



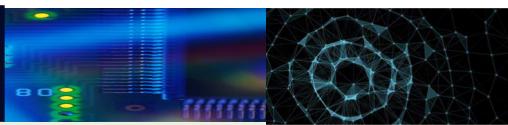
分布式系统

陈鹏飞
计算机学院

chchenpf7@mail.sysu.edu.cn

办公室：学院楼310

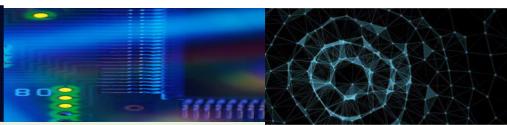
主页：<http://sdcs.sysu.edu.cn/node/3747>



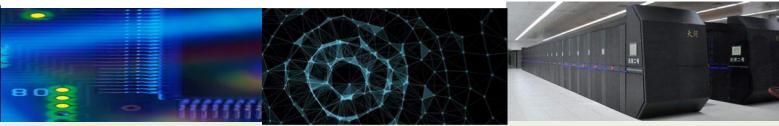
第五讲 — 命名系统



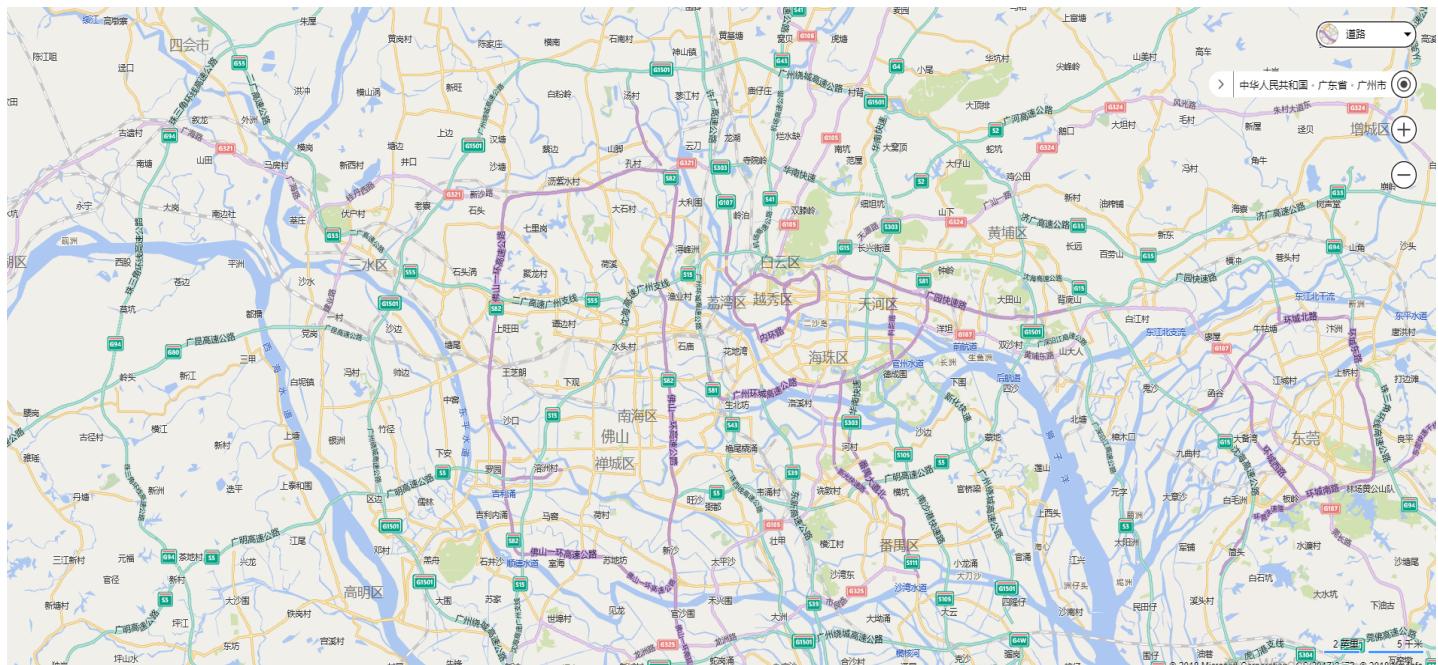
-  命名、标识、地址。
-  无层次命名
-  结构化命名
-  基于属性的命名
-  小结



命名、标识、地址



命名



分布式系统 VS 地图



分布式命名系统



命名系统的实现方式，即分布式命名系统是分布在多台机器上



名称

➤ 本质：

用名字标识分布式系统中的实体对象，由字符组成的字符串。实体可以是分布式系统中的任何事物，包括：主机、打印机、磁盘和文件等硬件资源，还包括：用户、邮箱、消息等抽象资源。

➤ 访问点（access point）

对实体可进行一系列的操作，需要访问实体，因此需要一个访问点。访问点是一类特殊的实体，访问点的名称成为地址。实体的访问点的地址即为该实体的地址。

一个独立于实体地址的名称通常是比较合理的，而且更为灵活，这样的名称是与位置无关



标识符 (Identifiers)

➤ 纯粹的名字:

一个名字没有任何意义；只是一个随机字串。

➤ 标识符：具有某些特殊属性的名字

- 一个标识符至多引用一个实体；
- 每一个实体最多由一个标识符引用；
- 一个标识符始终引用一个实体（标识符永远不会重新使用）

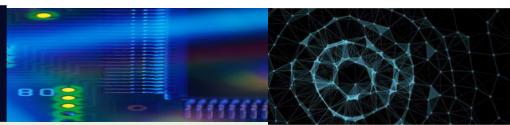
➤ 观察

- 一个标识符不一定是一个单纯的名字，它可以包含特定的内容；



名称到地址绑定

- 原则上，命名系统含有一个名称到地址的绑定，即一个 $(name, address)$ 对的表；
- 对应表的分布式化；
- 标识符解析为地址的过程：
ftp.cs.vu.nl 的解析过程。



无层次命名



广播/多播

➤ 广播 ID， 请求实体返回其当前地址

- 难以逾越局域网；
- 需要所有的进程监听到来的请求；

➤ 地址解析协议 (ARP)

- 找出与 IP 地址相关的MAC地址，广播查询 “who has this IP address?”

