Module 3

Accessing Remote Resources

Two Issues

- Remote service invocation
  - ➡ How do we request services from remote objects
- Distributed naming
  - ➡ How do we assign and know the names of remote objects

Remote Service Invocation

Programming Models for Distributed Applications

- Remote Procedure Call (RPC)
  - ➡ Extension of the conventional procedure call model
  - ➡ Allows client programs to call procedures in server programs
- Remote Method Invocation (RMI)
  - ➡ Object-oriented
  - ➡ Extension of local method invocation
  - ➡ Allows an object in one process to invoke the methods of an object in another process
- Message-Based Communication
  - ➡ Allows sender and receiver to communicate without blocking and without the receiver having to be running

Essentials of Interprocess Communication

- Ability to communicate - exchange messages
  - ➡ This is the responsibility of the networks and the request-reply protocol
- Ability to “talk” meaningfully
  - ➡ Interfaces
    - ➡ Processes have to be able to understand what each other is sending
      - ✦ Agreed standards for data representation

Interface

- Defines what each module can do in a distributed system
- Service interface (in RPC model)
  - ➡ specification of procedures offered by a server
- Remote interface (in RMI model)
  - ➡ specification of methods of an object that can be invoked by objects in other processes
- Example: CORBA IDL

```
//In file Person.idl
Struct Person {
   string name;
   string place;
   long year;
};
Inerface PersonList {
   readonly attribute string listname;
   void addPerson(in Person p);
   void getPerson(in string name, out Person p);
   long number();
};
```

Remote Procedure Call (RPC)

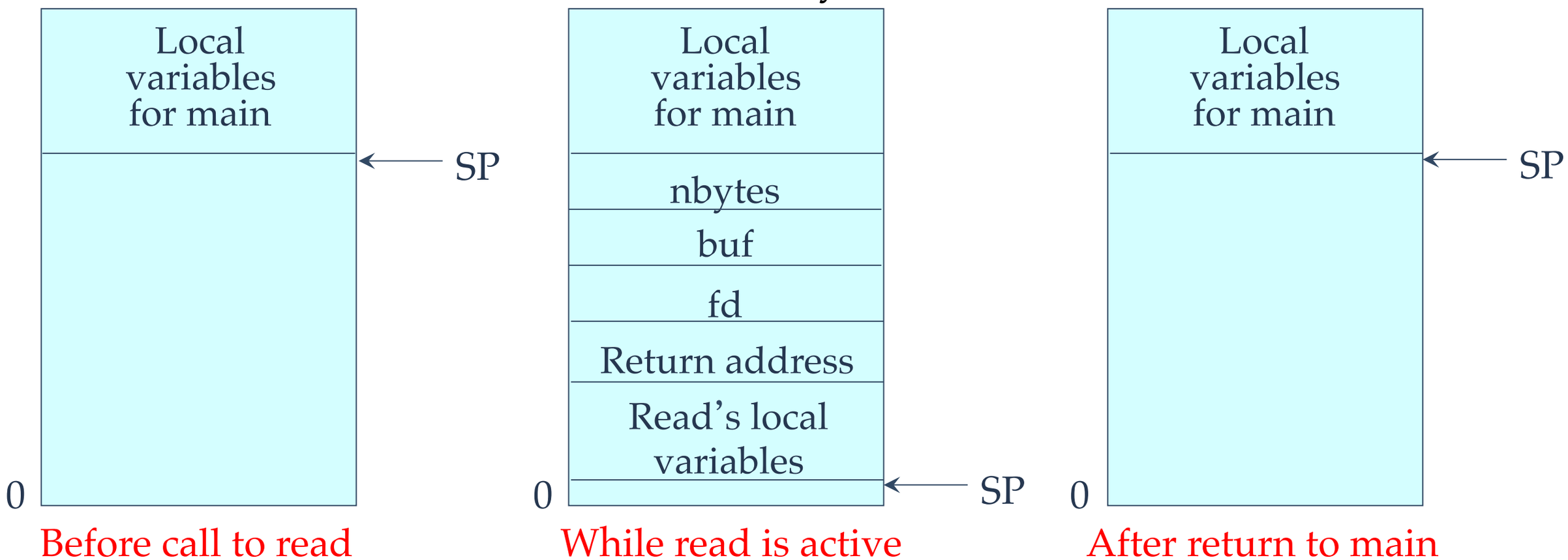
- Extension of the familiar procedure call semantics to distributed systems (with a few twists)
- Allows programs to call procedures on other machines
- Process on machine A calls a procedure on machine B:
  - The calling process on A is suspended; and execution of the called procedure takes place on B;
  - Information can be transported from the caller to the callee in the parameters; and information can come back in the procedure result
  - No message passing or I/O at all is visible to the programmer
- Subtle problems:
  - Calling and called procedures execute in different address spaces
  - Parameters and results have to be passed, sometimes between different machines
  - Both machines can crash

