Advanced Operating Systems

Chapter 3: **Processes**

Reihaneh khorsand

Contents

- Threads
- Multithreaded Clients
- Multithreaded Servers
- Code Migration

THREADS

A **process** is generally defined as **a program in execution**.

Having a form of multiple threads of control per process makes it much easier to build distributed applications and to attain better performance.

Threads *are often provided in the form of a thread package.*

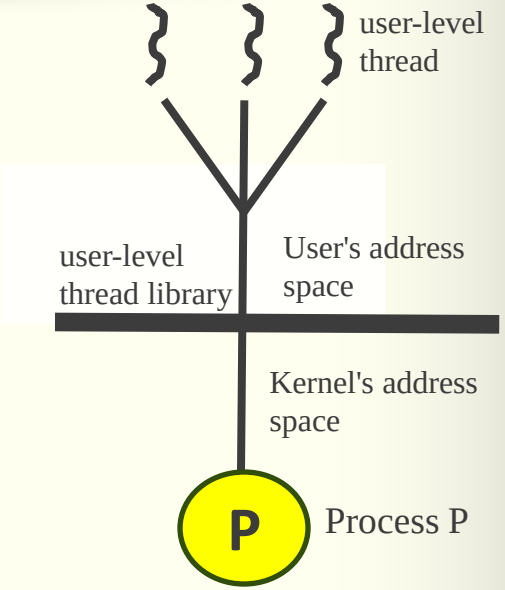
Such a package contains operations to create and destroy threads as well as operations on synchronization variables such as condition variables.

There are basically **two approaches** to **implement a thread package:**

1. The first approach is to construct a thread library that is executed entirely in user mode.
2. The second approach is to have the kernel be aware of threads and schedule them.

ULT s(User Level Threads)

Advantages



✓ It is cheap to create and destroy threads.
Because all thread administration is kept in the user's address space

✓switching thread context can often be done in just a few instructions.

✓Particular scheduling

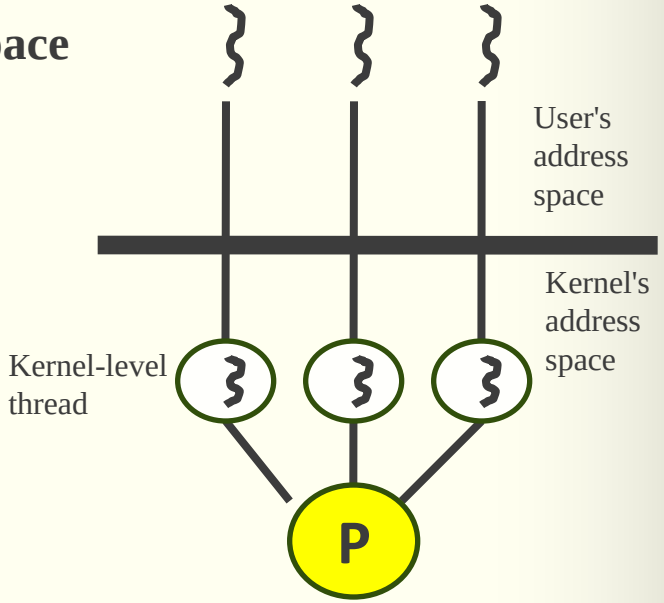
✓User-level threads can execute on every OS

Disadvantages

invocation of a blocking system call will immediately block the entire process to which the thread belongs, and thus also all the other threads in that process.

KLTs (Kernel Level Threads)

All thread administration is kept in the kernel's address space



Advantages

Invocation of a system call is non blocking

Disadvantages

- There is a high price to pay: every thread operation (creation, deletion, synchronization, etc.), will have to be carried out by the kernel. requiring a system call.
- Switching thread contexts may now become as expensive as switching process contexts.

A hybrid form of user-level and kernel-level threads: lightweight processes (LWP)

An LWP runs in the context of a single (heavy-weight) process, and there can be several LWPs per process.

In addition to having LWPs, a system also offers a user-level thread package.

Offering applications the usual operations for creating and destroying threads.
In addition, the package provides facilities for thread synchronization. such as condition variables.

The **important issue** is that the thread package is implemented entirely in user space.
In other words. all operations on threads are carried out without intervention of the kernel.

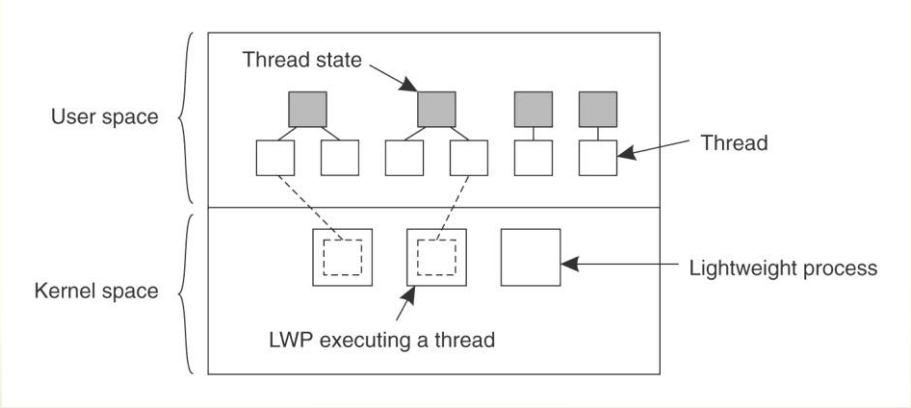


Fig 2: Combining kernel-level lightweight processes and user-level threads

lightweight processes (LWP) (continue)

Each LWP can be running its own (user-level) thread.

Multithreaded applications are constructed by creating threads, and subsequently assigning each thread to an LWP.

Assigning a thread to an LWP is normally implicit and hidden from the programmer.

When creating an LWP (which is done by means of a system call), the LWP is given its own stack, and is instructed to execute the scheduling routine in search of a thread to execute.

If there are several LWPs, then each of them executes the scheduler.
The **thread table**, which is used to keep track of the current set of threads, is thus shared by the LWPs.

Protecting this table to guarantee **mutually exclusive** access is done by means of mutexes that are implemented entirely in user space.

lightweight processes (LWP) (continue)

In other words, synchronization between LWPs does not require any kernel support.

When an LWP finds a runnable thread, it switches context to that thread.

Meanwhile, other LWPs may be looking for other runnable threads as well.

If a thread needs to block on a mutex or condition variable, it does the necessary administration and eventually calls the **scheduling routine**.

'When another runnable thread has been found, a context switch is made to that thread.

The beauty of all this is that the LWP executing the thread need not be informed: the context switch is implemented completely in user space and appears to the LWP as normal program code.

Several advantages to using LWPs in combination with a user-level thread package:

1. Creating, destroying, and synchronizing threads is relatively cheap and involves no kernel intervention at all.
2. Provided that a process has enough LWPs, a blocking system call will not suspend the entire process.
3. There is no need for an application to know about the LWPs. All it sees are user-level threads.
4. LWPs can be easily used in multiprocessing environments, by executing different LWPs on different CPUs. This multiprocessing can be hidden entirely from the application.

The **only drawback** of lightweight processes in combination with user-level threads is that we still need to create and destroy LWPs, which is just as expensive as with kernel-level threads.

However, creating and destroying LWPs needs to be done only occasionally, and is often fully controlled by the operating system.

Multithreaded Clients

Multithreaded Clients → To establish a high degree of distribution transparency

a Web document consists of an HTML file containing plain text along with a collection of images, icons, etc.

To fetch each element of a Web document, the browser has to set up a TCP/IP connection, read the incoming data, and pass it to a display component.

Setting up a connection as well as reading incoming data are inherently **blocking operations.**

Developing the **browser** as a **multithreaded client** simplifies matters considerably.

Multithreaded Clients (continue)

As soon as the main HTML file has been fetched, separate threads can be activated to take care of fetching the other parts.

Each thread sets up a separate connection to the server and pulls in the data.

Setting up a connection and reading data from the server can be programmed using the standard (blocking) system calls, assuming that a blocking call does not suspend the entire process.

Multithreaded Clients (continue)

There is another important benefit to using multithreaded Web browsers in which several connections can be opened simultaneously.

In the previous example, several connections were set up to the same server.

If that **server** is **heavily loaded**, or just plain slow

Web servers have been **replicated** across multiple machines, where each server provides exactly the same set of Web documents.

Multithreaded Clients (continue)

The replicated servers are located at the same site, and are known under the same name.

When a request for a Web page comes in, the request is forwarded to one of the servers, often using a round-robin strategy or some other load-balancing technique.

When using a multithreaded client, connections may be set up to different replicas, allowing data to be transferred in parallel, effectively establishing that the entire Web document is fully displayed in a much shorter time than with a no replicated server.

Multithreaded Servers

Consider the organization of a file server that occasionally has to block waiting for the disk.

The file server normally waits for an incoming request for a file operation, subsequently carries out the request, and then sends back the reply.

Here one thread, the **dispatcher**, reads incoming requests for a file operation.

The requests are sent by clients to a well-known **end point** for this server.

After examining the request, the server chooses an idle (i.e., blocked) **worker** thread and hands it the request.

Multithreaded Servers

The **worker** proceeds by performing a blocking read on the *local file system,* which may cause the thread to be suspended until the data are fetched from disk.

If the thread is suspended, another thread is selected to be executed.

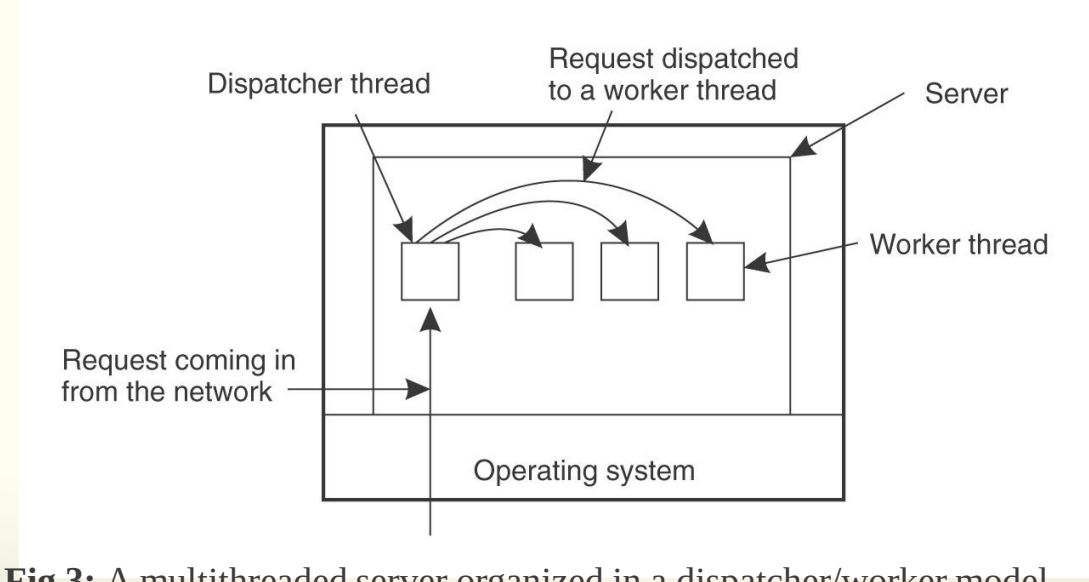


Fig 3: A multithreaded server organized in a dispatcher/worker model.

CODE MIGRATION

Traditionally, code migration in distributed systems took place in the form of process migration in which an entire process was moved from one machine to another

Reasons for Migrating Code:

✓Performance

✓Flexibility

Reasons for Migrating Code/Performance

The basic idea is that overall system performance can be improved if processes are moved from **heavily-loaded** to **lightly-loaded** machines.

Load is often expressed in terms of the CPU queue length or CPU utilization.

A client-server system in which the server manages a huge database.
If a client application needs to perform many database operations involving large quantities of data, it may be better to ship part of the client application to the server and send only the results across the network.

In this case, code migration is based on the assumption that it generally makes sense **to process data close to where those data reside**.

Reasons for Migrating Code/Performance

This same reason can be used for migrating parts of the server to the client.

In many interactive database applications, clients need to fill in forms that are subsequently translated into a series of database operations.

Processing the form at the client side, and sending only the completed form to the server, can sometimes avoid that a relatively large number of small messages need to cross the network.

The result is that the client perceives better performance, while at the same time the server spends less time on form processing and communication.

Reasons for Migrating Code/Performance

Support for code migration can also help improve performance **by exploiting parallelism**, but without the usual intricacies related to parallel programming.

A typical example is searching for information in the Web.

It is relatively simple to implement a search query in the form of a small mobile program, called a mobile agent, that moves from site to site.

By making several copies of such a program, and sending each off to different sites, we may be able to achieve a linear speedup compared to using just a single program instance.

Reasons for Migrating Code/Flexibility

If code can move between different machines, it becomes possible to dynamically configure distributed systems.

For example;

Suppose a server implements a standardized interface to a file system.
To allow remote clients to access the file system, the server makes use of a proprietary protocol.

Normally, the client-side implementation of the file system interface, which is based on that protocol, would need to be linked with the client application.

This approach requires that the software be readily available to the client at the time the client application is being developed.

Reasons for Migrating Code/Flexibility

An alternative is to let the server provide the client's implementation no sooner than is strictly necessary, that is, when the client binds to the server.

At that point, the client dynamically downloads the implementation, goes through the necessary initialization steps, and subsequently invokes the server.