➤ 多播

- 网络规模过大后，广播开始变得低效。不仅网络带宽被浪费，而且有太多主机被他们不能回答的请求中断，解决方案:多播。



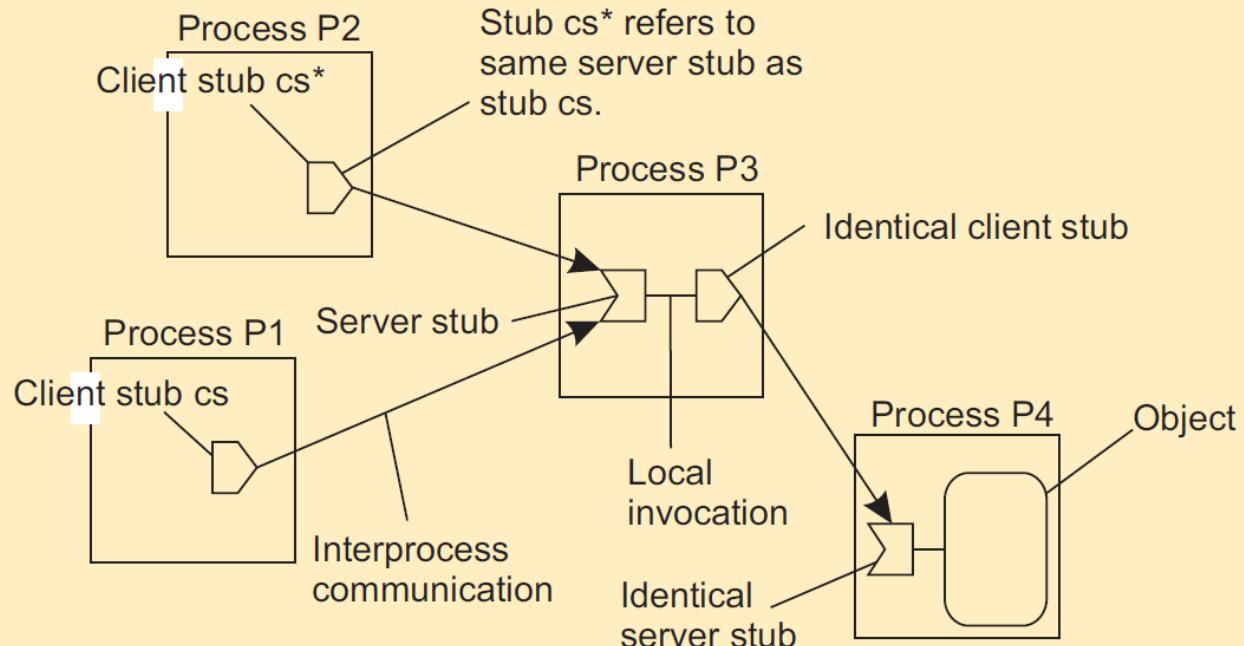
转发指针

- 当实体移动的时候，会在当前位置留下到下一个位置的指针
- 去引用对于客户端来讲变得完全透明，只需要沿着指针链搜索；
- 当实体的当前地址找到后，更新客户端的引用；
- 扩展性问题（地理可扩展性）：
 - 较长的传播链很脆弱，容易断开；
 - 实体的定位开销比较大；



SSP (Stub Scion Pairs) Chains

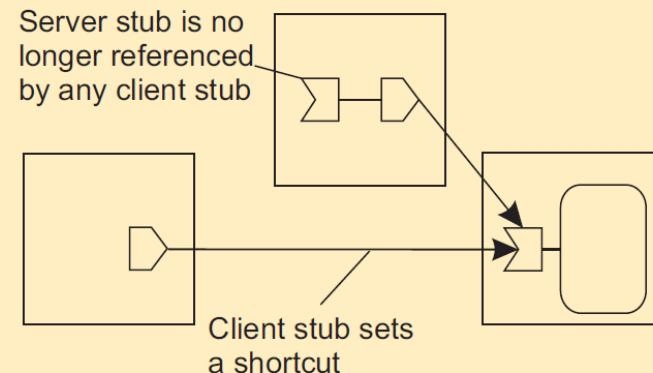
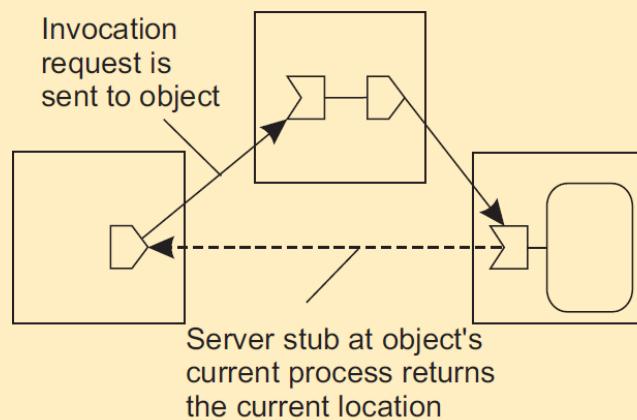
- 利用 (client stub, server stub) 转发指针的原理





SSP (Stub Scion Pairs) Chains

在客户存根中可以通过存储一个捷径达到重定向转发指针的目的；



(a) **如果链中有服务器崩溃，如何解决？** (b)

直接向客户存根发送响应或沿着转发指针的相反方向发送响应，二者各有所长，如果服务器存根不再被任何客户引用，就可以删除它。



基于宿主位置的方法

- 广播和转发指针的使用的主要缺点是可扩展性的问题，且容易受到链断开的影响；
 - 单层模式：让宿主机记录实体的位置
-
- 实体的宿主地址注册在命名服务上；
 - 宿主机注册实体的外部地址；
 - 客户端需要先与宿主机联系，然后与外部地址联系；



移动IP的主要原理

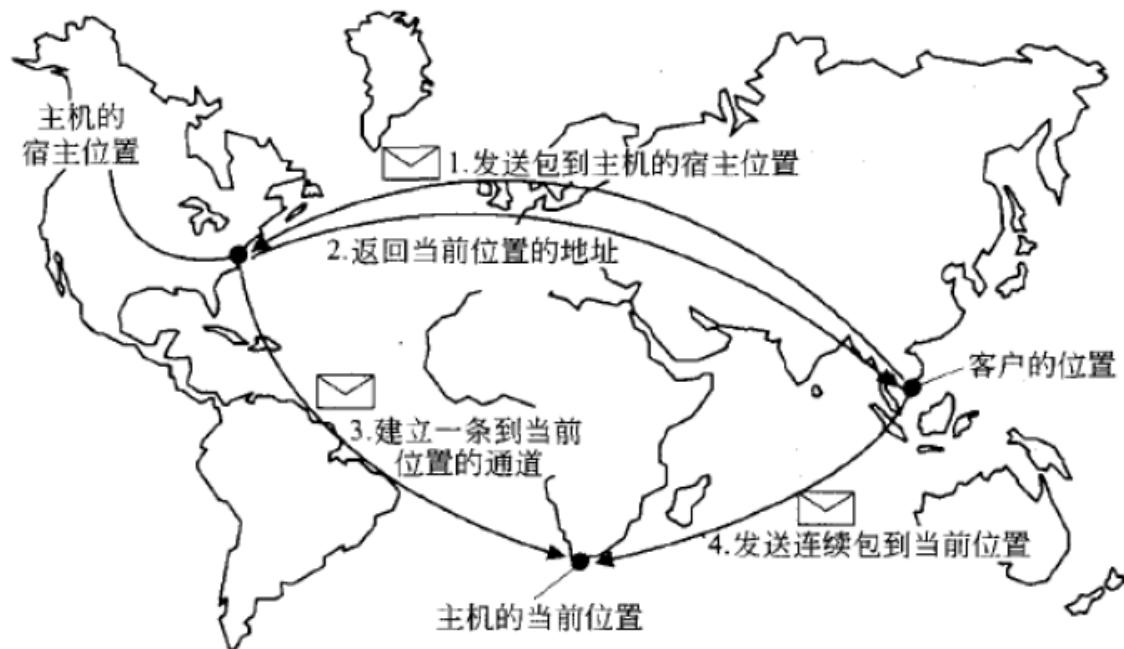


图 5.3 移动 IP 原理



基于宿主机的方法

➤ 基于宿主机的方法存在的问题

- 宿主机需要伴随实体的整个生命周期；
- 宿主机的地址是固定的 => 如果实体对象永久迁移会带来不必要的问题；
- 较差的地域可扩展性（实体可能就在客户端旁边）；