Local Procedure Call

- Consider a C program with the following call in the main program

```
count = read(fd, buf, nbytes)
```

where **fd** (file descriptor) and **nbytes** (no. bytes to read) are integers, and **buf** (buffer into which data are read) is an array of characters

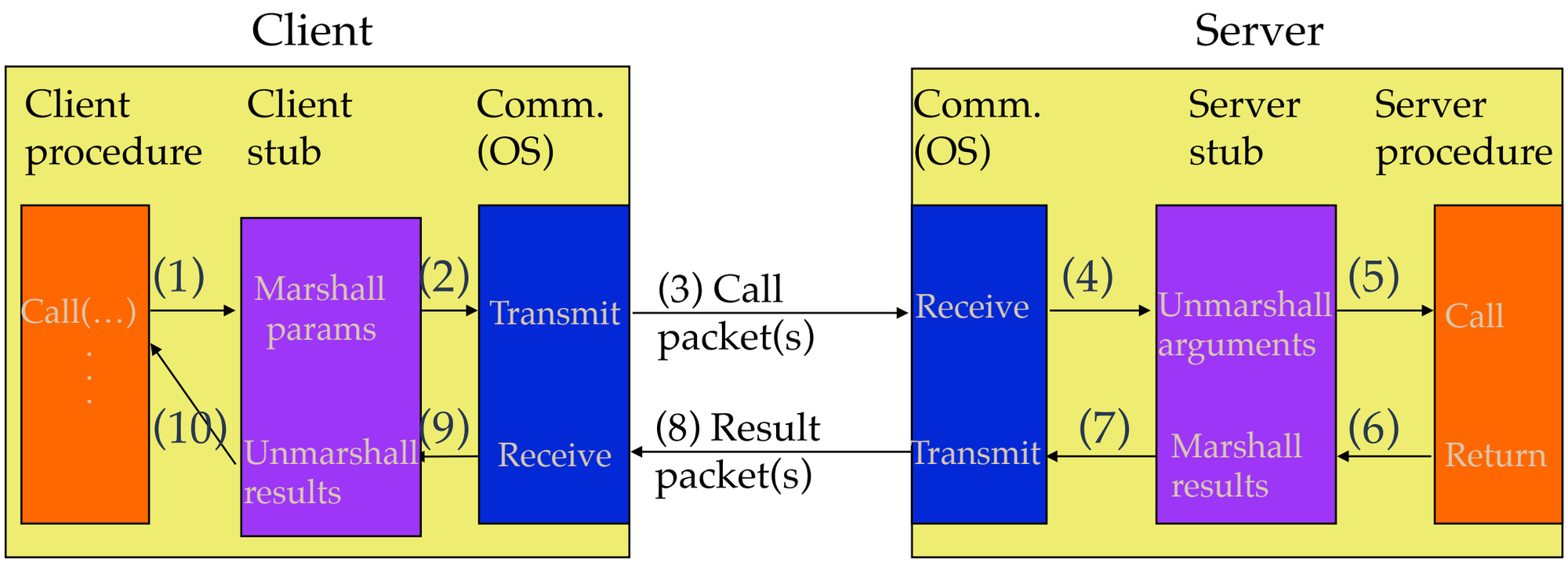


The read routine is extracted from library by linker and inserted into the object program. It puts the parameters in registers and issues a READ system call by trapping to the kernel.