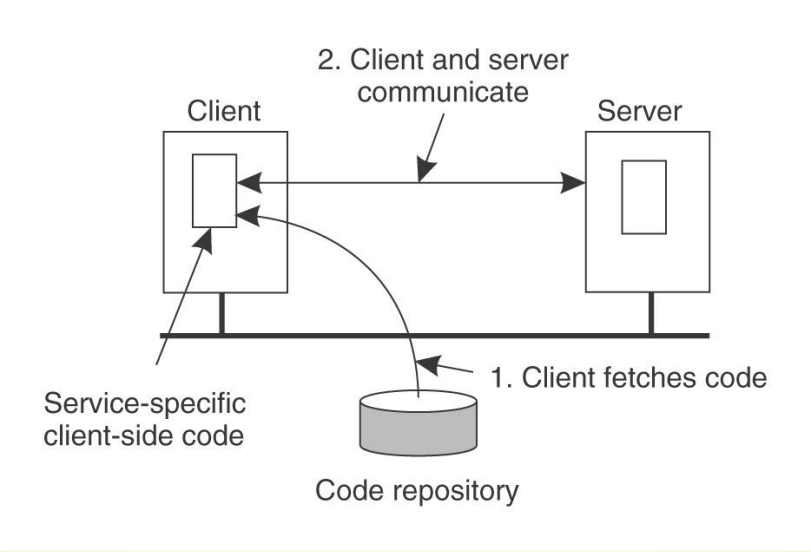


Fig 17: The principle of dynamically configuring a client to communicate to a server. The client first fetches the necessary software, and then invokes the server.

Reasons for Migrating Code/Flexibility

This model of dynamically moving code from a remote site does require that the protocol for downloading and initializing **code** is standardized.

Clients need not have all the software preinstalled to talk to servers.

Instead, the software can be moved in as necessary, and likewise, discarded when no longer needed.

Another advantage is that as long as interfaces are standardized, we can change the client-server protocol and its implementation as often as we like.

Changes will not affect existing client applications that rely on the server.

Disadvantages: Low security

Code migration types

Exchanging **data** between processes

Moving **programs** between machines, with the intention to have those programs be executed at the target

In **process** migration, the execution status of a program, pending signals, and other parts of the environment must be moved as well

A process consists of three segments.

Code segment :

The *code segment is the part that contains the set of instructions* that make up the program that is being executed.

Resource segment :

The *resource segment* contains references to external resources needed. by the process, such as files, printers, devices, other processes, and so on.

Execution segment:

an *execution segment is used* to store the current execution state of a process, consisting of private data, the stack, and, of course, the program counter.

Models for Code Migration

Weak mobility:

In this model, it is possible to transfer only the code segment, along with perhaps some initialization data.

A characteristic feature of weak mobility is that a transferred program is always started from one of several predefined starting positions.

This is what happens, for example, with Java applets, which always start execution from the beginning.

The benefit of this approach is its **simplicity**.

Weak mobility requires only that the target machine can execute that code, which essentially boils down to making the code portable.

Models for Code Migration (continue)

Strong mobility:

in systems that support strong mobility the execution segment can be transferred as well.

The characteristic feature of strong mobility is that a running process can be stopped, subsequently moved to another machine, and then resume execution where it left off.

Clearly, strong mobility is much more general than weak mobility, but also much harder to implement.

Migration and Local Resources

What often makes code migration so difficult is that the resource segment cannot always be simply transferred along with the other segments without being changed.

Three types of process-to-resource bindings:

Binding by identifier:

The **strongest binding** is when a process refers to a resource by its identifier.

In that case, the process requires precisely the referenced resource, and nothing else.

An **example** of such a binding by identifier is when a process uses a URL to refer to a specific Web site or when it refers to an FTP server by means of that server's Internet address.

Three types of process-to-resource bindings:

Binding by value:

A weaker form of process-to-resource binding is when only the value of a resource is needed.

In that case, the execution of the process would not be affected if another resource would provide that same value.

A typical **example** of binding by value is when a program relies on standard libraries, such as those for programming in C or Java.

Such libraries should always be locally available, but their exact location in the local file system may differ between sites.

Not the specific files, but their content is important for the proper execution of the process.

Three types of process-to-resource bindings:

Binding by types:

The **weakest form** of binding is when a process indicates it needs only a resource of a specific type.

This binding by type is exemplified by references to local devices, such as monitors, printers, and so on.

Three types of resource-to-machine bindings

Unattached resources:

They can be easily moved between different machines, and are typically (data) files associated only with the program that is to be migrated

Fastened resources:

Moving or copying a fastened resource may be possible, but only at relatively high costs. Typical examples of fastened resources are local databases and complete Web sites.

Fixed resources:

Fixed resources are intimately bound to a specific machine or environment and cannot be moved. Fixed resources are often local devices.
Another example of a fixed resource is a local communication end point

Self study

- Migration in Heterogeneous Systems

End of Chapter 3

Advanced Operating Systems

Chapter 4: **NAMING**

Contents

Names, Identifiers, and Addresses

- **Name**

A name in a distributed system is a string of bits or characters used to refer to an entity.

- **Entities:**

An entity in a distributed system can be practically anything

For example: hosts, printers, disks, files, processes, users, mailboxes, web pages, graphical windows, messages, network connections, etc.

Names, Identifiers, and Addresses

Entities can be operated on. To operate on an entity, it is necessary to access it, for which we need an **access point** (address).

An entity can offer more than one access point.

As a comparison, a telephone can be viewed as an access point of a person, whereas the telephone number corresponds to an address.

In a distributed system, a typical example of an access point is a host running a specific server, with its address formed by the combination of, for example, an IP address and port number

An address is just a special kind of name

Use the address as a name:

An entity may change its access points in the course of time.

For example when a mobile computer moves to another location, it is often assigned a different IP address than the one it had before.

So, **If an address is used to refer to an entity**, we will have an invalid reference the instant the access point changes or is reassigned to another entity.

if **an entity offers more than one access point**, it is not clear which address to use as a reference.

A name for an entity that is independent from its addresses is often much easier and more flexible to use.

Names, Identifiers, and Addresses

Identifier

A true identifier is a name that is used to uniquely identify an entity

Properties of a true identifier:

- An identifier refers to at most one entity.
- Each entity is referred to by at most one identifier.
- An identifier always refers to the same entity (i.e., it is never reused)

By using identifiers, it becomes much easier to unambiguously refer to an entity.

Types of Naming Systems

Flat naming: The identifier is simply a random bit string.

It does not contain any information whatsoever on how to locate an access point of its associated entity.

Structured naming: Composed of simple human-readable names.

Examples are file system naming and host naming on the Internet.

Attribute-based naming: Allows an entity to be described by (attribute, value) pairs. This allows a user to search more effectively by constraining some of the attributes.

Flat Naming Systems

- Broadcasting and multicasting
- Forwarding pointers
- Home-based approaches
- Distributed hash tables
- Hierarchical approaches

Broadcasting/Multicasting

- Consider a LAN that offers efficient ***broadcasting*** facility.

A message containing the identity of the entity is broadcast to each machine.

Only the machines that can offer access to that entity send a reply containing the address of the access point. For example: ARP (Address Resolution Protocol)

Problem:
- Broadcasting is not suitable for larger networks
- Bandwidth is wasted

- A more efficient approach for larger networks is ***multicasting***, by which only a restricted group of machines receive the request.

For example: data-link level in Ethernet networks

Forwarding Pointers

- When an entity moves from A to B, it leaves behind in A a reference to its new location at B.

- *The main advantage of this approach is its simplicity:*

as soon as an entity has been located using a traditional naming service, a client can look up the current address by following the chain of forwarding pointers.

Example: Remote objects that can move from host to host.

Forwarding Pointers

- A server stub contains either a local reference to the actual object or a local reference to a remote client stub for that object.
- Whenever an object moves from address A to B, it leaves behind a client stub in its place on A and installs a server stub that refers to it in B.

This makes the migration completely transparent to a client.

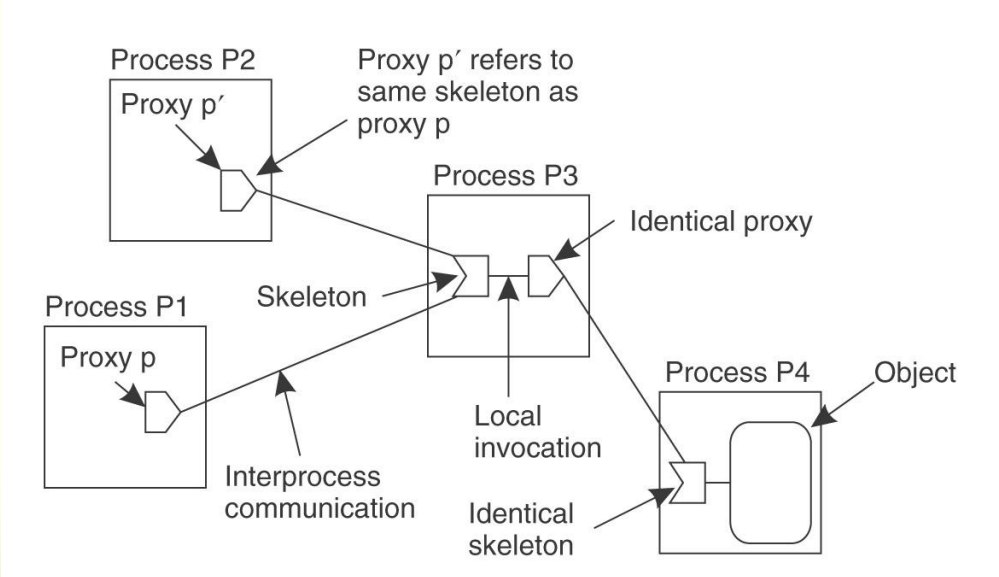


Fig 18: The principle of forwarding pointers using (client stub, server stub) pairs

Forwarding Pointers (important drawbacks)

Problems:

First, if no special measures are taken, a chain for a highly mobile entity can become so long that locating that entity is prohibitively expensive.

Second, all intermediate locations in a chain will have to maintain their part of the chain of forwarding pointers as long as needed.

A **third** (and related) drawback is the vulnerability to broken links. As soon as any forwarding pointer is lost (for whatever reason) the entity can no longer be reached.

It is important to keep the forwarding chains relatively short and to ensure that the forwarding pointers are robust.

Home-Based Approach

A popular approach to supporting mobile entities in large-scale networks is to introduce a **home location**, which keeps track of the current location an entity.

In practice, the home location is often chosen to be the place where an entity was created.

The home-based approach is used as a fall-back mechanism for location services based on forwarding pointers.

Home-Based Approach

An example where the home-based approach is followed is in Mobile IP.

Each mobile host uses a fixed IP address.
All communication to that IP address is initially directed to the mobile host's home agent.

This home agent is located on the local-area network corresponding to the network address contained in the mobile host's IP address.

Whenever the mobile host moves to another network, it requests a temporary address that it can use for communication.

This care-of address is registered at the home agent.

When the home agent receives a packet for the mobile host, it looks up the host's current location.

Home-Based Approach

If the host is on the current local network, the packet is simply forwarded.

Otherwise, it is tunneled to the host's current location, that is, wrapped as data in an IP packet and sent to the care-of address.

At the same time, the sender of the packet is informed of the host's current location.

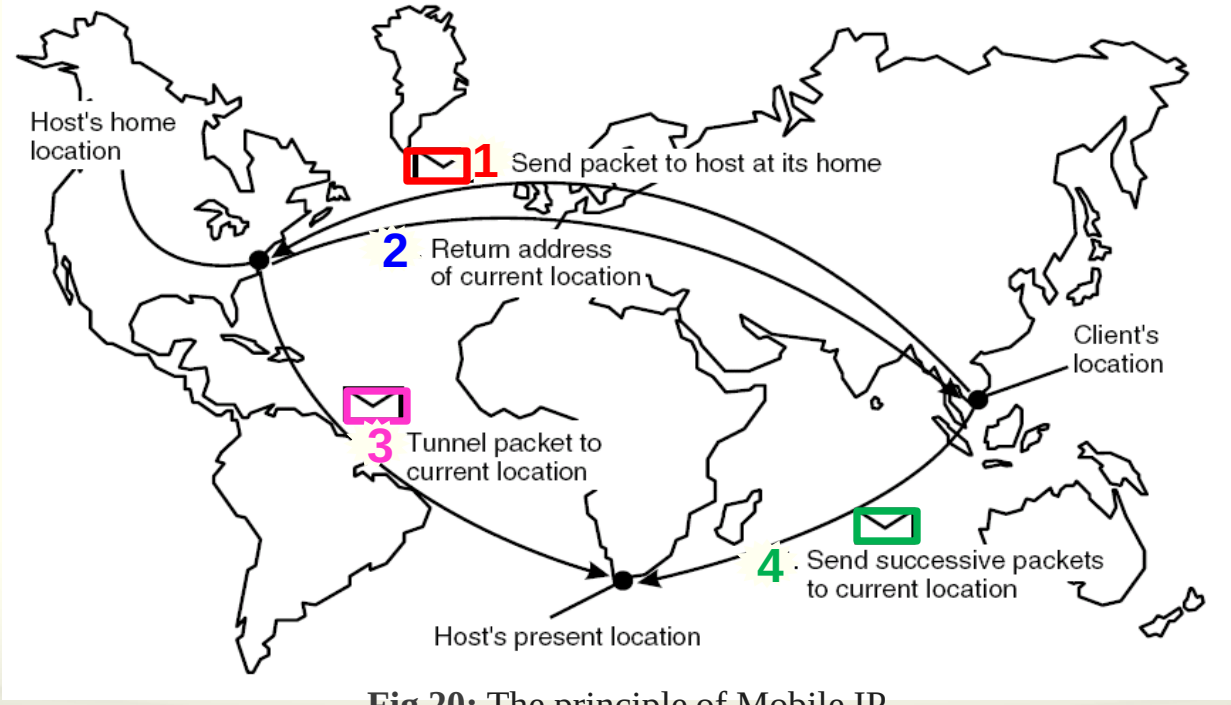


Fig 20: The principle of Mobile IP.