➤ 注意

- 实体永久移动可以通过 **DNS**解决，先通过**DNS**查找宿主机，然后通过宿主机找到实体；



分布式散列表

➤ 众多节点组织成环形结构

- 每一个节点被赋予一个由m位构成的标识符；
- 每一个实体被赋予一个唯一的m位的键值；
- 键值为 k 的实体存储在满足 $id \geq k$ 的最小标识符节点上，成为 k 的后继者， $\text{succ}(k)$ 。

➤ 无扩展性搜索方法

- 令每一个节点记录它的邻居，并且进行线性的搜索；



Chord指状表 (Chord finger tables)

➤ 原理

- Each node p maintains a **finger table** $FT_p[]$ with at most m entries:

$$FT_p[i] = succ(p + 2^{i-1})$$

Note: the i -th entry points to the first node succeeding p by at least 2^{i-1} .

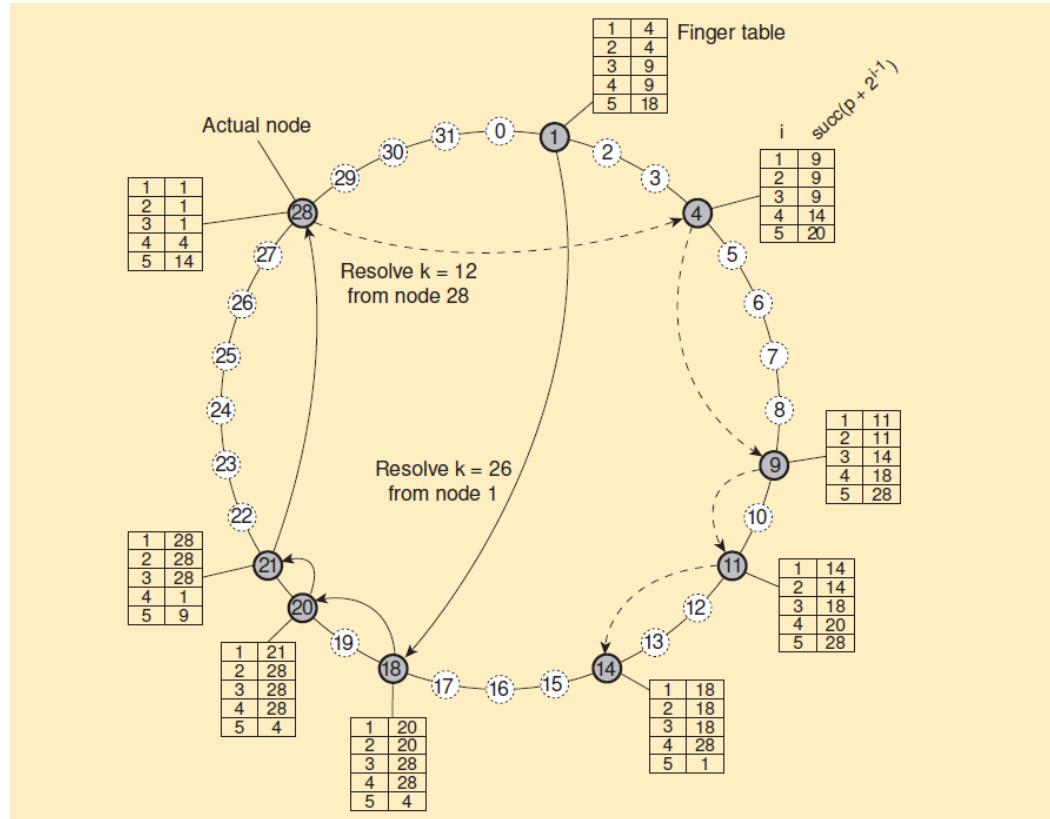
- To look up a key k , node p forwards the request to node with index j satisfying

$$q = FT_p[j] \leq k < FT_p[j+1]$$

- If $p < k < FT_p[1]$, the request is also forwarded to $FT_p[1]$



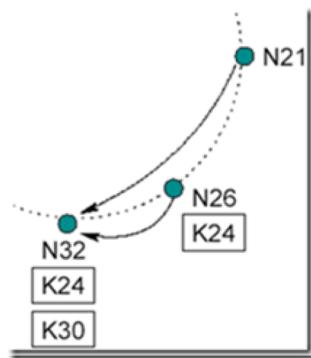
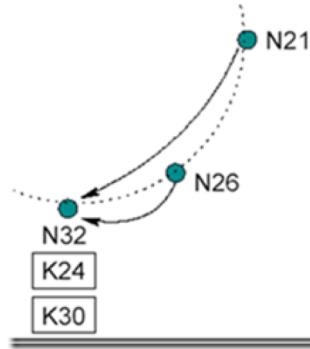
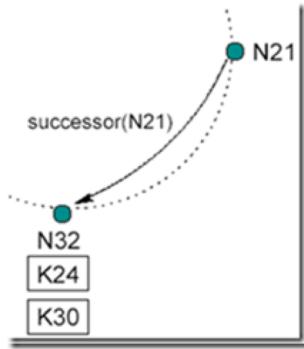
Chord查找例子



在节点1解析
 $k=26$;
在节点28解
析 $k=12$;