RPC Operation

- Objective: Make interprocess communication among remote processes similar to local ones (i.e., make distribution of processes transparent)
- RPC achieves its transparency in a way analogous to the read operation in a traditional single processor system
  - ➡ When read is a remote procedure (that will run on the file server's machine), a different version of read, called client stub, is put into the library
  - ➡ Like the original read, client stub is called using the calling sequence described in previous slide
  - ➡ Also, like the original read, client stub traps to the kernel
  - ➡ Only unlike the original read, it does not put the parameters in registers and ask the kernel to give it data
  - ➡ Instead, it packs the parameters into a message and asks the kernel to send the message to the server
  - ➡ Following the call to send, client stub calls receive, blocking itself until the reply comes back
  - ➡ When reply returns, extract results and return to caller.

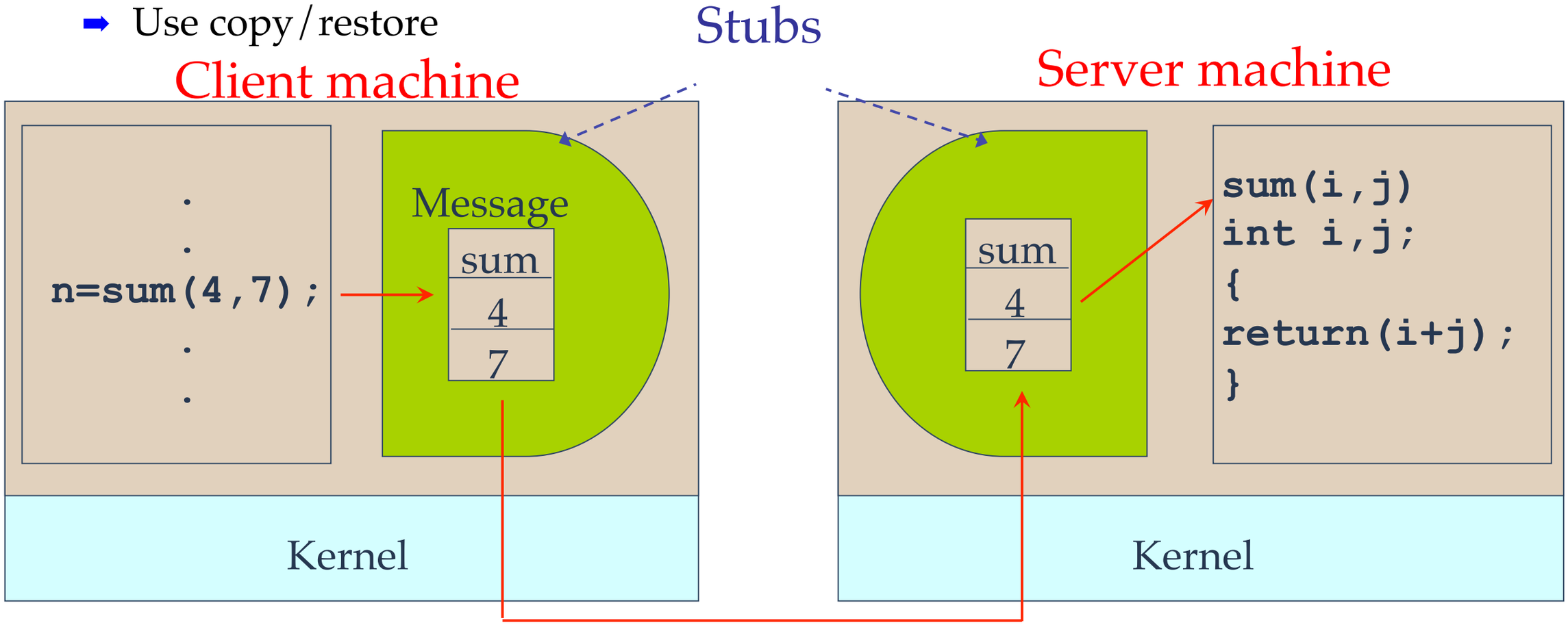
RPC Operation (2)