Home-Based Approach (important drawbacks)

To communicate with a mobile entity, a client first has to contact the home, which may be at a completely different location than the entity itself.

The result is an increase in communication latency.

A drawback of the home-based approach is the use of a fixed home location.

For one thing, it must be ensured that the home location always exists. Otherwise, contacting the entity will become impossible.

Distributed Hash Tables

➢A distributed technique on resolving an identifier to the address of an associated entity.

➢Selected applications

- BTDigg: BitTorrent DHT search engine
- Oracle Coherence: An in-memory data grid built on a Java DHT Implementation
- WebSphere eXtreme Scale: proprietary DHT implementation by IBM, used for object caching
- YaCy: Java-based distributed search engine

Distributed Hash Tables

How to efficiently resolve a key k to the address of succ(k)?

- Unbounded binary search: Each node maintains a finger table of at most m entries

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

- The ith entry points to the first node succeeding p by at least 2^{i-1} these are exponentially increasing short-cuts in the identifier space
- To look up a key k, a node p will then forward the request to node q with index j in p's finger table where:

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

- Uses modulo arithmetic

Distributed Hash Tables

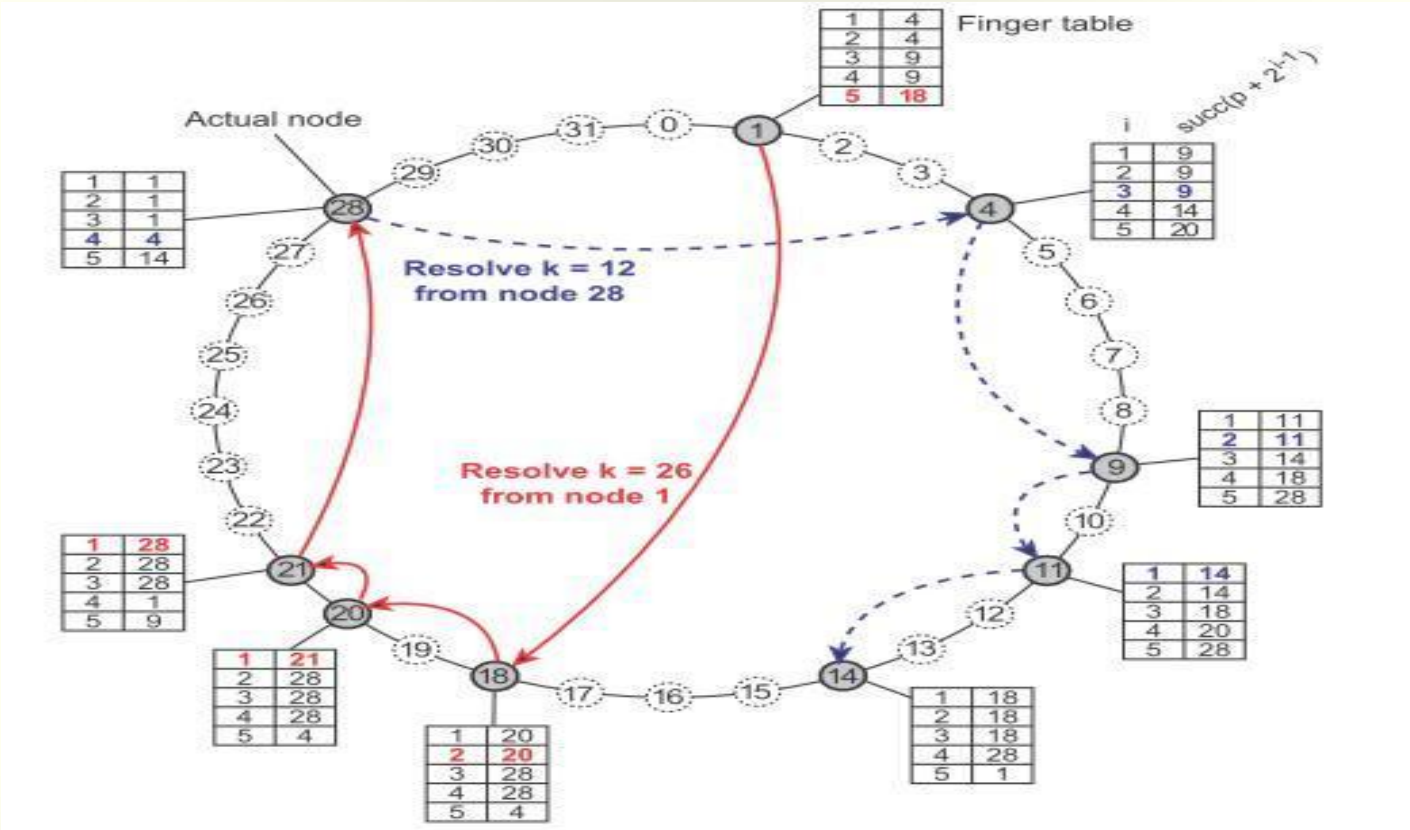


Fig : Resolving key 26 from node 1 and key 12 from node 28 in a Chord system

Hierarchical Approaches

In a hierarchical scheme, **a network is divided into a collection of domains**.

There is a single top-level domain that spans the entire network.

Each domain can be subdivided into multiple, smaller sub domains.

A lowest-level domain, called a **leaf domain**, typically corresponds to a local-area network in a computer network or a cell in a mobile telephone network.

Each domain D has an associated directory node dir(D) that keeps track of the entities in that domain. This leads to a tree of directory nodes.

The directory node of the top-level domain, called the root (directory) node, knows about all entities.

Hierarchical Approaches

The root node will have a location record for each entity, where each location record stores a pointer to the directory node of the next lower-level sub domain where that record's associated entity is currently located.

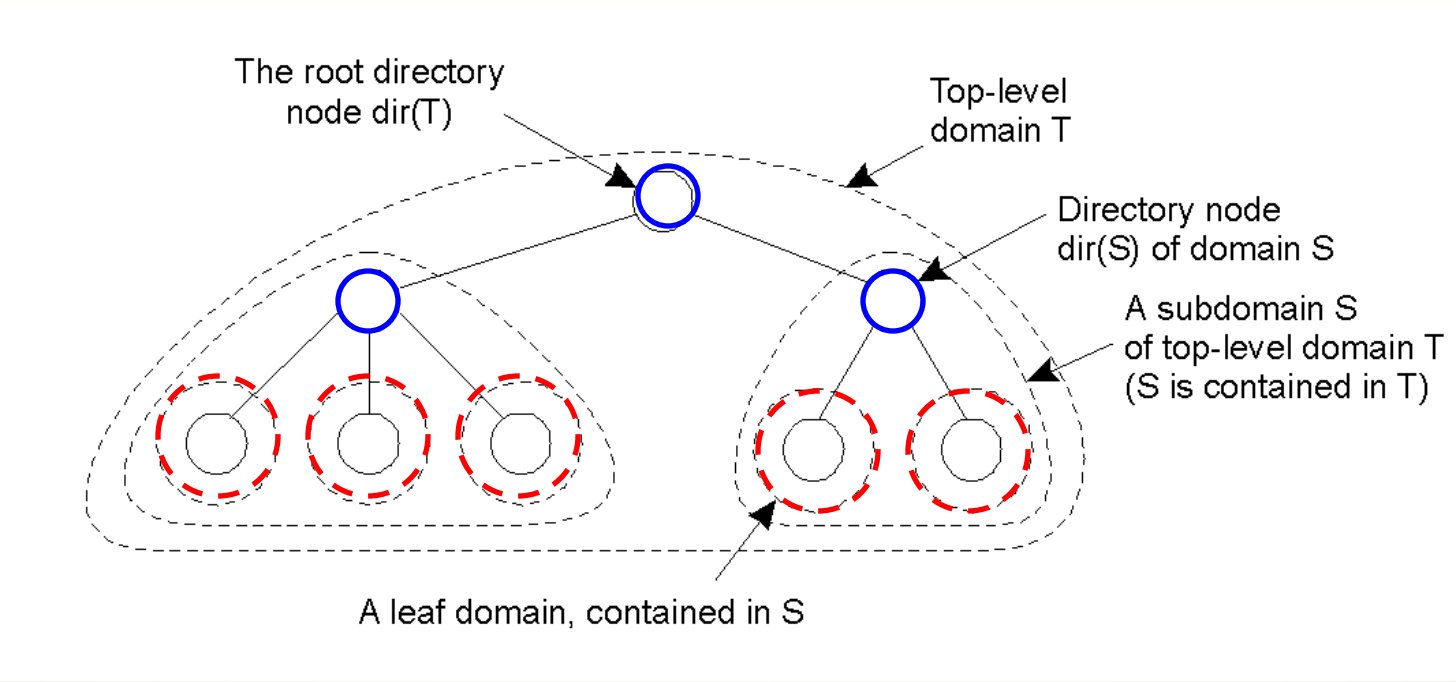


Fig21:Hierarchical organization of a location service into domains, each having an associated directory node.

Hierarchical Approaches

An entity may have multiple addresses, for example if it is replicated.
If an entity has an address in leaf domain *D 1 and D2 respectively, then the directory* node of the smallest domain containing both *D 1 and D 2, will have two pointers* one for each sub domain containing an address.

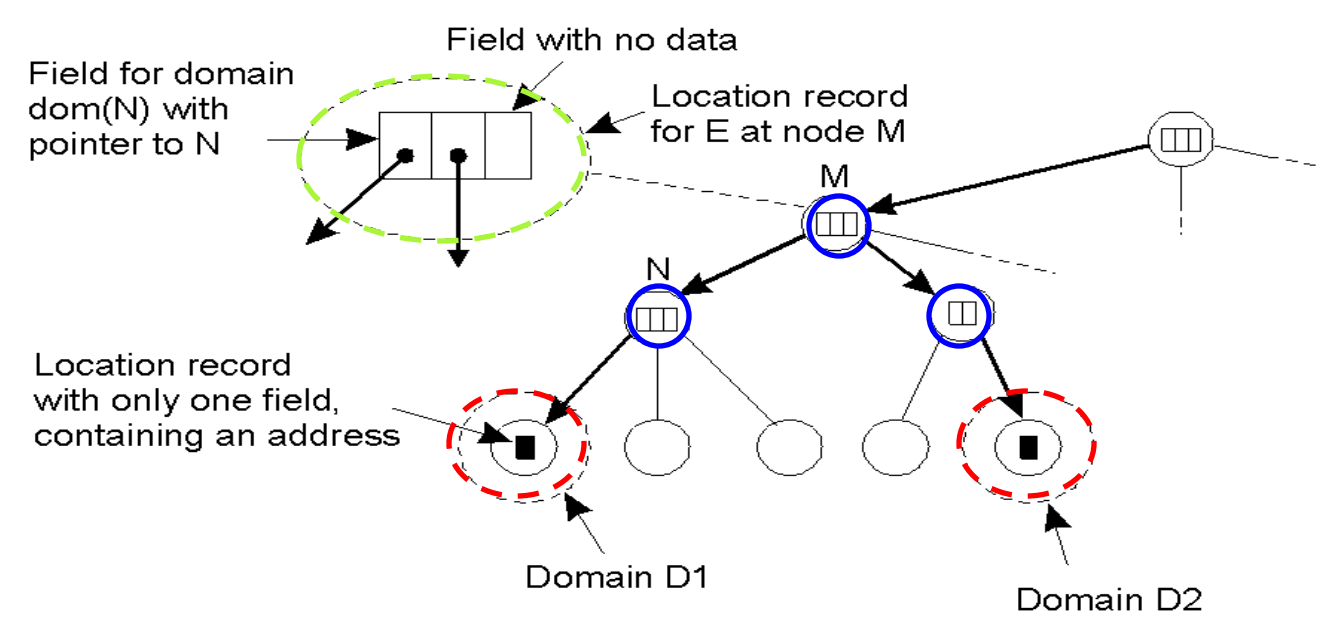


Fig21:An example of storing information of an entity having two addresses in different leaf domains.

Hierarchical Approaches/Looking up a location in a hierarchically organized location service

A client wishing to locate an entity *E,* issues a lookup request to the directory node of the leaf domain D in which the client resides.

If the directory node does not store a location record for the entity, then the entity is currently not located in D.

Consequently, the node forwards the request to its parent.

Note that the parent node represents a larger domain than its child.
If the parent also has no location record for *E, the lookup request is forwarded* to a next level higher, and so on.

As soon as the request reaches a directory node M that stores a location record for entity E, we know that E is somewhere in the domain dom(M) represented by node M.

In figure, M is shown to store a location record containing a pointer to one of its sub domains.

Hierarchical Approaches/Looking up a location in a hierarchically organized location service

The lookup request is then forwarded to the directory node of that sub domain, which in turn forwards it further down the tree, until the request finally reaches a leaf node.

The location record stored in the leaf node will contain the address of *E in that leaf domain.*
This address can then be returned to the client that initially requested the lookup to take place.