Chord 节点的加入和退出



后台进程定期巡查，保持更新



Chord的Python实现

```
1  class ChordNode:
2      def finger(self, i):
3          succ = (self.nodeID + pow(2, i-1)) % self.MAXPROC      # succ( $p+2^{(i-1)}$ )
4          lwbi = self.nodeset.index(self.nodeID)                  # self in nodeset
5          upbi = (lwbi + 1) % len(self.nodeset)                  # next neighbor
6          for k in range(len(self.nodeset)):                      # process segments
7              if self.inbetween(succ, self.nodeset[lwbi]+1, self.nodeset[upbi]+1):
8                  return self.nodeset[upbi]                         # found successor
9          (lwbi, upbi) = (upbi, (upbi+1) % len(self.nodeset)) # next segment
10
11     def recomputeFingerTable(self):
12         self.FT[0] = self.nodeset[self.nodeset.index(self.nodeID)-1] # Pred.
13         self.FT[1:] = [self.finger(i) for i in range(1, self.nBits+1)] # Succ.
14
15     def localSuccNode(self, key):
16         if self.inbetween(key, self.FT[0]+1, self.nodeID+1): # in (FT[0],self]
17             return self.nodeID                                # responsible node
18         elif self.inbetween(key, self.nodeID+1, self.FT[1]): # in (self,FT[1])
19             return self.FT[1]                                 # succ. responsible
20         for i in range(1, self.nBits+1):                   # rest of FT
21             if self.inbetween(key, self.FT[i], self.FT[(i+1) % self.nBits]): # in [FT[i],FT[i+1]]
22                 return self.FT[i]                                # in [FT[i],FT[i+1])
```



利用网络邻近

➤ 问题

- 覆盖网络中的节点之间的组织结构可能导致在底层互联网上的异常的消息传输：如节点 p 和 节点 $\text{succ}(p+1)$ 实际距离非常远；

➤ 解决方法

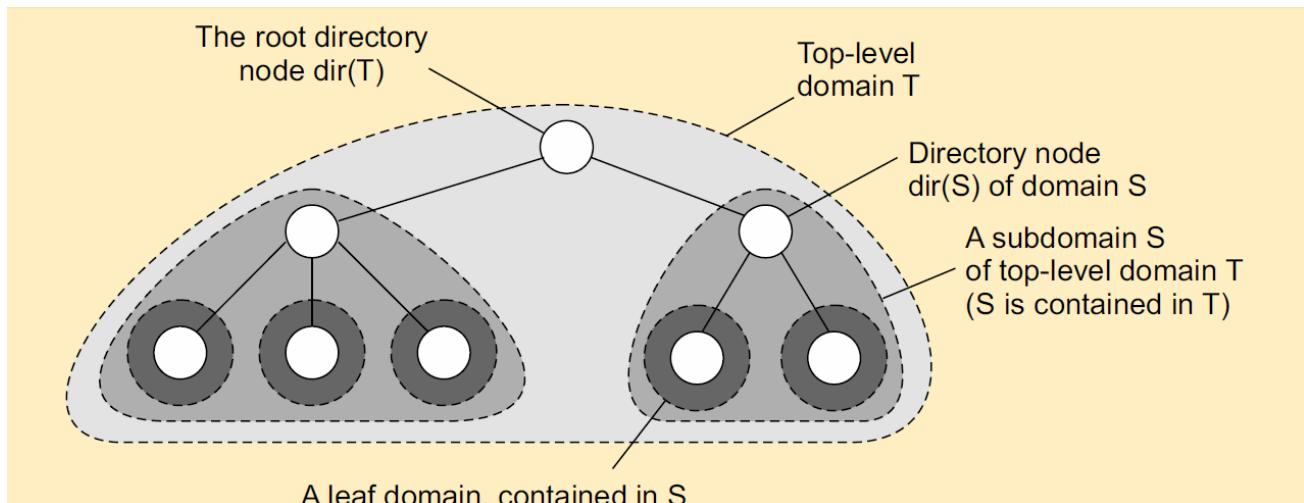
- 基于拓扑的节点标识符赋值：其思想是在标识符赋值时，两个邻近节点所赋予的标识符也是靠近的，在chord系统中存在严重问题；
- 邻近路由：节点维护一个转发请求的可选列表。每个节点有多个后继者，查询时选择最近的一个。例如：对于节点 p $FT_p[i]$ 指向 $[p + 2^{i-1}, p + 2^i - 1]$ 区间内的第一个节点，但是也可以指向其他节点，选择距离最近的一个；
- 邻节点选择，优化路由表，使得选择最近的节点作为邻结点；



分层定位方法

➤ 基本原理

- 创建一个大规模的搜索树，底层的网络被划分成多个分层的域。每一个域由一个目录节点表示。



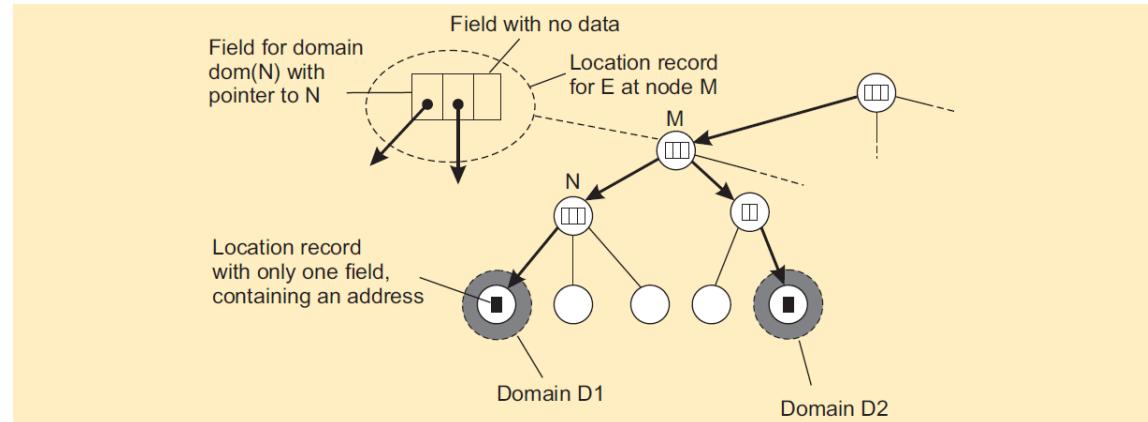


树的组织结构

➤ 不变量

- 实体 E 的地址存储在叶节点或者中间节点上；
- 中间结点包含了指向其孩子节点的指针，当且仅当根植于孩子节点的子树存储有实体的地址；
- 根节点知道所有实体的地址；