1. Client procedure calls the stub as local call
2. Client stub builds the message and calls the local OS
3. Client OS sends msg to server OS
4. Server OS hands over the message to server stub
5. Server stubs unpacks parameters and calls server procedure
6. Server procedure does its work and returns results to server stub
7. Server stub packs results in a message and calls its local OS
8. Server OS sends msg to client OS
9. Client OS hands over the message to client stub
10. Client stubs unpacks result and returns from procedure call

Parameter Passing

- Parameter passing by value
  - ➡ Compose messages that consist of structures for values
  - ➡ Problem: Multiple machine types in a distributed system. Each often has its own representation for data! ➔Agree on data representation
- Parameter passing by reference
  - ➡ Difficult
  - ➡ Use copy/restore



Parameter passing by value

RPC in Presence of Failures

- Five different classes of failures can occur in RPC systems
  - The client is unable to locate the server
  - The request message from the client to the server is lost
  - The reply message from the server to the client is lost
  - The server crashes after receiving a request
  - The client crashes after sending a request

Client Cannot Locate the Server

- Examples:
  - Server might be down
  - Server evolves (new version of the interface installed and new stubs generated) while the client is compiled with an older version of the client stub
- Possible solutions:
  - Use a special code, such as “-1”, as the return value of the procedure to indicate failure. In Unix, add a new error type and assign the corresponding value to the global variable `errno`.
    - “-1” can be a legal value to be returned, e.g., `sum(7, -8)`
  - Have the error raise an exception (like in ADA) or a signal (like in C).
    - Not every language has exceptions/signals (e.g., Pascal). Writing an exception/signal handler destroys the transparency

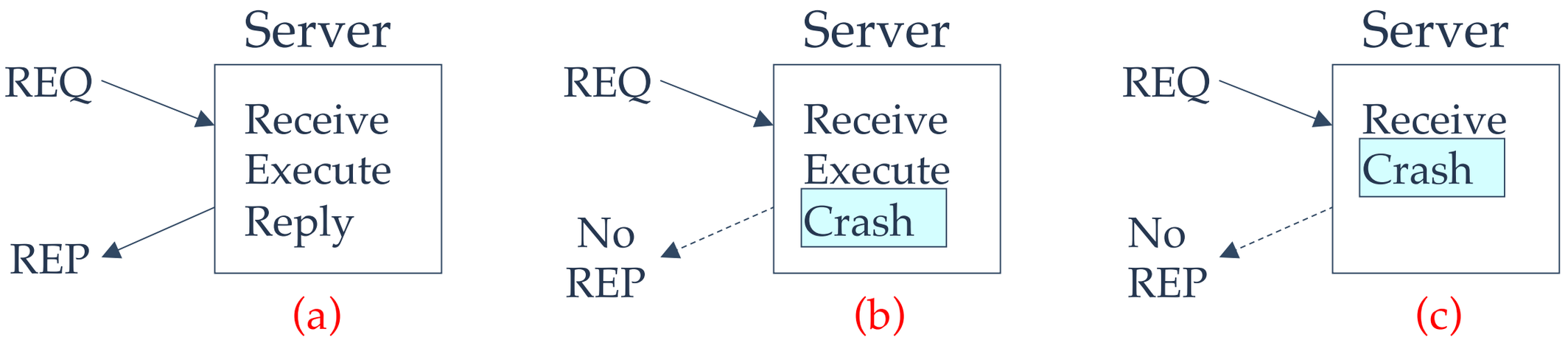
Lost Request Message

- Kernel starts a timer when sending the message:
  - ➡ If timer expires before reply or ACK comes back: Kernel retransmits
  - ➡ If message truly lost: Server will not differentiate between original and retransmission ➔ everything will work fine
  - ➡ If many requests are lost: Kernel gives up and falsely concludes that the server is down ➔ we are back to “Cannot locate server”

