA by a standardized protocol).

*Example***:** look for a list of all main servers at a specific university

answer = search("&(=NL)(O=Vrije University)(OU=*)(CN=Main server)")

Searching that way is expensive. The example requires searching for all entries at each department and combining the result in a single answer, that means we need to access **several leaf nodes** of a DIT in order to get an answer. In contrast naming services can be implemented in a way that lookup operation only access **one leave node**.
To circumvent scalability problems with that LDAP has a **global index server** (global catalog) that can be searched first, indicating which domains need to be searched further.
Furthermore it is common to combine **LDAP with DNS**, whereas every tree in LDAP needs to be accessible at the root, which access is provided under a DNS name, that in turn can be found through an SVR record.

6. **How does a location service work in a flat namespace? Describe different solution approaches with respect to mobility and discovery. Explain pro's and con's of "Forwarding Pointers". How do "Home-based approaches" work for mobile devices?**

Identifiers are usually random bit string which lead to so called flat names.

Broadcasting and Multicasting:

Broadcasting: **sending** the **identifier to each machine** (in a local LAN). Each machine has to check whether it has the entity and only machines with an access point send a reply (containing the address of an access point).
Same principle is used in the internet with Address Resolution Protocol (**ARP**) to find data-link address of a machine when given an IP.
Broadcasting becomes inefficient when a network grows (waste of bandwidth/host interruption).
Multicasting: Where hosts can join so called **multicast groups**, which then are identified by an multicast address. When a host sends a message to such an address the networks provides a best-effort service to deliver to all group members.

Example: A is a mobile computer, when it connects to the local LAN it gets an multicast IP address assigned (therefore it is within the multicast group). If you want to locate the entitiy, the network sends a “Where is A” message to the multicast group, A answers the request by replying its current IP. One step further would be to **replicated entities** – then it would require finding the nearest replica that is simply the first that answers. That might sound simply, but in reality it is not easy to select the nearest replica.

Pointers: Another approach to find an entity, in terms of mobility, is to use forwarding pointers. When an entity moves it simply does leave a pointer at its old location that forwards to its new. *Drawback* - if an entity is highly mobile that leads to long chains of pointers, making localization of an entity prohibitively expansive. Secondly, not a single link must be broken (broken links – keep the chain as short as possible!).

*Example***:** just fill in a stub or in that case a proxy-skeleton (is the same as the original idea), only for migration in terms of remote invocation it makes things much easier.

P1 or P2 calls > P3 which is forwarding to the right skeleton at the new place!

Short cuts: In order to keep chains short and simple, after the new forwarded location (skeleton) processed the request it not just sends the response but also the new skeleton identifier back tot he client. The client then can ajust its companion skeleton shortcut.
Only the clients proxy can be adjusted and therefore for server it makes no sense to send the response over the reversed path, but rather directly to the client.
All the skeletons that are no longer referred to by any client become garbage collected.

But what if a chain crashes?

Home Based Approach: A home location is introduced that keeps track of the current location of an entity (mostly the place where the entity was created) – as a fall back mechanism.
Each mobile host uses a **fixed IP** address (**identifier**) from which all communication is directed to its home agent. Whenever a host moves it requests a temporary address for communication (**care-of-address**) that is registered at this home agent, which **forwards** (local network) or **tunnels** (wrapped as an IP packet and send to the care-of-address) all incoming communication for that mobile host. A drawback with that approach is **communication latency**, because you always have to talk to the home location first, which might be at a completely different location. The fixed home location also implies that it always has to exist! Solution traditional naming service - let a client first look up the location of the home (**caching**).