➤ 如果一个实体有两个地址位于不同的叶子域时





树结构的查询操作

➤ 基本原理

- 开始在一个叶节点上搜索；
- 如果节点知道实体 E => 继续搜索下游指针，否则回退到父节点；
- 向上搜索的过程最终会以到达根节点而停止；

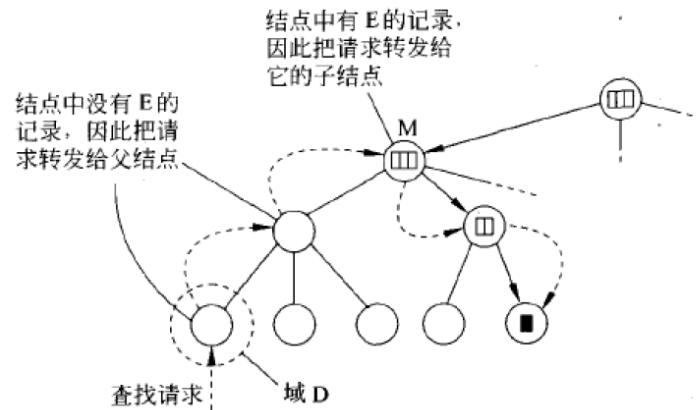
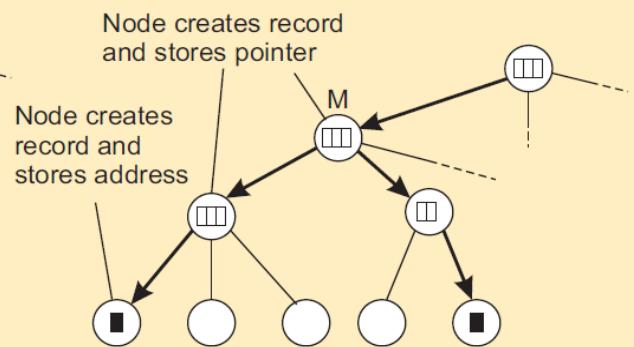
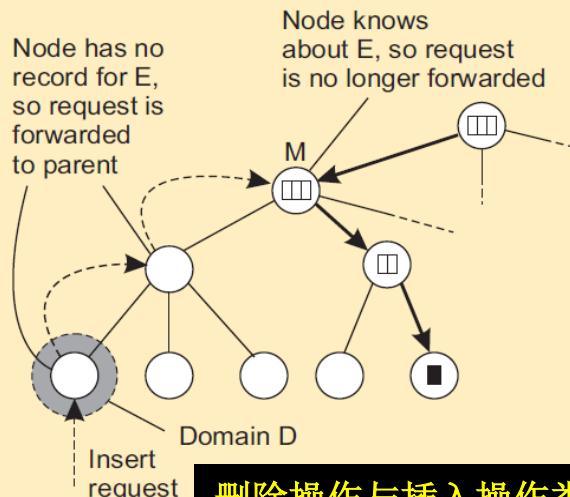


图 5.7 在分层组织的定位服务中的位置查找



树结构的插入操作

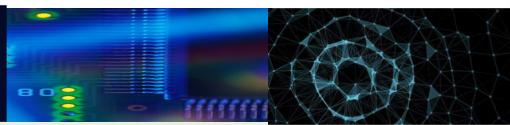
- 插入请求被转发到知道实体 E 地址的第一个节点；
- 建立一条从第一个知道实体 E 地址的节点到实体 E 的指针链；



删除操作与插入操作类似，自底向上删除存储实体 E 的记录

(a)

(b)



结构化命名



结构化命名

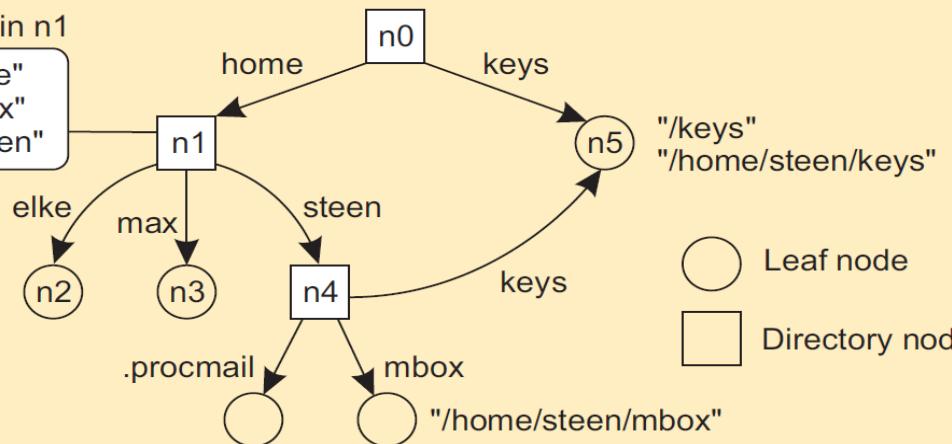
➤ 命名空间

命名图（naming graph）：一张叶节点表示实体的图。此外，叶节点还可以存储实体的属性、状态等信息。目录节点具有一定数量的边，用于引用其他节点。

➤ 一张通用的具有一个根节点的命名图

Data stored in n1

n2: "elke"
n3: "max"
n4: "steen"





命名空间

- 我们很容易在命名图的节点中存储各种属性:
 - 实体的类型;
 - 实体的标示图;
 - 实体的地址信息;
 - 别名;
 -
- 注意
 - 目录节点也可以拥有属性，除了存储目录表以外；



名称解析

➤ 名称解析：

给定一个路径名，查找出存储在由该名称所指向的节点中的任何信息，查询名称的过程称为名称解析。

➤ 问题

为了解析一个名字我们需要一个目录节点，那么该如何找到初始节点？



名称解析-闭包机制 (closure mechanism)

➤ 闭包机制：知道如何启动以及在何处启动名称解析通常称为闭包机制

- www.distributed-systems.net, 从DNS服务器开始;
- /home/maarten/mbox: 从本地的NFS服务器开始;
- 13587569903: 拨打电话号码;
- 222.200.145.180: 把消息路由到特定的IP地址;
-



链接 (Name linking)

➤ 硬链接 (Hard link)

我们所描述的路径名即用于在命名图中按照特定路径搜索节点的名字就是“硬链接”；

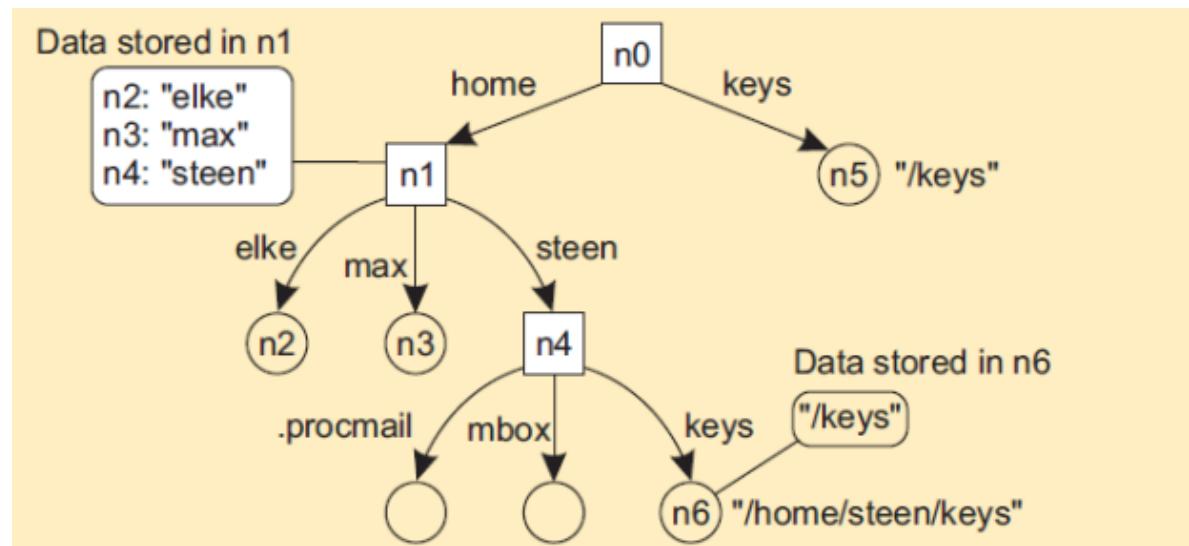
➤ 软链接 (Soft link)：允许一个节点 N 包含另外一个节点名字用叶节点表示实体，而不是存储实体的地址和位置，该节点存储绝对路径名。