Lost Reply Message

- Kernel starts a timer when sending the message:
  - ➡ If timer expires before reply comes back: Retransmits the request
    - ✦ Problem: Not sure why no reply (reply/request lost or server slow) ?
  - ➡ If server is just slow: The procedure will be executed several times
    - ✦ Problem: What if the request is not idempotent, e.g. money transfer
  - ➡ Way out: Client's kernel assigns sequence numbers to requests to allow server's kernel to differentiate retransmissions from original

Server crashes



- Problem: Clients' kernel cannot differentiate between (b) and (c)

  (Note: Crash can occur before Receive, but this is the same as (c).)

- 3 schools of thought exist on what to do here:
  - ➡ Wait until the server reboots and try the operation again. Guarantees that RPC has been executed at least one time (at least once semantics)
  - ➡ Give up immediately and report back failure. Guarantees that RPC has been carried out at most one time (at most once semantics)
  - ➡ Client gets no help. Guarantees nothing (RPC may have been carried out anywhere from 0 to a large number). Easy to implement.

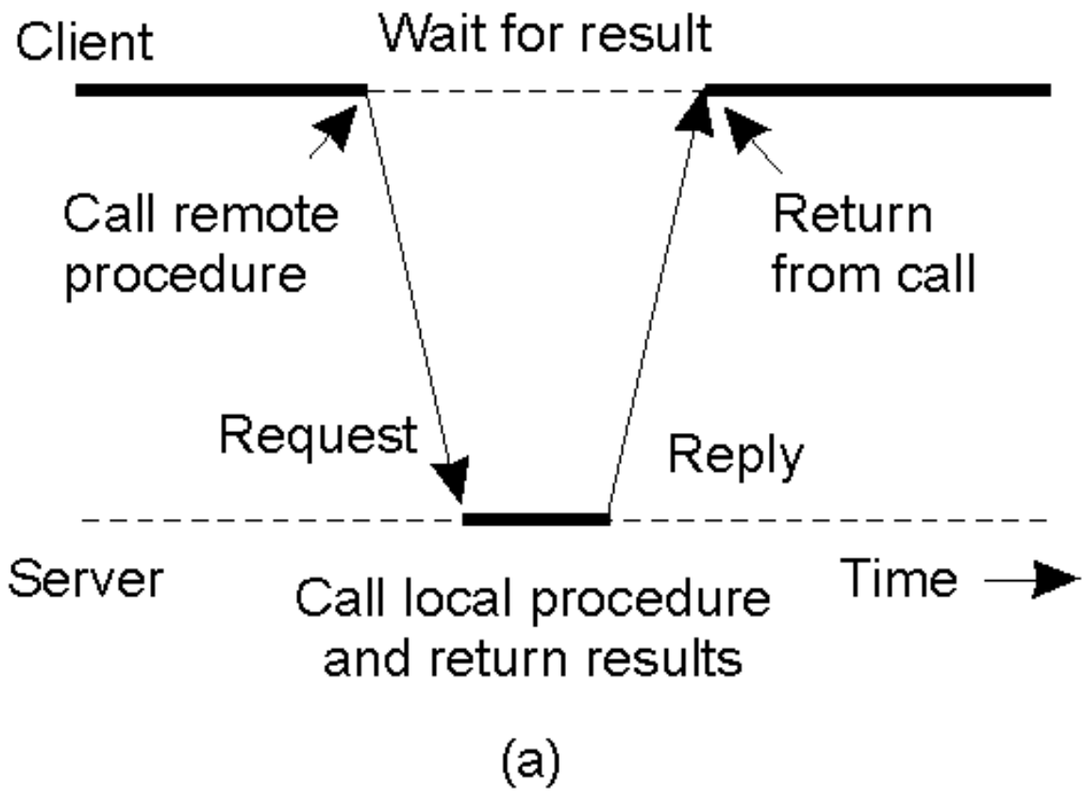
Client Crashes

- Client sends a request and crashes before the server replies: A computation is active and no parent is waiting for result (orphan)
  - ➡ Orphans waste CPU cycles and can lock files or tie up valuable resources
  - ➡ Orphans can cause confusion (client reboots and does RPC again, but the reply from the orphan comes back immediately afterwards)