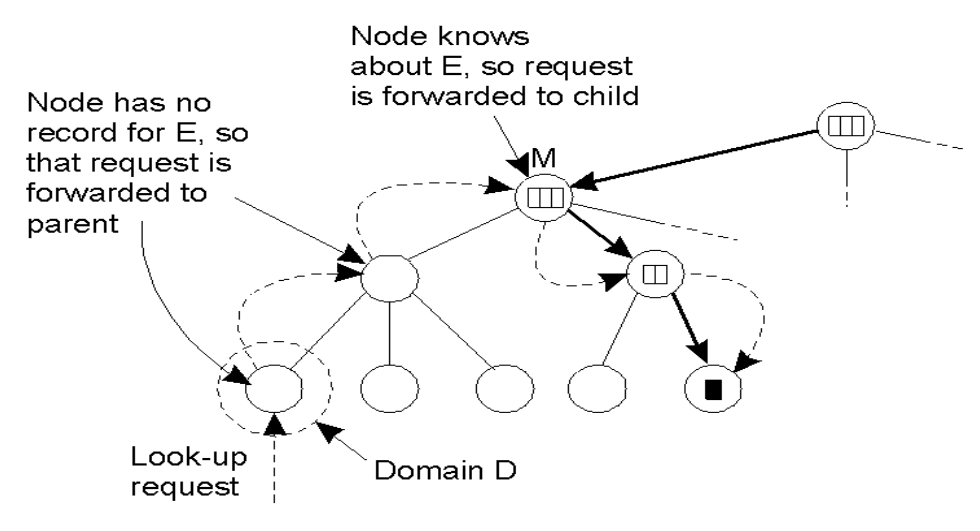


Fig21:Looking up a location in a hierarchically organized location service.

Structured Naming

Flat names are good for machines, but are generally not very convenient for humans to use.

As an alternative, naming systems generally support structured names that are composed from simple, human-readable names.

Not only file naming, but also host naming on the Internet follow this approach.

Name spaces

Names in a distributed system are organized into what is commonly referred to as a **name space.**

A name space can be represented as a labeled, directed graph with two types of nodes.

A **leaf node** represents a named entity and has the property that it has no outgoing edges.

A leaf node generally stores information on the entity it is representing for example its address so that a client can access it.

Alternatively, it can store the state of that entity.

A **directory node** has a number of outgoing edges , each labeled with a name.

A directory node stores a table in which an outgoing edge is represented as a pair (edge label, node identifier). Such a table is called a directory table.

name spaces

Path name N:<label-1, label-2,…, label-n>

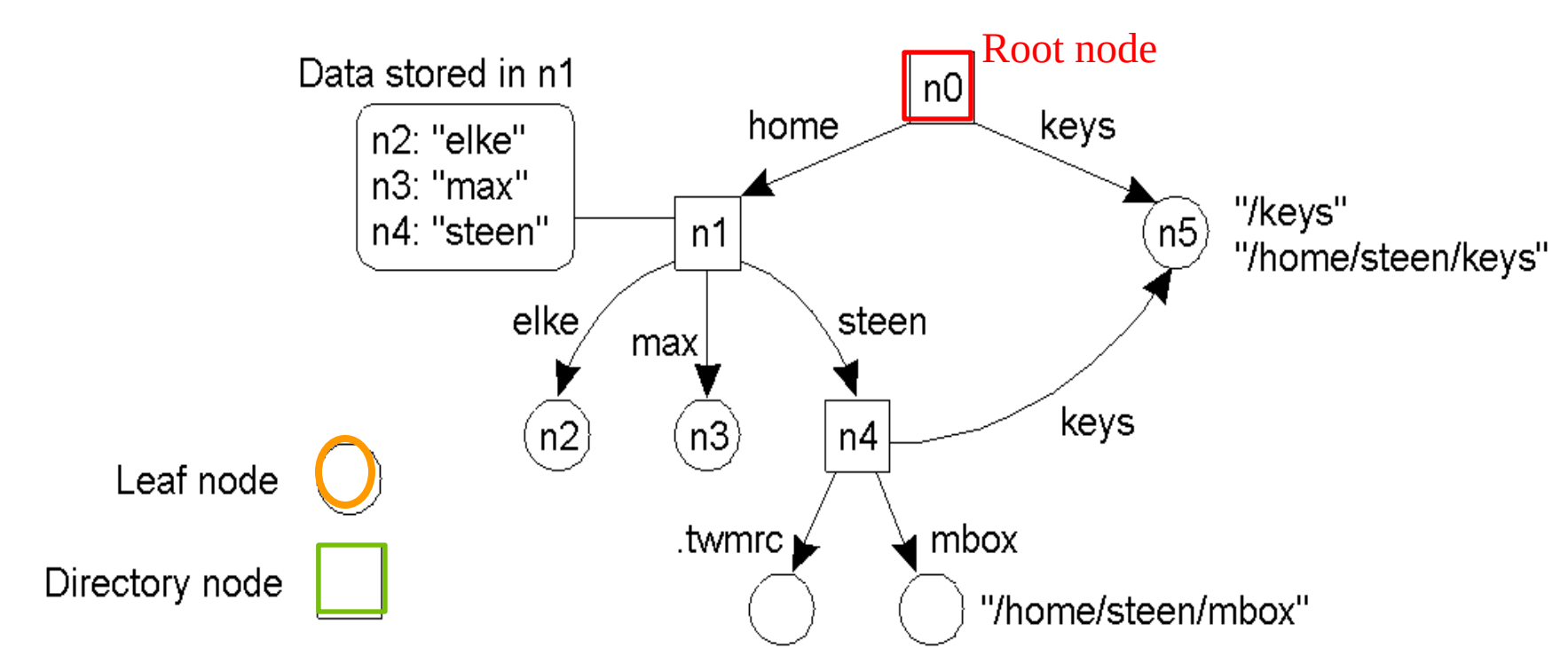


Fig 1: A general naming graph with a single root node

Name Resolution & Closure mechanism

- Name spaces offer a convenient mechanism for storing and retrieving information about entities by means of names.

More generally, given a path name, it should be possible to look up any information stored in the node referred to by that name.

The process of looking up a name is called **name resolution**.

- Knowing how and where to start name resolution is generally referred to as a **closure mechanism.**

Essentially, a closure mechanism deals with selecting the initial node in a name space which name resolution is to start.

Linking and Mounting

Name resolution can be used to merge **different name spaces** in a transparent way.

Let us first consider a mounted file system.
In terms of our naming model, a mounted file system corresponds to letting a directory node store the identifier of a directory node from a different name space, which we refer to as a foreign name space.

- *The directory node storing the node identifier is called a **mount point.***
- *Accordingly, the directory node in the foreign name space is called a **mounting point.***

During name resolution, the mounting point is looked up and resolution proceeds by accessing its directory table.

Linking and Mounting

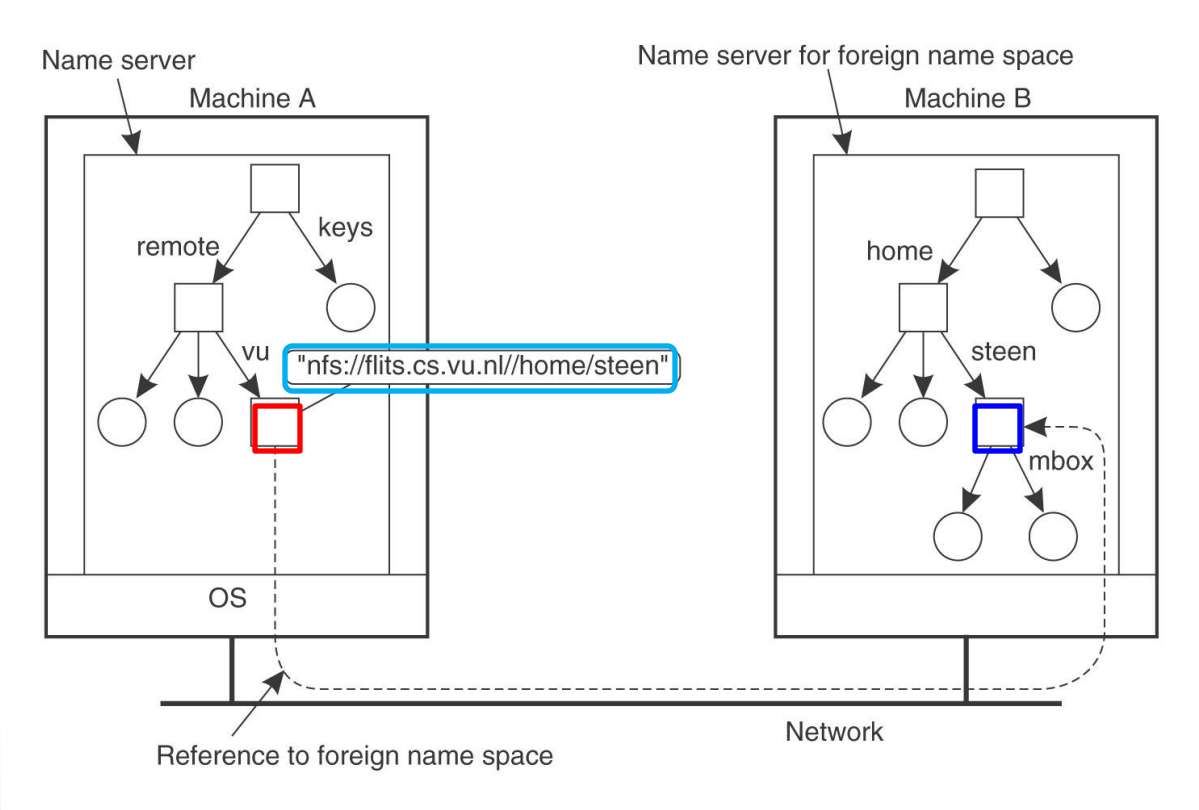


Fig 4: Mounting remote name spaces through a specific access protocol

Linking and Mounting

Consider a collection of name spaces that is distributed across different machines.

In particular, each name space is implemented by a different server, each possibly running on a separate machine.

Consequently, if we want to mount a foreign name space NS 2 into a name space NS 1, it may be necessary to communicate over a network with the server of NS 2, as that server may be running on a different machine than the server for NS1.

To mount a foreign name space in a distributed system requires at least the following information:

1. The name of an access protocol.
2. The name of the server.
3. The name of the mounting point in the foreign.

The Implementation of a Name space

A name space forms the heart of a naming service, that is, a service that allows users and processes to add, remove, and look up names.

A naming service is implemented by name servers.

If a distributed system is restricted to a local area network, it is often feasible to implement a naming service by means of only a single name server.

However, in large-scale distributed systems with many entities , possibly spread across a large geographical area, it is necessary to distribute the implementation of a name space over multiple name servers.

Name Space Distribution

- Global Layer
- Administration Layer
- Manegerial Layer

Name Space Distribution

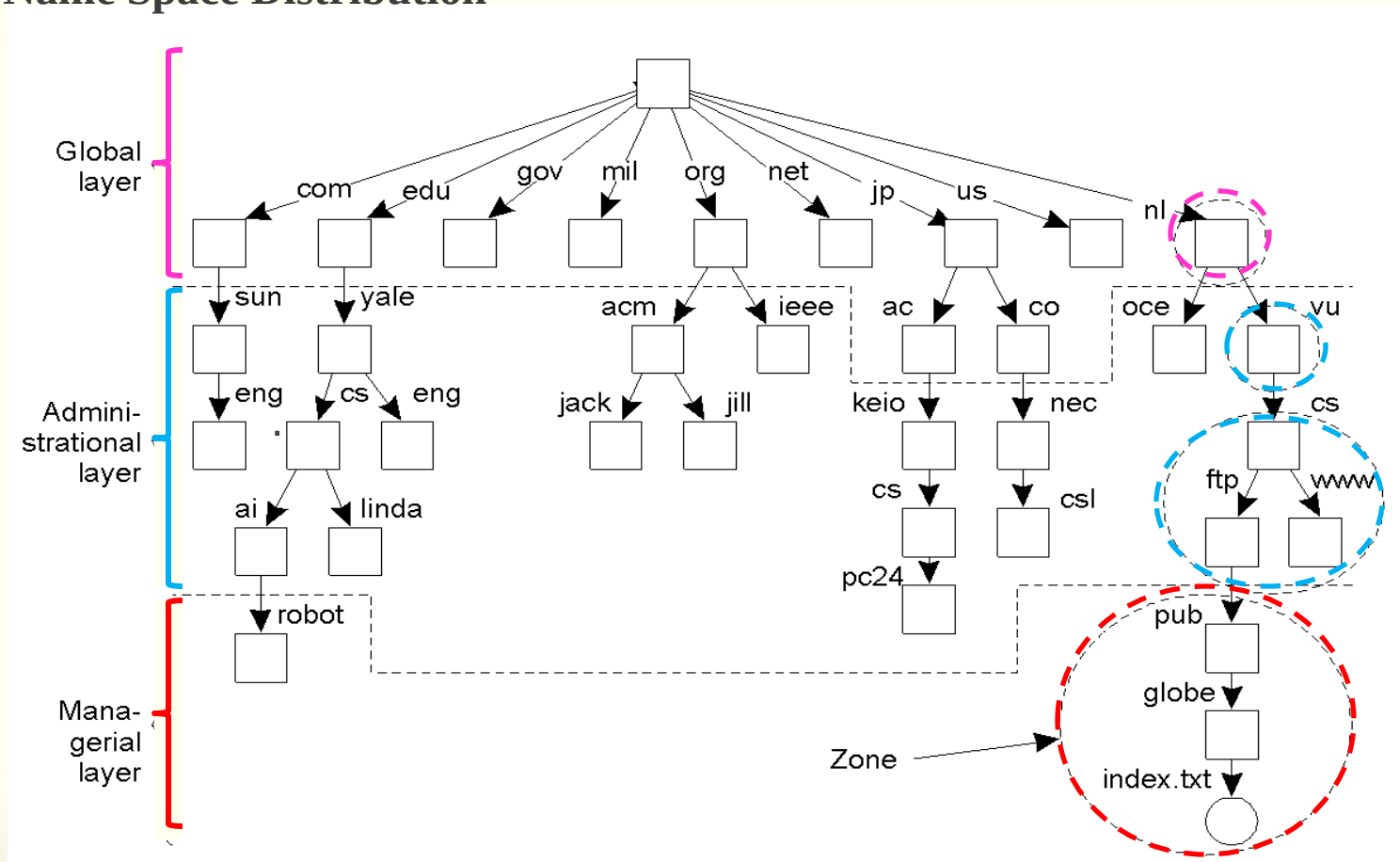


Fig 6: An example partitioning of the DNS name space, including Internet-accessible files, into three layers

Name Space Distribution/ Global layer

Global layer :

The global layer is formed by highest-level nodes, that is, the root node and other directory nodes logically close to the root, namely its children.