- 首先解析 N 的名字；
- 读取 N 的内容返回名字；
- 利用返回的名字进行名字解析；



链接 (Name linking)

- 命名图中的符号链接





挂载

➤ 问题

命名解析也可以应用于合并不同的命名空间，通过挂载的方法透明地实现；将另一个空间的节点标识符与当前命名空间的节点相关联；

➤ 术语

- 外部命名空间：需要访问的命名空间；
- 挂接点：在当前命名空间中用于存储节点标识符的目录节点成为挂接点；
- 挂载点：外部名称空间中的目录节点称为挂载点；挂载点是命名空间的“根”；



挂载

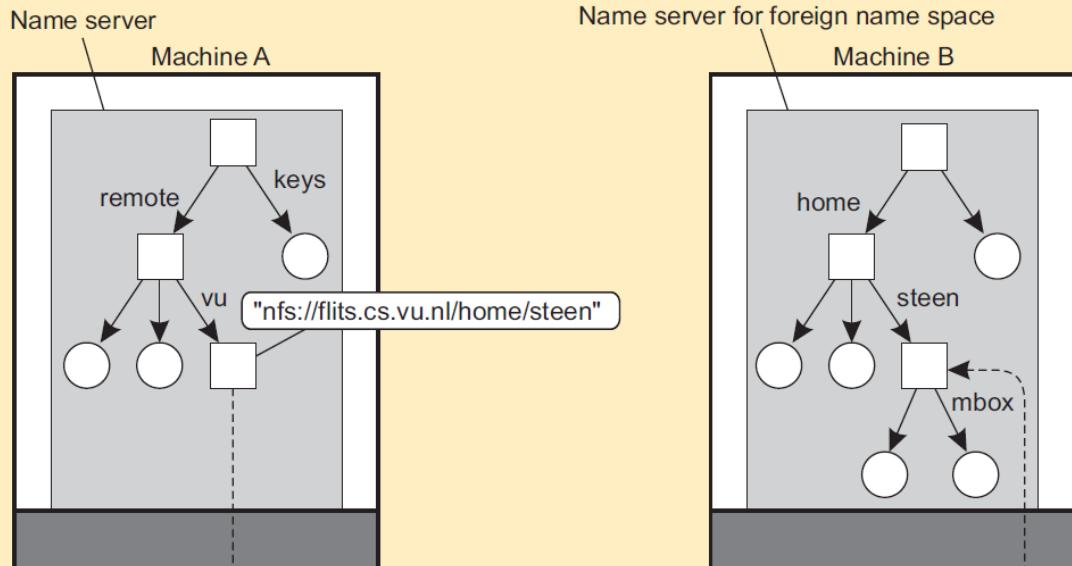
➤ 通过网络挂载

- 访问协议的名称；
- 服务器的名称；
- 外部名称空间中的挂载点的名称；



分布式系统挂载

- 通过特定的访问协议挂载远程的命名空间



NFS文件系统是否可以嵌套挂载?

Reference to foreign name space

Network



命名空间实现

➤ 基本问题

跨越多个机器的分布式命名解析过程与命名空间管理是通过将命名图分布在多个节点上实现的；

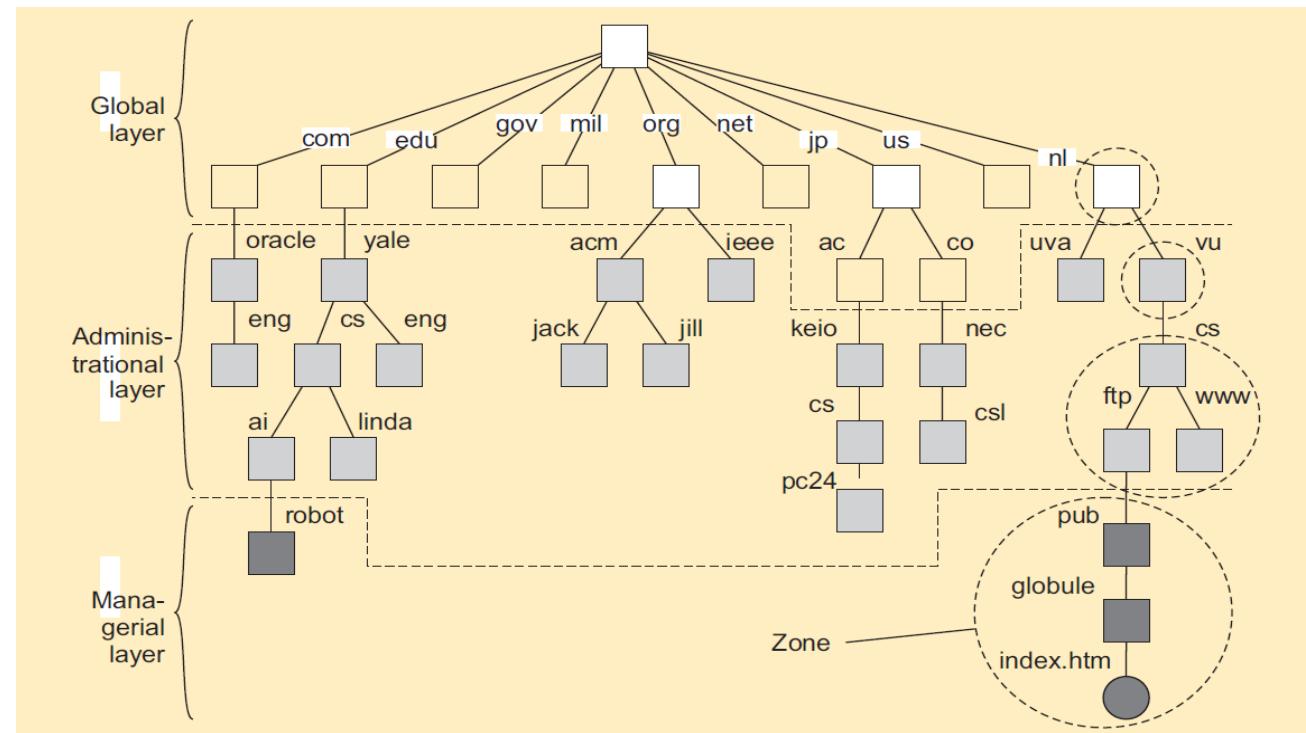
➤ 三层名称空间

- 全局层：由最高级别的节点构成，即由根节点以及其他逻辑上靠近根节点的目录节点组成。特点是：稳定，目录表很少改变，可以代表组织或者组织群。
- 行政层：由那些在单个组织内一起被管理的目录节点组成。行政层中的目录节点所具有的特点是代表属于同一组织或行政单位的实体组；相对稳定，但是比全局层的目录变化频繁；
- 管理层：由经常改变的节点组成。如代表本地主机的节点及本地文件系统等，由终端用户维护；