- Possible solutions
  - ➡ Extermination: Before a client stub sends an RPC, it makes a log entry (in safe storage) telling what it is about to do. After a reboot, the log is checked and the orphan explicitly killed off.
    - ✦ Expense of writing a disk record for every RPC; orphans may do RPCs, thus creating grand orphans impossible to locate; impossibility to kill orphans if the network is partitioned due to failure.

Client Crashes (2)

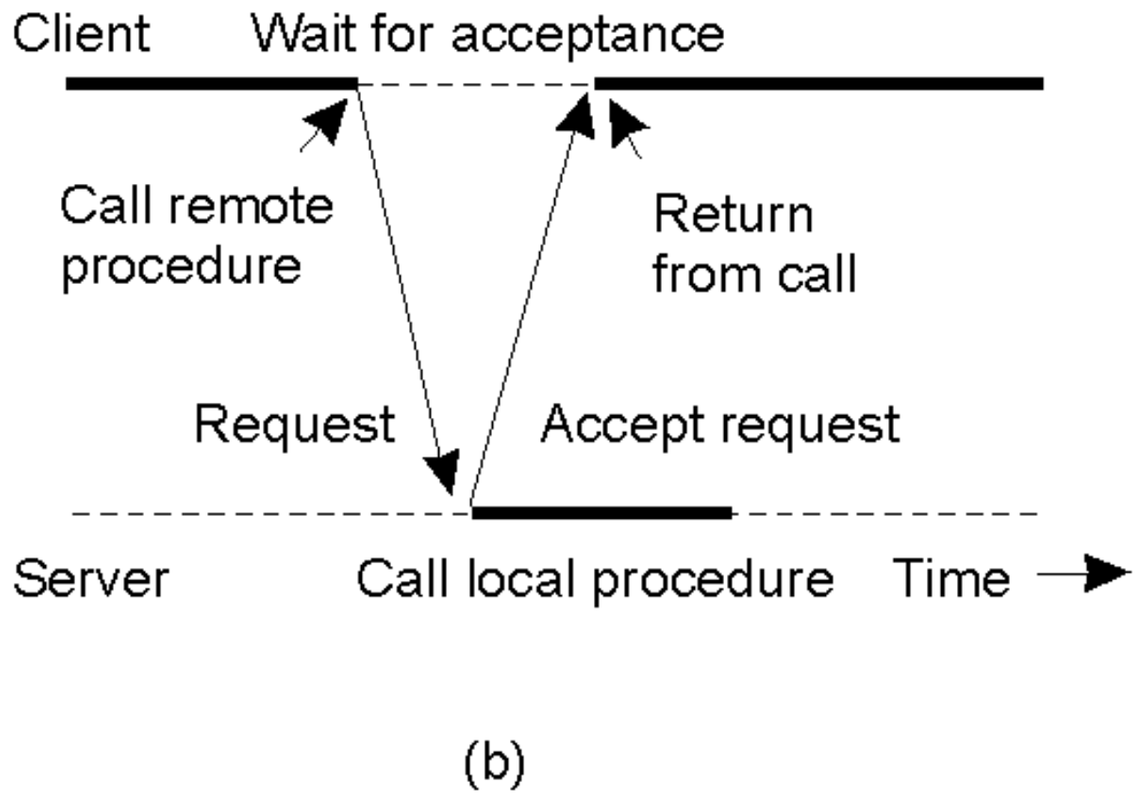
- Possible solutions (cont'd)
  - ➡ Reincarnation: Divide time up into sequentially numbered epochs. When a client reboots, it broadcasts a message declaring the start of a new epoch. When broadcast comes, all remote computations are killed. Solve problem without the need to write disk records
    - ✦ If network is partitioned, some orphans may survive. But, when they report back, they are easily detected given their obsolete epoch number
  - ➡ Gentle reincarnation: A variant of previous one, but less Draconian
    - ✦ When an epoch broadcast comes in, each machine that has remote computations tries to locate their owner. A computation is killed only if the owner cannot be found
  - ➡ Expiration: Each RPC is given a standard amount of time, T, to do the job. If it cannot finish, it must explicitly ask for another quantum.
    - ✦ Choosing a reasonable value of T in the face of RPCs with wildly differing requirements is difficult