Nodes in the global layer are often characterized by their stability, in the sense that directory tables are rarely changed.

Such nodes may represent organizations, or groups of organizations, for which names are stored in the name space.

Name Space Distribution/ Administrational layer

Administrational layer:

The administrational layer is formed by directory nodes that together are managed within a single organization.
A characteristic feature of the directory nodes in the administrational layer is that they represent groups of entities that belong to the same organization or administrational unit.

For example, there may be a directory node for each‘ department in an organization, or a directory node from which all hosts can be found.

Another directory node may be used as the starting point for naming all users, and so forth.

The **nodes** in the administrational layer are **relatively stable**, although changes generally occur more frequently than to nodes in the global layer.

Name Space Distribution/ Managerial layer

Managerial layer:

The managerial layer consists of **nodes that may typically change regularly**.

For example, nodes representing hosts in the local network belong to this layer.

For the same reason, the layer includes nodes representing shared files such as those for libraries or binaries.

Another important class of nodes includes those that represent user-defined directories and files.

In contrast to the global and administrational layer, the nodes in the managerial layer are maintained not only by system administrators, but also by individual end users of a distributed system.

Name Space Distribution/ Comparison

Item	Global	Administrational	Managerial
Geographical scale of network	Worldwide	Organization	Department
Total number of nodes	Few	Many	Vast numbers
Responsiveness to lookups	Seconds	Milliseconds	Immediate
Update propagation	Lazy	Immediate	Immediate
Number of replicas	Many	None or few	None
Is client-side caching applied?	Yes	Yes	Sometimes

Fig 7: A comparison between name servers for implementing nodes from a large-scale name space partitioned into a global layer, an administrational layer, and a managerial layer.

Implementation of Name Resolution

Each client has access to a local name resolver, which is responsible for ensuring that the name resolution process is carried out.

Assume the (absolute) path name root: «nl, VU, CS, ftp, pub, globe, index.html> is to be resolved.

Using a URL notation, this path name would correspond to ftp://ftp.cs. vu.nl/pub/globe/index.html.

- ***There are now two ways to implement name resolution:***
  - **Iterative name resolution**
  - **Recursive name resolution**

Iterative name resolution

In iterative name resolution, a name resolver hands over the complete name to the root name server.
It is assumed that the address where the root server can be contacted is well known.
The root server will resolve the path name as far as it can, and return the result to the client.

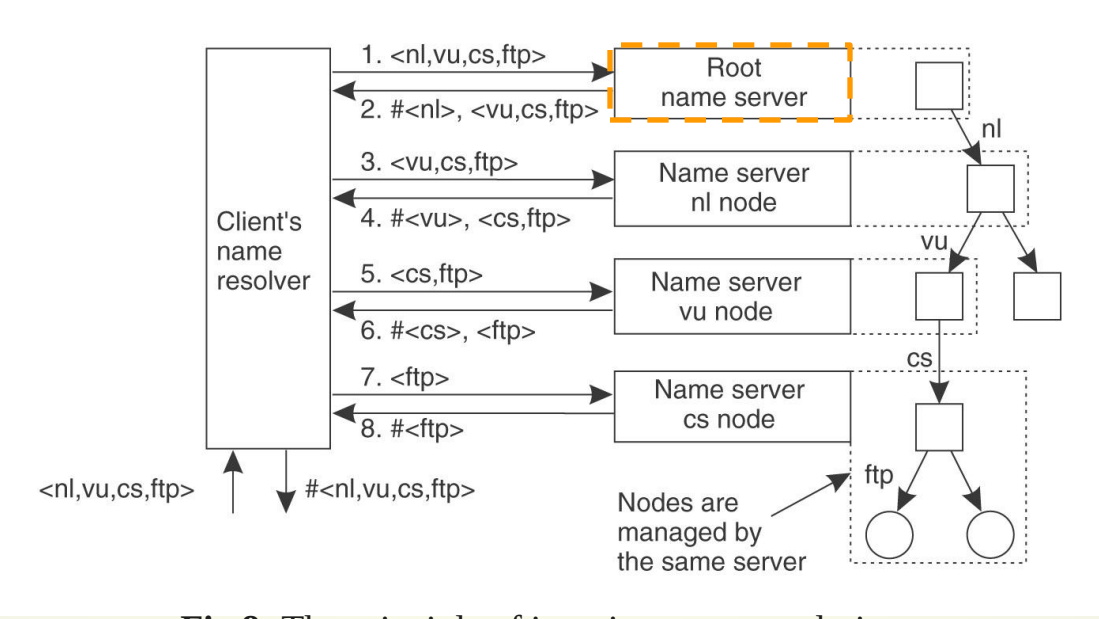


Fig 8: The principle of iterative name resolution.

Recursive name resolution

In recursive name resolution, Instead of returning each intermediate result back to the client's name resolver, a name server passes the result to the next name server it finds.

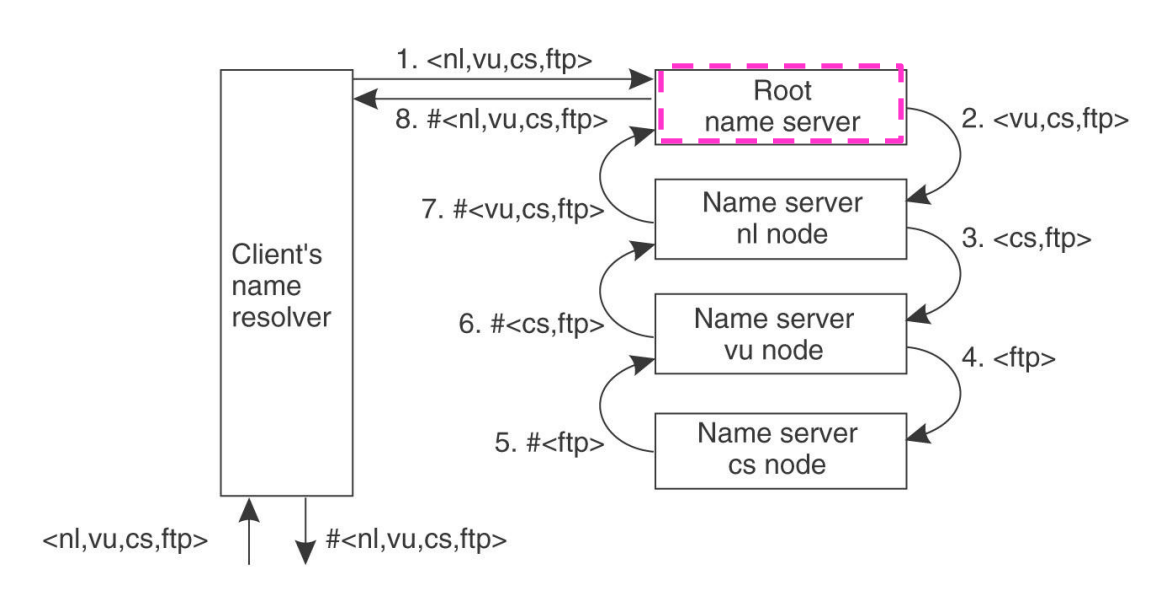


Fig 9: The principle of recursive name resolution.

Recursive name resolution

Drawback

The main drawback of recursive name resolution is that it puts a higher performance demand on each name server.
Basically, a name server is required to handle the complete resolution of a path name, although it may do so in cooperation with other name servers.

This additional burden is generally so high that name servers in the global layer of a name space support only iterative name resolution.

Advantages

The first advantage is that caching results is more effective compared to iterative name resolution.
The second advantage is that communication costs may be reduced.

Attribute-based Naming

Flat and structured names generally provide a unique and location-independent way of referring to entities

as more information is being made available it becomes important to effectively search for entities describe an entity in terms of *(attribute, value)* pairs, generally referred to as attribute-based

Each attribute says something about entity.

By specifying which values a specific attribute should have, a user essentially constrains the set of entities that he is interested in Attribute-based naming systems are also known as directory services

With directory services, entities have a set of associated attributes that can be used for searching.

End of Chapter 4

Advanced Operating Systems

Chapter 5: Synchronization

Reihaneh khorsand

Contents

- CLOCK SYNCHRONIZATION
- LOGICAL CLOCKS
- GLOBAL STATE
- ELECTION ALGORITHMS
- MUTUAL EXCLUSION
- DISTRIBUTED TRANSACTION

Clock Synchronization

In a centralized system, time is unambiguous. When a process wants to know the time, it makes a system call and the kernel tells it.

If process A asks for the time. and then a little later process B asks for the time, the value that B gets will be higher than (or possibly equal to) the value A got**.** ***It will certainly not be lower.*** In a distributed system, achieving agreement on time is not trivial.

And in a distributed system were no global agreement on time

Is it possible to synchronize all the clocks in a distributed system?

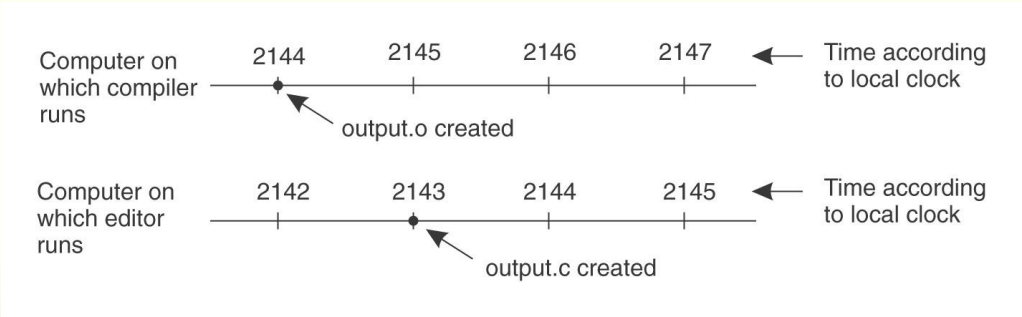


Fig1: When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Clock Synchronization

Physical Clocks

✓A computer timer is usually a precisely machined quartz crystal. When kept under tension, quartz crystals oscillate at a well-defined frequency

✓each crystal are two registers, a counter and a holding register.

✓ Each oscillation of the crystal decrements the counter by one. When the counter gets to zero, an interrupt is generated and the counter is reloaded from the holding register

✓But in the distributed system , it is impossible to guarantee that the crystals in different computers all run at exactly the same frequency.

Clock Synchronization

Physical Clocks

✓when a system has *n* computers, all n crystals will run at slightly different rates, causing the (software) clocks gradually to get out of synch and give different values when read out. This difference in time values is called **clock skew**

✓In some systems (e.g., real-time systems), the actual clock time is important. Under these circumstances, external physical clocks are needed

For reasons of efficiency and redundancy, multiple physical clocks are generally considered desirable, which yields two problems:

1. **How do we synchronize them with real world clocks?**
2. **How do we synchronize the clocks with each other?**

The Berkeley Algorithm

Here the time server (actually, a time daemon) is active, polling every machine from time to time to ask what time it is there

Based on the answers, *it computes an average time* and tells all the other machines to advance their clocks to the new time or slow their clocks down until some specified reduction has been achieved

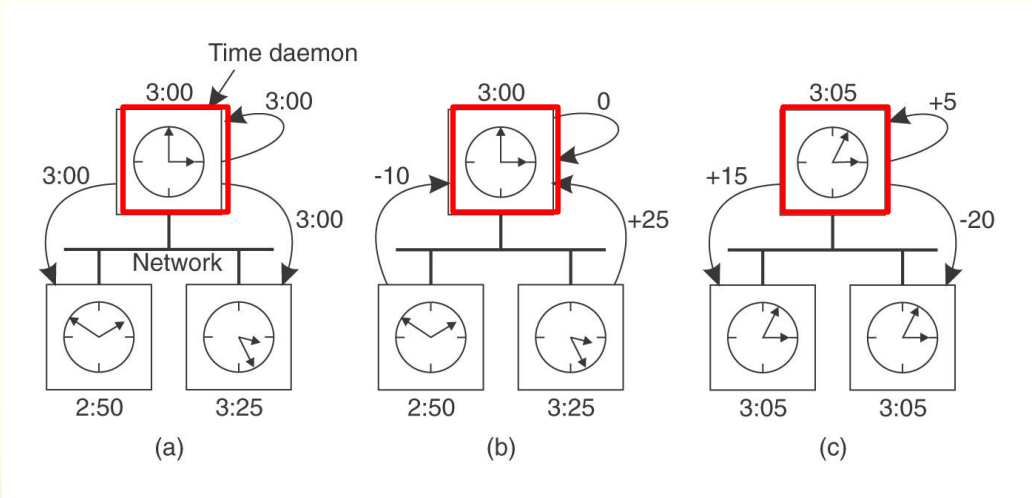


Fig2: (a) The time daemon asks all the other machines for their clock values. (b) The machines answer(c) The time daemon tells everyone how to adjust their clock.