命名空间实现

➤ DNS命名空间划分示例包括网络文件系统，共分三层





命名空间实现

➤ 可用性问题

- 全局层要求具有很高的可用性；
- 行政层可用性要求最高；
- 管理层的可用性要求不高；

➤ 性能问题

- 使用缓冲机制可以增加全局层命名解析的性能；
- 使用高性能服务器来运行命名服务器；



命名空间实现

- 全局层、行政层和管理层实现节点的名称服务器之间的比较

Item	Global	Administrational	Managerial
1	Worldwide	Organization	Department
2	Few	Many	Vast numbers
3	Seconds	Milliseconds	Immediate
4	Lazy	Immediate	Immediate
5	Many	None or few	None
6	Yes	Yes	Sometimes

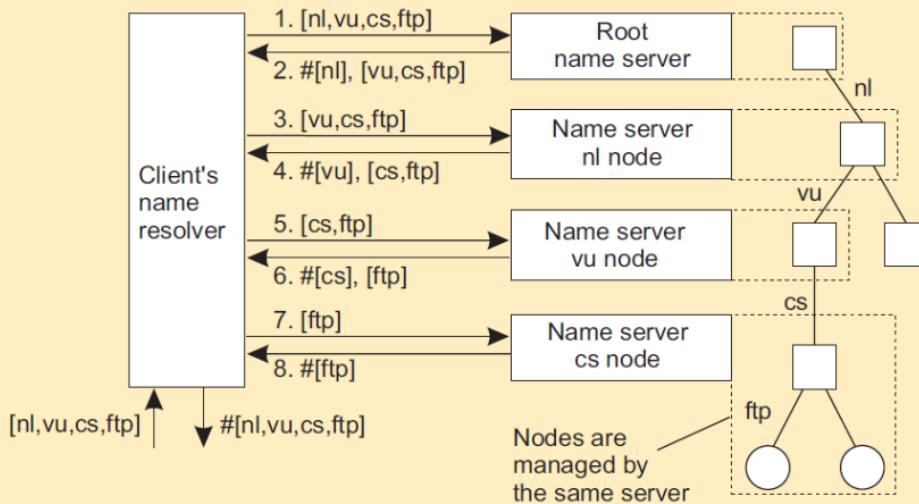
1: Geographical scale	4: Update propagation
2: # Nodes	5: # Replicas
3: Responsiveness	6: Client-side caching?



迭代命名解析

➤ 原理 解析: `ftp://ftp.cs.vu.nl/pub/globe/index.txt`

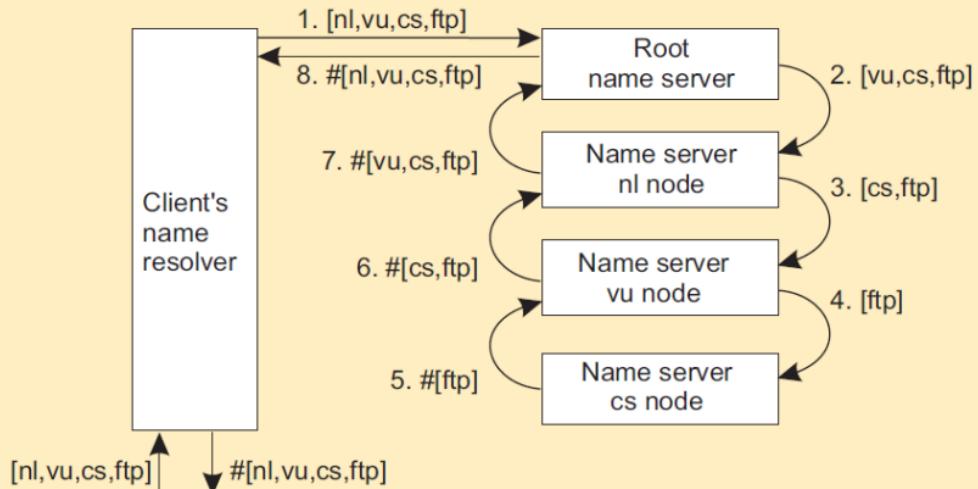
- ① *resolve(dir, [name₁, ..., name_K])* sent to Server₀ responsible for *dir*
- ② Server₀ resolves *resolve(dir, name₁)* → *dir₁*, returning the identification (address) of Server₁, which stores *dir₁*.
- ③ Client sends *resolve(dir₁, [name₂, ..., name_K])* to Server₁, etc.





递归命名解析

- ① $\text{resolve}(\text{dir}, [\text{name}_1, \dots, \text{name}_K])$ sent to Server_0 responsible for dir
- ② Server_0 resolves $\text{resolve}(\text{dir}, \text{name}_1) \rightarrow \text{dir}_1$, and sends $\text{resolve}(\text{dir}_1, [\text{name}_2, \dots, \text{name}_K])$ to Server_1 , which stores dir_1 .
- ③ Server_0 waits for result from Server_1 , and returns it to client.



这种命名解析方式的优缺点？



递归命名解析中的缓存机制

➤ [nl, vu, cs, ftp] 递归解析过程

Server for node	Should resolve	Looks up	Passes to child	Receives and caches	Returns to requester
cs	[ftp]	# [ftp]	—	—	# [ftp]
vu	[cs, ftp]	# [cs]	[ftp]	# [ftp]	# [cs] # [cs, ftp]
nl	[vu, cs, ftp]	# [vu]	[cs, ftp]	# [cs] # [cs, ftp]	# [vu] # [vu, cs] # [vu, cs, ftp]
root	[nl, vu, cs, ftp]	# [nl]	[vu, cs, ftp]	# [vu] # [vu, cs] # [vu, cs, ftp]	# [nl] # [nl, vu] # [nl, vu, cs] # [nl, vu, cs, ftp]