Asynchronous RPC



(a)

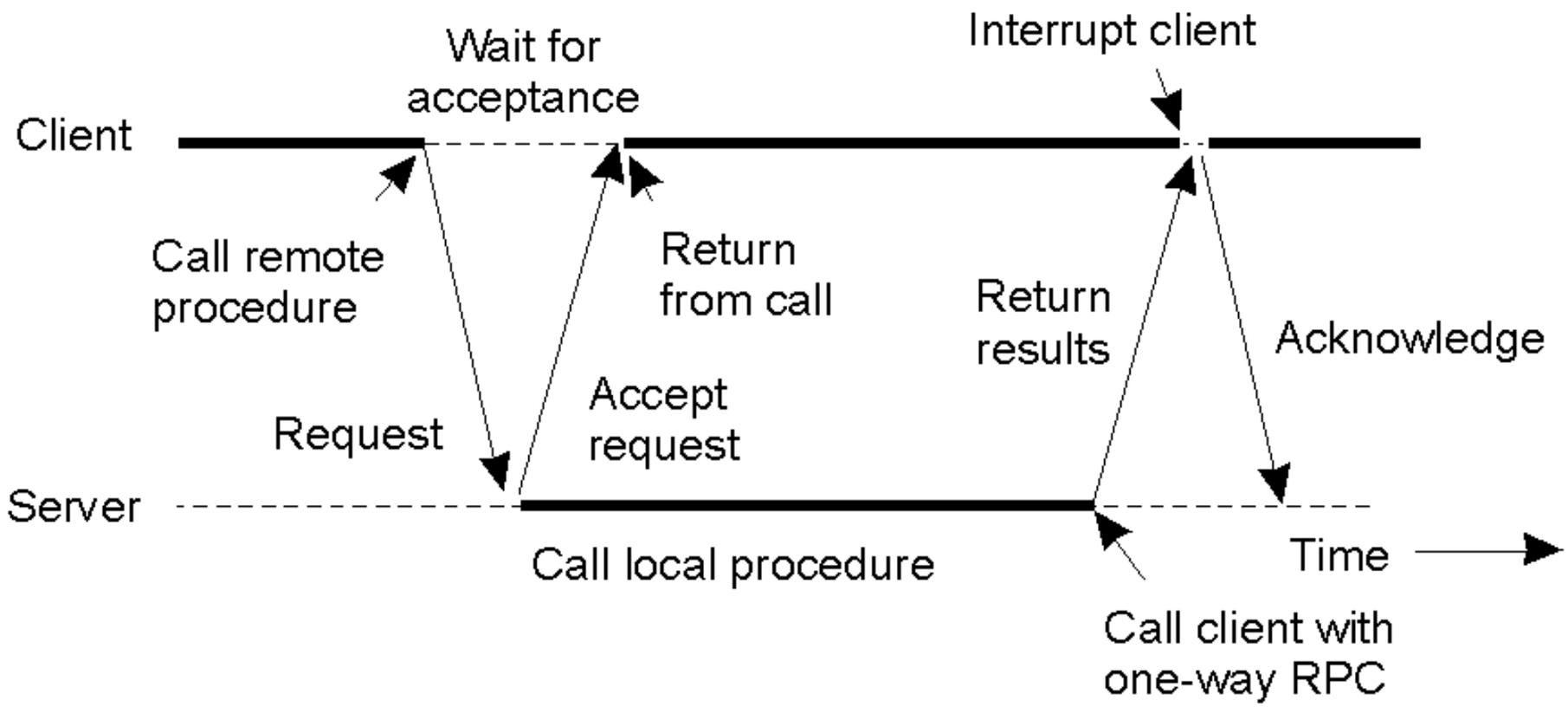
Traditional (synchronous) RPC interaction