Logical Clocks

For many purposes, it is sufficient that all machines agree on the same time. It is not essential that this time also agrees with the real time as announced on the radio every hour.

for the curtain class of algorithms, it is the internal consistency of the clocks that matters, not whether they are particularly close to the real time. for these algorithms, it is conventional to speak of the clocks as **logical clocks.**

Lamport's Logical Clocks

Lamport showed **that although clock synchronization is possible, it need not be absolute:**

If two processes do not interact, it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable and thus could not cause problems

what usually matters is not that all processes agree on exactly what time it is, but rather
that they agree on the order in which events occur

Lamport's algorithm, synchronizes logical clocks.

Lamport's timestamps

To synchronize logical clocks, Lamport defined a relation called happens-before:

The expression (a $\longrightarrow$ b) is read "a happens before b" and means that all processes agree that first event a occurs, then afterward, event b occurs, then afterward, event b occurs.

The happens-before relation can be observed directly in two situations:

1. If a and b are events in the same process, and a occurs before b, then (a $\rightarrow$ b) is true.
2. If a is the event of a message being sent by one process, and b is the event of the message being received by another process, then *a* $\rightarrow$ *b* is also true. A message cannot be received before it is sent, or even at the same time it is sent, since it takes a finite, nonzero amount of time to arrive.

Lamport's timestamps

Happens-before is a transitive relation, so if $(a \rightarrow b)$ *and* $(b \rightarrow c)$ *, then* $(a \rightarrow c)$.

If two events, *x and y, happen in different processes that do not exchange messages,* then $x \sim y$ *is not true, but neither is* $y \sim x$. These events are said to be **concurrent, which simply means that nothing can be** said (or need be said) about when the events happened or which event happened first.

a If *a and b are events in the same process, and a occurs before b:*

$(a \rightarrow b)$ *then* $C(a)<C(b)$

b Similarly, If *a is the event of a message being sent by one process, and b is the* event of the message being received by another process :

$(a \rightarrow b)$ *then* $C(a)<C(b)$

In addition, the clock time, C, must always go forward (increasing), never backward (decreasing).

Lamport's algorithm

Lamport's solution follows directly from the happens-before relation.

Example:

Consider the three processes depicted in Figure
The processes run on different machines, each with its own clock, running at its own speed.

As can be seen from the figure, when the clock has ticked 6 times in process PI, it has ticked 8 times in process Pz and 10 times in process P3' Each clock runs at a constant rate, but the rates are different due to differences in the crystals.

Lamport's solution follows directly from the happens-before relation.

Lamport's algorithm

Each message carries the sending time according to the sender's clock. When a message arrives and the receiver‘s clock shows a value prior to the time the message was sent, the receiver fast forwards its clock to be one more than the sending time.

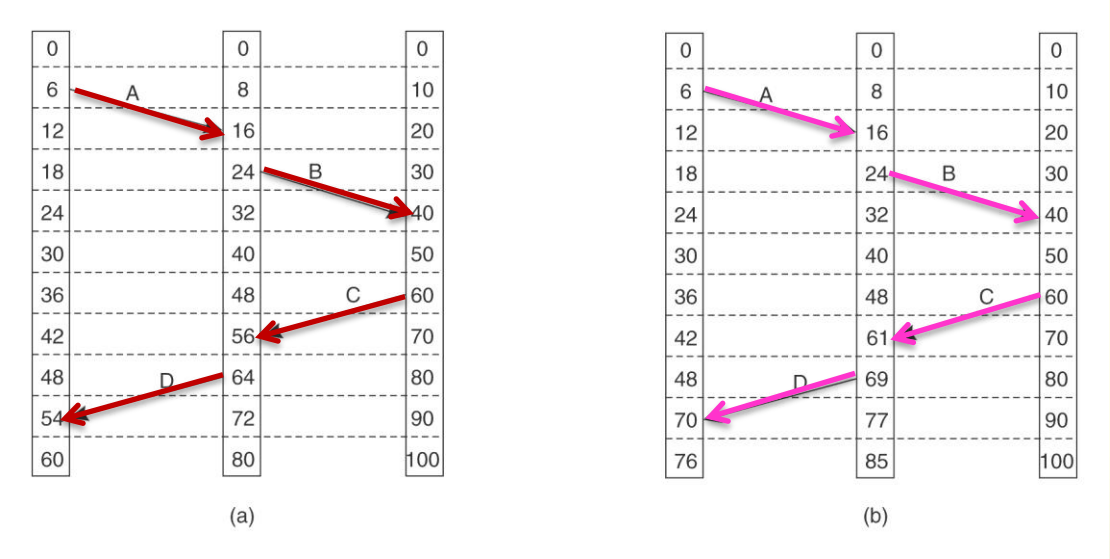


Fig3: The processes run on different machines, each with its own clock, running at its own speed.

Lamport's algorithm

To implement Lamport's logical clocks, each process *Pi maintains a local counter* Ci. These counters are updated as follows steps :

1. Before executing an event (i.e., sending a message over the network, delivering a message to an application, or some other internal event), *Pi executes Ci← Ci+ 1.*

2. When process Pi sends a message m to Pj it sets ts (m) equal to Ci after having executed the previous step.

3. Upon the receipt of a message m, process Pj adjusts its own local counter as Cj ← max{Cj, ts (m) }, after which it then executes the first step and delivers the message to the application.

Lamport's algorithm

then **using Lamport's timestamps**, we now have a way to assign time to all events in a distributed system subject to the following conditions:

1. If a happens before b in the same process ,C(a)<C(b).
2. If a and b represent the sending and receiving of message, respectively, C(a)<C(b).
3. For all distinctive events a and b , C(a)≠ C(b).

Election Algorithms

Many distributed algorithms require **one process to act as coordinator, initiator**, or otherwise **perform some special role**.

It does not matter **which process takes on this special responsibility**, but one of them has to do it.

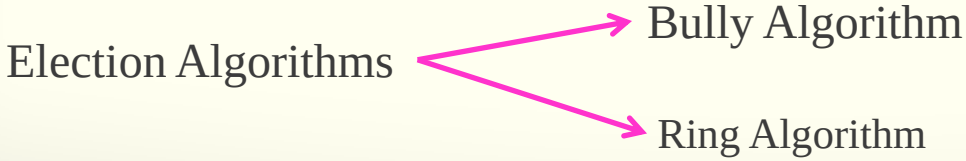
If all processes are exactly the same, with no distinguishing characteristics, there is no way to select one of them to be special.

Election Algorithms

- we will assume that each process has a unique number, for example, its network address.

- In general, election algorithms attempt to locate the process with the highest process number and designate it as coordinator.

- **The algorithms differ in the way they do the location**.



The Bully Algorithm

When any process notices that the coordinator is no longer responding to requests, it initiates an election.

A process, *P, holds an election as follows:*

1. **P sends an ELECTION message to all processes with higher numbers.**
2. **If no one responds, P wins the election and becomes coordinator.**
3. **If one of the higher-ups answers, it takes over. P's job is done.**

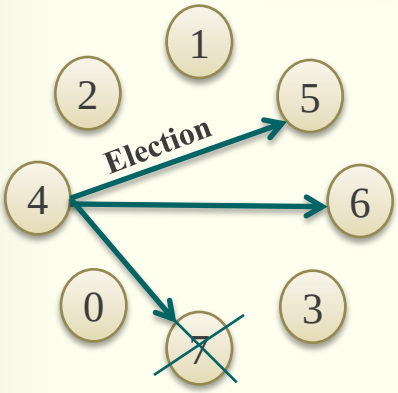
The Bully Algorithm

At any moment, a process can **get an ELECTION message** from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an **OK message** back to the sender to indicate that he is alive and will take over.

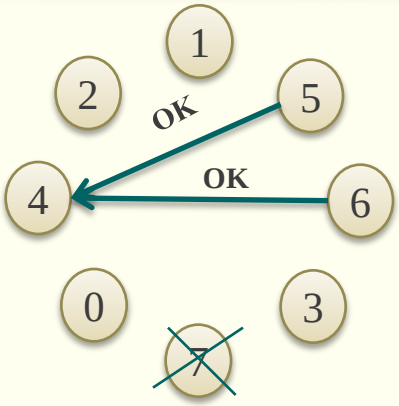
Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

If a process that was previously down comes back up, **it holds an election**. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. **Thus the biggest guy in town always wins, hence the name "bully algorithm."**

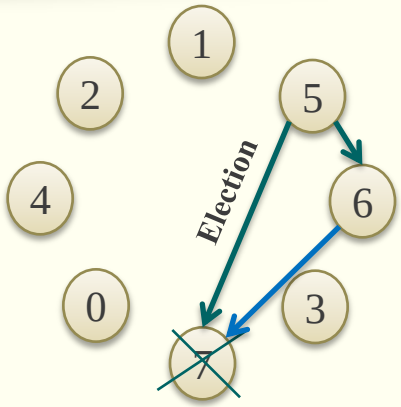
The Bully Algorithm



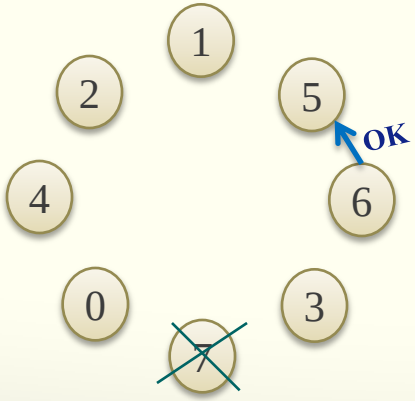
(a)



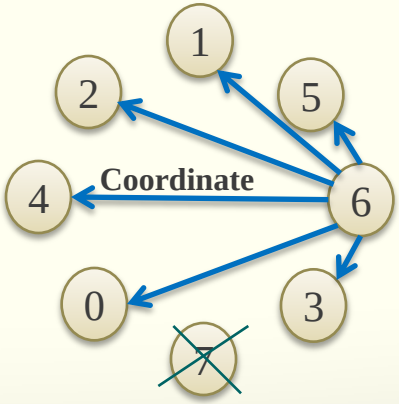
(b) Previous coordinator has crashed



(c)



(d)



(e)

A Ring Algorithm

✓This algorithm is based on the use of a ring.

✓We assume that the processes are physically or logically ordered, so that each process knows who its successor is.

✓When any process notices that the coordinator is not functioning, it builds an *ELECTION* message containing **its own process number** and **sends the message to' its successor.**

✓If the successor is down, the sender skips over the successor and goes to the next member along the ring. or the one after that, until a running process is located.

✓At each step along the way, the sender adds its own process number to the list in the message effectively making itself a candidate to be elected as coordinator.

A Ring Algorithm

Eventually, the message gets back to the process that started it all.

That process recognizes this event when it receives an incoming message containing its own process number.

At that point, the message type is changed to *COORDINATOR* and circulated once again, this time to inform everyone else who the coordinator is (the list member with the highest number) and who the members of the new ring are.

When this message has circulated once, it is removed and everyone goes back to work.

A Ring Algorithm

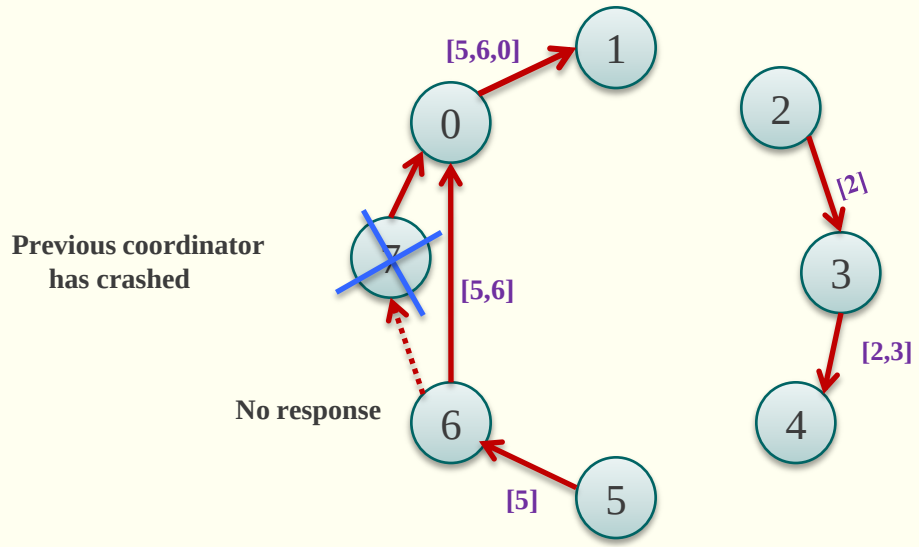


Fig3: Election algorithm using a ring.

MUTUAL EXCLUSION

Centralized Algorithm

Distributed Algorithm

Token Ring Algorithm

A Centralized Algorithm

One process is elected as the **coordinator**

Whenever a process wants to access a shared resource, it sends a request message to the coordinator stating which resource it wants to access and asking for permission.

If no other process is currently accessing that resource, the coordinator sends back a reply granting permission When the reply arrives, the requesting process can go ahead.

A Centralized Algorithm

Now suppose that another process asks for permission to access the resource.

The coordinator knows that a different process is already at the resource, so it cannot grant permission.

The coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply.

Alternatively, it could send a reply saying "**permission denied**."

A Centralized Algorithm

When process 1 is finished with the resource, it sends a message to the coordinator releasing its exclusive access

The coordinator takes the first item off the queue of deferred requests and sends that process a grant message.

If the process was still blocked , it unblocks and accesses the resource.