命名解析的扩展性问题

➤ 规模可扩展性

我们需要保证命名服务器每秒钟可处理大量的请求 => 高层次的服务器面临较大的挑战；

➤ 地理可扩展性

假定（至少在全局层和行政层）节点的内容几乎不发生变化。我们可以应用扩展备份将节点的内容映射到多个服务器，从最近的服务器上开始命名解析；

➤ 观察发现

很多节点的一个很重要的属性是代表实体的地址是可以访问的，但是复制节点会让大规模传统的命名服务器不适合用于定位移动实体；



命名解析的扩展性问题

- 在大规模跨地域范围内的命名解析扩展性

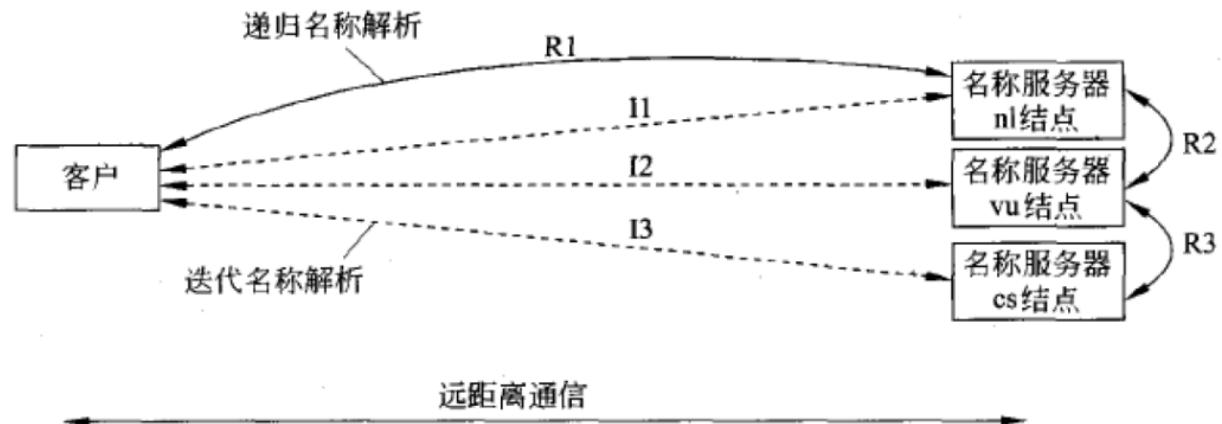


图 5.18 递归名称解析与迭代名称解析在通信开销方面的对比

➤ 问题

通过将节点内容映射到能够在任何地点访问的服务器上，我们引入了隐性的位置依赖性。



DNS 域名解析系统

➤ 本质

- 命名空间被组织成分层结构，每一个节点有一个显式的进入边；
- 域（domain）：子树；
- 域名：指向域根节点的路径名字，可以是绝对的也可以是相对

记录类型	相关实体	描述
SOA	区域	容纳表示区域的信息
A	主机	容纳结点所代表主机的 IP 地址
MX	域	指向邮件服务器，该服务器处理发往该结点的邮件
SRV	域	指向处理某种服务的服务器
NS	区域	指向实现指定区域的名称服务器
CNAME	结点	包含指定结点主名称的符号链接
PTR	主机	容纳主机的规范名称
HINFO	主机	容纳结点所代表主机的信息
TXT	任意类型	容纳任何被认为有用、特定于实体的信息

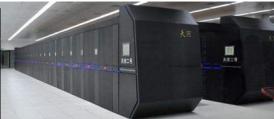


DNS实现

每一个区域都有一个名称服务器来实现。

名称	记录类型	记录值
cs.vu.nl	SOA	star 1999121502,7200,3600,2419200,86400
cs.vu.nl	NS	star.cs.vu.nl
cs.vu.nl	NS	top.cs.vu.nl
cs.vu.nl	NS	solo.cs.vu.nl
cs.vu.nl	TXT	“Vrije Universiteit Math. & Comp. Sc.”
cs.vu.nl	MX	1 zephyr.cs.vu.nl
cs.vu.nl	MX	2 tornado.cs.vu.nl
cs.vu.nl	MX	3 star.cs.vu.nl
star.cs.vu.nl	A	
star.cs.vu.nl	A	
star.cs.vu.nl	MX	
star.cs.vu.nl	MX	
star.cs.vu.nl	HINFO	
zephyr.cs.vu.nl	A	
zephyr.cs.vu.nl	A	
zephyr.cs.vu.nl	MX	
zephyr.cs.vu.nl	MX	
zephyr.cs.vu.nl	HINFO	
ftp.cs.vu.nl	CNAME	
www.cs.vu.nl	CNAME	

非集中式DNS实现



基于属性的命名

➤ 观察

- 在很多场景中，通过实体的属性查询实体要方便的多，如传统的目录服务（黄页）；

➤ 问题

- 查找操作的代价很高，因为需要匹配请求的属性值 => 理论上需要检查所有的实体；



目录服务

- 搜索可扩展性方案
- 利用数据库实现基本的目录服务，并结合传统的结构化的命名系统；
- 轻量级的目录访问协议（**LDAP**）
- 每一个目录项包含（属性，值）对，并且被赋予唯一的名字方便查找。

Attribute	Abbr.	Value
Country	C	NL
Locality	L	Amsterdam
Organization	O	VU University
OrganizationalUnit	OU	Computer Science
CommonName	CN	Main server
Mail_Servers	-	137.37.20.3, 130.37.24.6, 137.37.20.10
FTP_Server	-	130.37.20.20
WWW_Server	-	130.37.20.20

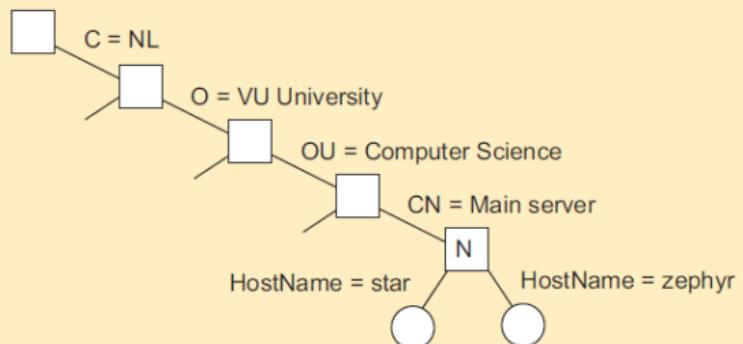


LDAP

Essence

- **Directory Information Base:** collection of all directory entries in an LDAP service.
- Each record is uniquely named as a sequence of naming attributes (called **Relative Distinguished Name**), so that it can be looked up.
- **Directory Information Tree:** the naming graph of an LDAP directory service; each node represents a directory entry.

Part of a directory information tree



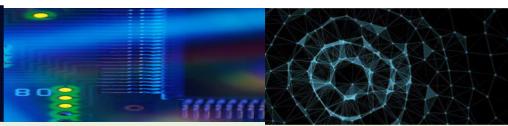


LDAP

- 以RDN作为主机名的两个目录项

Attribute	Value	Attribute	Value
Locality	Amsterdam	Locality	Amsterdam
Organization	VU University	Organization	VU University
OrganizationalUnit	Computer Science	OrganizationalUnit	Computer Science
CommonName	Main server	CommonName	Main server
HostName	star	HostName	zephyr
HostAddress	192.31.231.42	HostAddress	137.37.20.10

Result of search(``(C=NL) (O=VU University) (OU=*) (CN>Main server)``)



谢谢！