A Centralized Algorithm

In the OSI model, communication is divided up into seven levels or layers

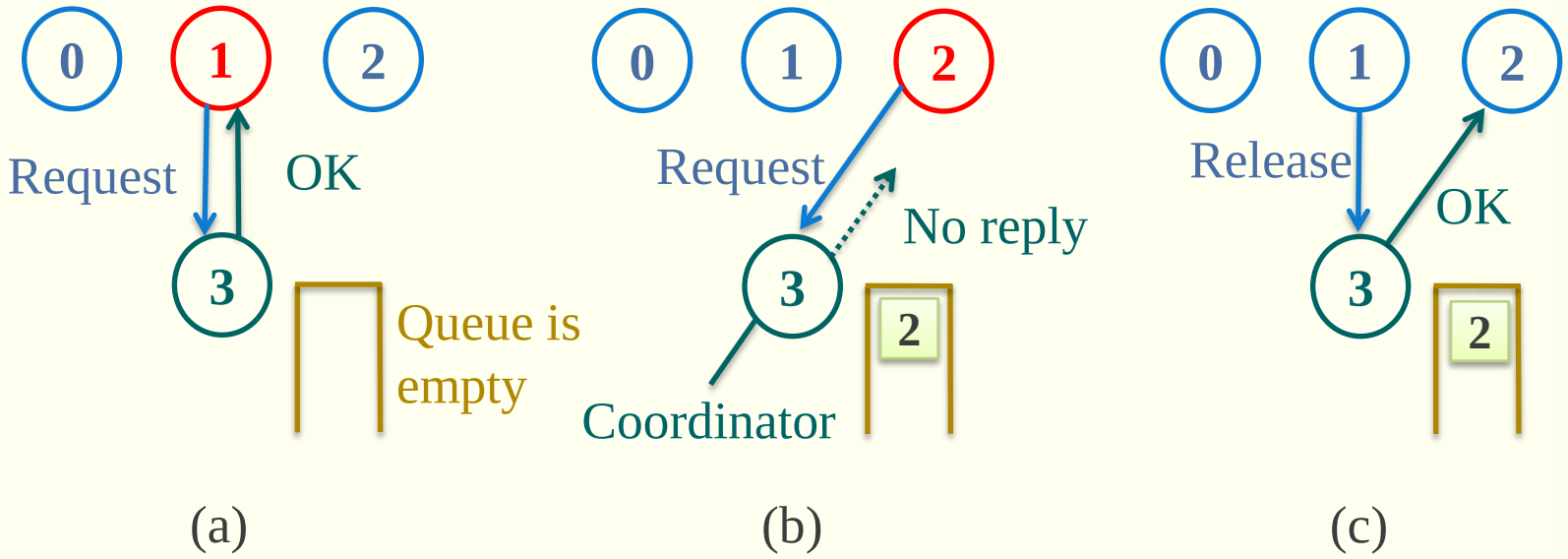


Fig 13: A Centralized Algorithm

A Distributed algorithm

When a process wants to access a shared resource, it builds a message containing the name of the resource, its process number, and the current (logical) time.

When a process receives a request message from another process

Distributed algorithm

1. If the receiver is not accessing the resource and does not want to access it, it sends back an ***OK*** message to the sender.

2. If the receiver already has access to the resource, it simply does not reply. Instead, it **queues** the request.

3. If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with me. one contained in the message that it has sent everyone. The lowest one wins. If the incoming message has a lower timestamp, the receiver sends back an *OK* message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

Distributed algorithm

After sending out requests asking permission, a process sits back and waits until everyone else has given permission.

As soon as all the permissions are in, it may go ahead.

When it is finished, it sends *OK* messages to all processes on its queue and deletes them all from the queue.

This algorithm is **slower**, **more complicated**, **more expensive**, and **less robust** that the original centralized one.

Distributed algorithm

- Process 0 sends everyone a request with timestamp 8, while at the same time, process 2 sends everyone a request with timestamp 12.
- Process 1 is not interested in the resource, so it sends *OK* to both senders.
- Processes 0 and 2 both see the conflict and compare timestamps.
- Process 2 sees that it has lost, so it grants permission to 0 by sending *OK.*
- Process 0 now queues the request from 2 for later processing and access the resource, as shown in the Figure.
- When it is finished, it removes the request from 2 from its queue and sends an *OK* message to process 2, allowing the latter to go ahead,

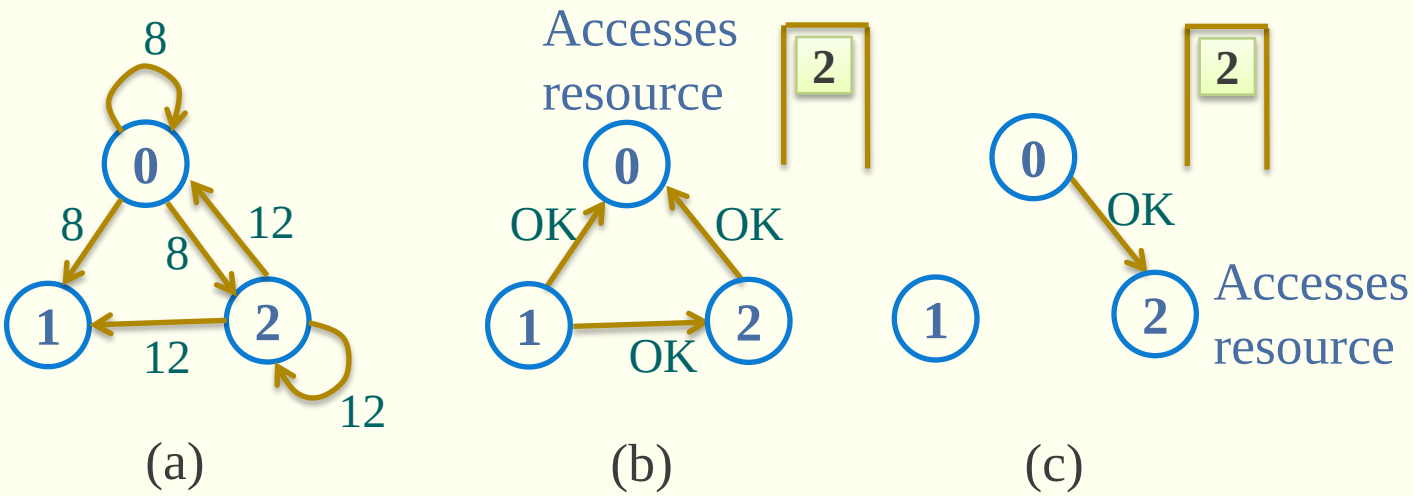


Fig 14: Distributed Algorithm

A Token Ring Algorithm

In software, a logical ring is constructed in which each process is assigned a **position in the ring**, as shown The ring positions may be allocated in numerical order of network addresses or some other means.

When the ring is initialized, process 0 is **given a token**.

The token circulates around the ring.

A Token Ring Algorithm

When a process acquires the token from its neighbor, it **checks** to see if it needs to access **the shared resource**.

If so, the process goes ahead, does all the work it needs to, and releases the resources.

After it has finished, it passes the token along the ring.

It is **not permitted** to immediately enter the resource again using the same token.

As a consequence, when no processes need the resource, the token just circulates at high speed around the ring.

A Token Ring Algorithm

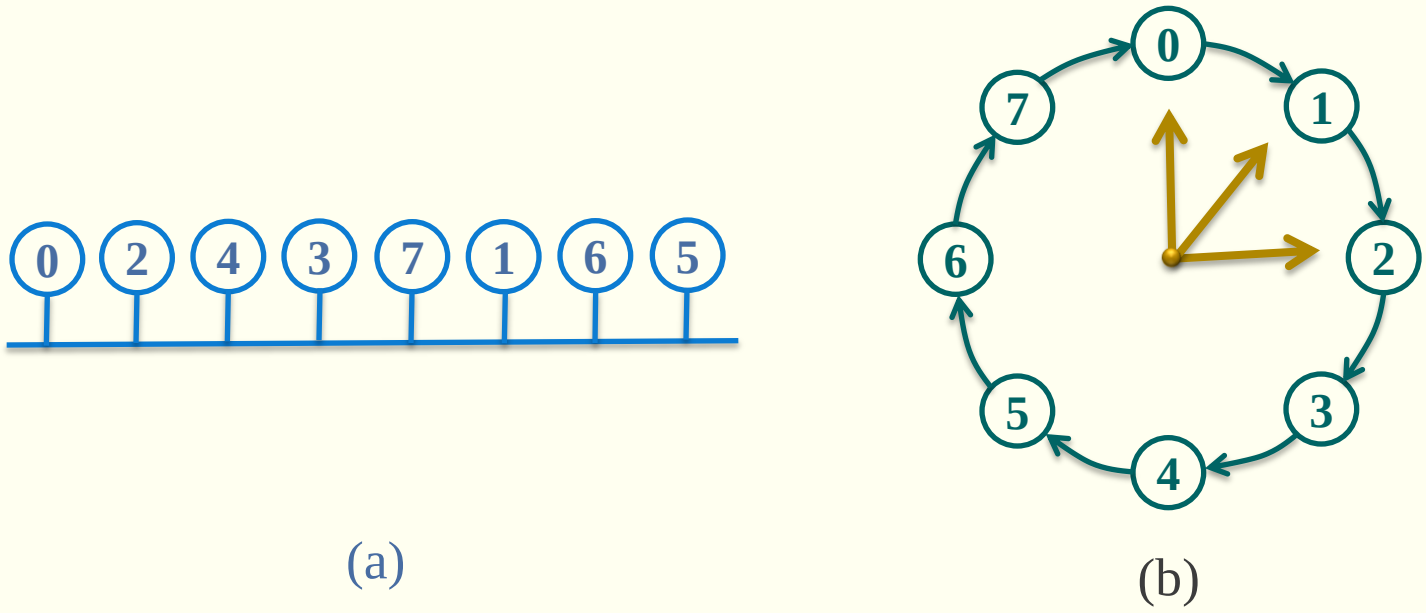


Fig 15: A Token Ring Algorithm

Distributed transactions

- The Transaction model
- Classification of transactions

The Transaction model

- One process announces that it wants to begin a translation with one or more other processes.
- They can negotiate various options, **create** and **delete** entities, and **perform** operations for a while.
- Then the **initiator** announces that it wants all the other to commit themselves to the work done so far .
- If all of them agree, the results are made **permanent**.
- If one or more processes refuse (or crash before agreement), the situation reverts to exactly the state it was in **before the transaction** begin.

The Transaction model

Low level communicative objects in transactions

Primitive	description
BEGIN_TRANSACTION	Mark the start of a transaction
END_TRANSACTION	Terminate the transaction and try to commit
ABORT_TRANSACTION	Kill the transaction and restore to old value
READ	Read data from a file , a table , or otherwise
WRITE	Write data to a file , a table , or otherwise

Features of transactions

- Atomic: To the outside world, the transaction happens indivisibly.
- Consistent: The transaction does not violate system invariants.
- Isolated: Concurrent transactions do not interfere with each other.
- Durable: Once a transaction commits, the changes are permanent.

Atomic : each transaction either happens completely, or not at all,

Classification of transaction

- **Flat transaction**
- **Nested transaction**
- **Distributed transaction**

The strength of the atomicity property of the flat transaction also is partly its weakness .

Nested transaction

A Nested transaction is constructed from a number of sub transactions.

The top-level transaction may fork off children that run in parallel with one another , on deferent machines **, to gain performance** or **simplify programming** .

If an enclosing (higher-Level)transaction aborts , all its underlying **sub transactions** have to be aborted as well.

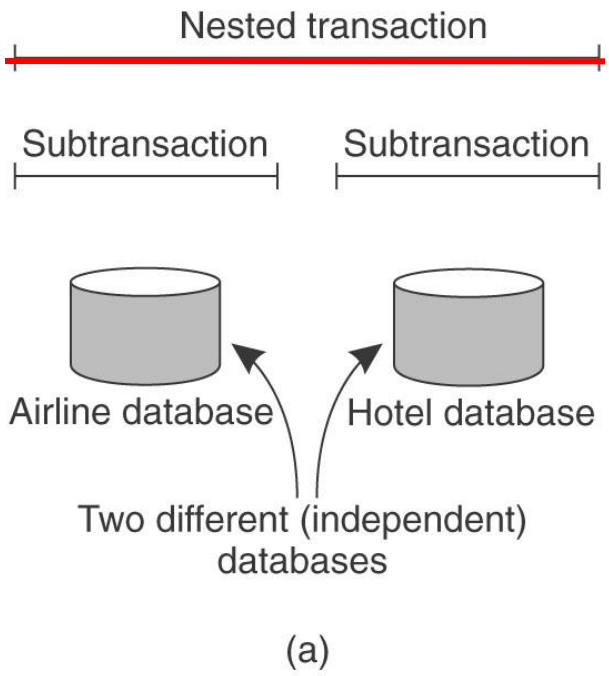
Distributed transaction

A **nested transaction** is a transaction that is logically decomposed into a hierarchy of **sub transactions**.

A **distributed transaction** is logically a flat , **indivisible transaction** that operates on **distributed data**.

The main problem with Distributed transactions is that separate Distributed algorithms are needed to handle the **locking of data** and **committing the entire transaction**.

Distributed transaction



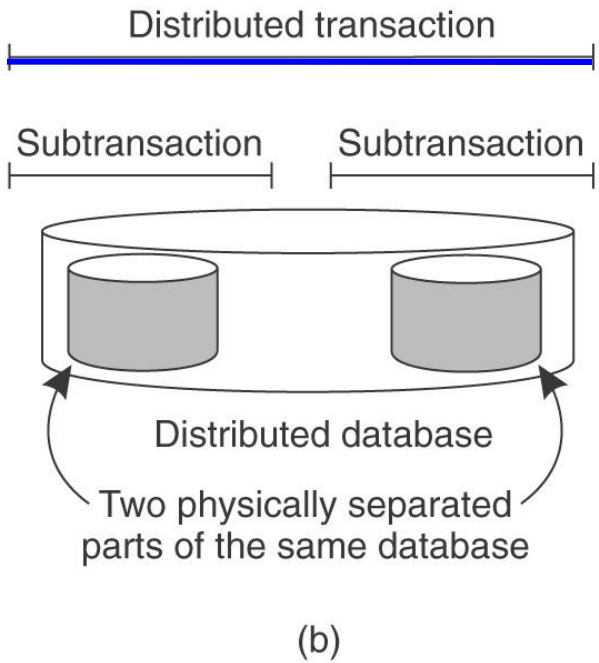


Fig 20: Distributed transaction

Concurrency control

The goal of concurrency control id to allow several transactions to be executed simultaneously , but in such a way that collection of data items that is begin manipulated , is left in a consistent state.

Concurrency control is best understood in terms of three different managers Which are organized in a layered fashion.

Concurrency control

1. The bottom later consists of a data manager that performs the actual read and write operations on data.
2. The middle layer consists of a scheduler and carries tha main responsibility for properly controlling. It determines which transaction is allowed to pass a read or write operation to the data manager and at which time.
3. The highest layer contains the transaction manager , which is primarily responsible for guaranteeing atomicity of transactions.

Concurrency control

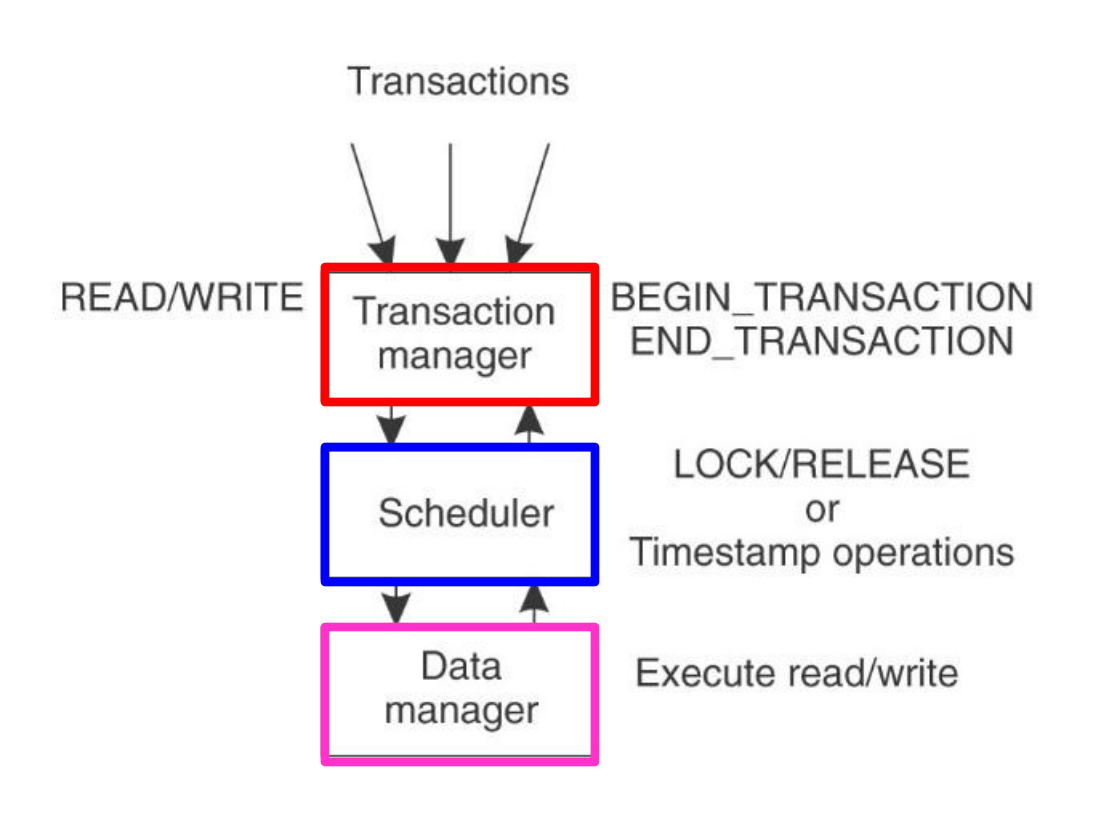


Fig 23: Concurrency control

Two-phase locking

The oldest and most widely used Concurrency control algorithm is **locking**.

When a process needs to read or write a data item as part of a transaction , it requests the scheduler to grand it a lock for that data item.

Scheduler needs to apply Two-Phase locking algorithm.

In TWO-Phase locking (2PL) , The scheduler first acquires all the locks it needs during the growing phase, and the releases them during the shrinking phase.

Two-phase locking

The following three rules

1. When the scheduler receives an operation oper(T.x) from the transaction manager , it tests whether that operation conflicts with any other operation for which it already grandy a lock. If there is a conflict , operation oper(T.x) is delayed (and thus also transaction T). if there is no conflict , the scheduler grants a lock for data item x, and passes the operation to the data manager.
2. The scheduler will never release a lock for data item x , until the data manager acknowledges it has performed the operation for which the lock was set.
3. Once the scheduler has released a lock on behalt of a transaction T,it will never grant another lock on behalf of T, no mather for which data item T is requesting a lock . Any attempt by T to acquire another lock is a programming error that abort T.

Two-phase locking

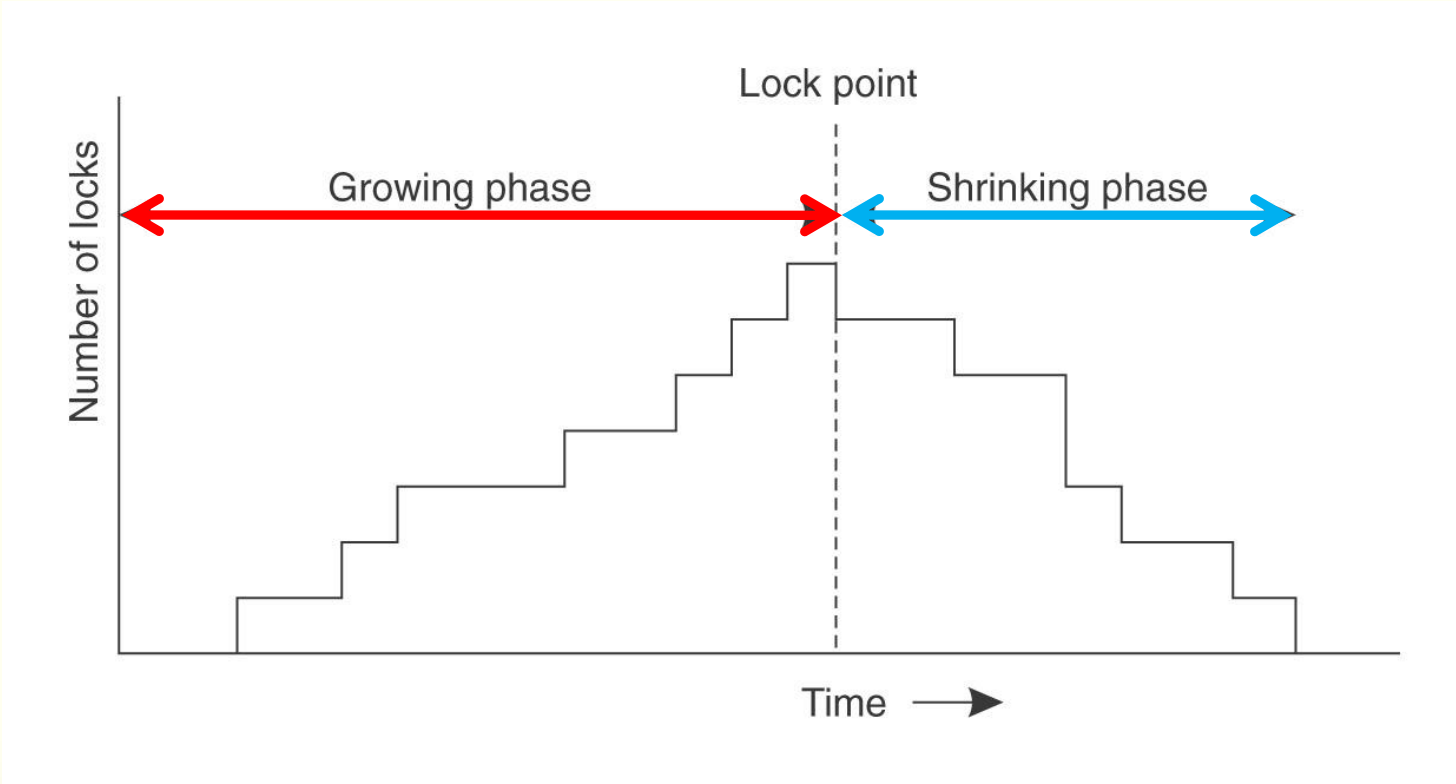


Fig 26: Two-phase locking

Two-phase locking

The shrinking phase does not take place until the transaction has finished running and has either committed or aborted , leading to the release of locks as shown.

This policy , called **strict two-phase locking** , has two main advantages.

First , a transaction always read a value written by a committed transaction ; therefore, one never has to abort a transaction

Second, all lock acquisitions and releases can be handled by the system without the transaction begin aware of them

lock are acquired whenever a data item is to be accessed are released when the transaction has finished.

Strict two-phase locking

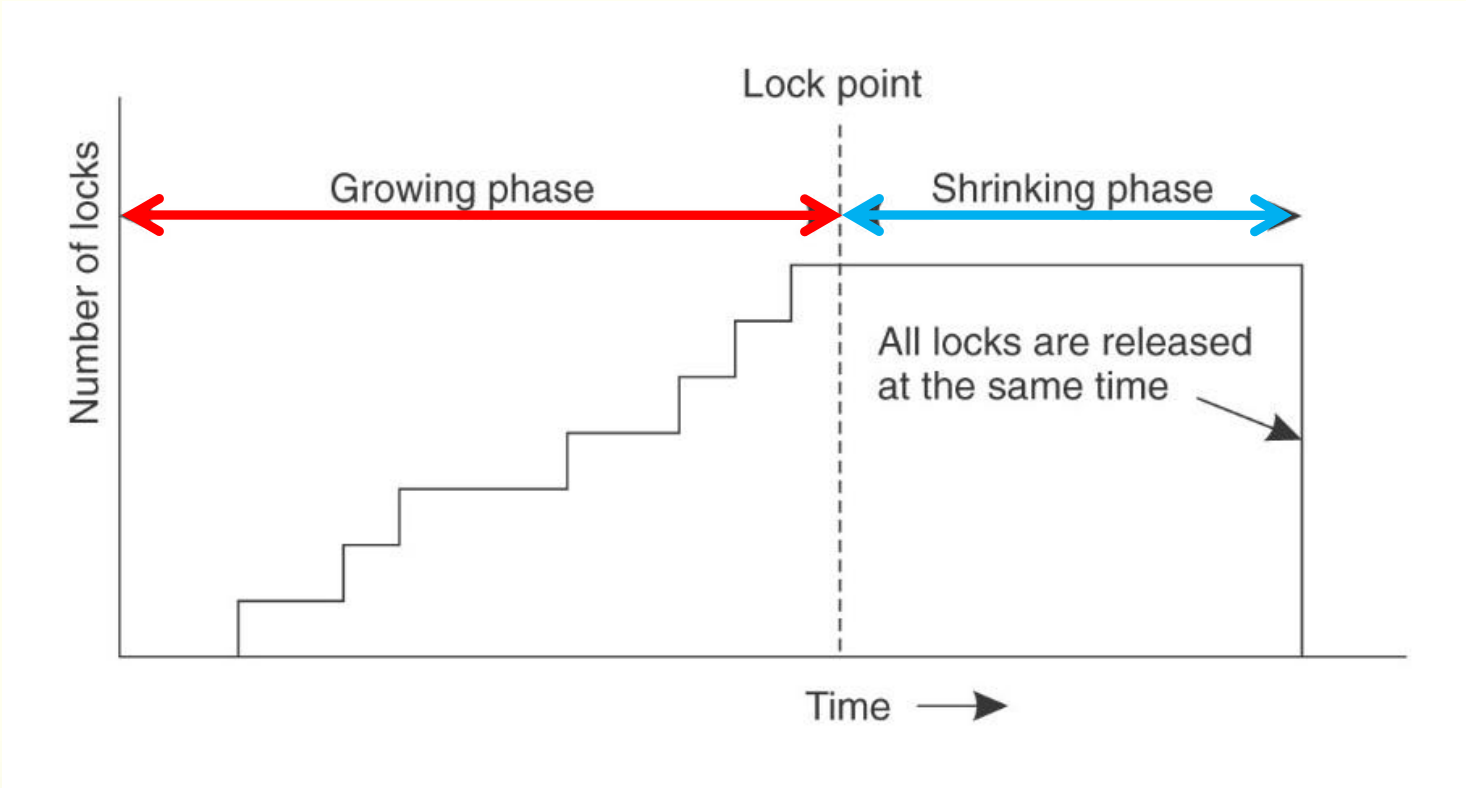


Fig 27: Two-phase locking

Two-phase locking

Both two-phase locking and strict two-phase locking can lead to **deadlocks**.

The usual techniques apply here , such as acquiring all locks in some canonical order to prevent **hold-and-wait** cycles.

Also possible is deadlock detection by maintaining an explicit graph of which process has which locks and wants which locks.

when it is known in advance that a lock will never be held longer than t sec, a timeout scheme van be used: if a lock remains continuously under the same ownership for longer than t sec , there must be a deadlock.

Two-phase locking

Basic two-phase locking scheme can be implemented in a distributed system.

Transaction manager communicates with this centralized lock manager, from which it receives lock grants.

When a lock has been granted, the transaction manager subsequently communicates directly with the data managers .

Finally , in **distributed 2PL** , it is assumed that **data** may be replicated across multiple machines.

End of Chapter 5