(b)

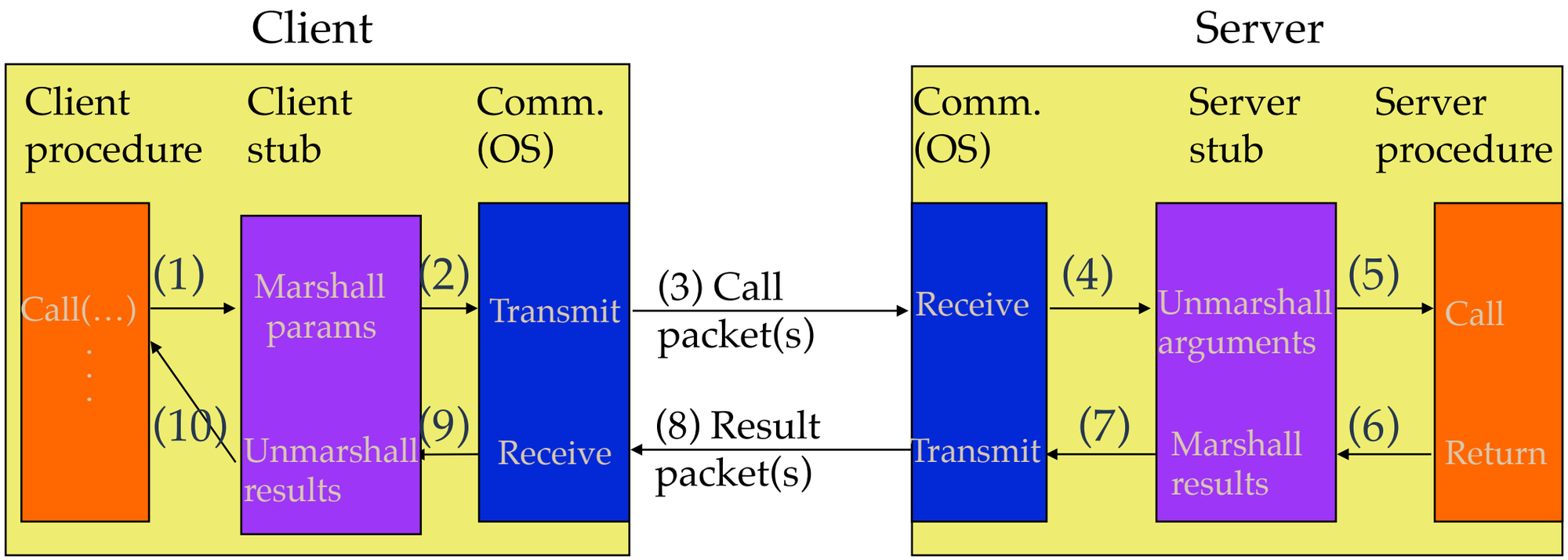
Asynchronous RPC interaction

Asynchronous RPC (2)



A client and server interacting through two asynchronous RPCs

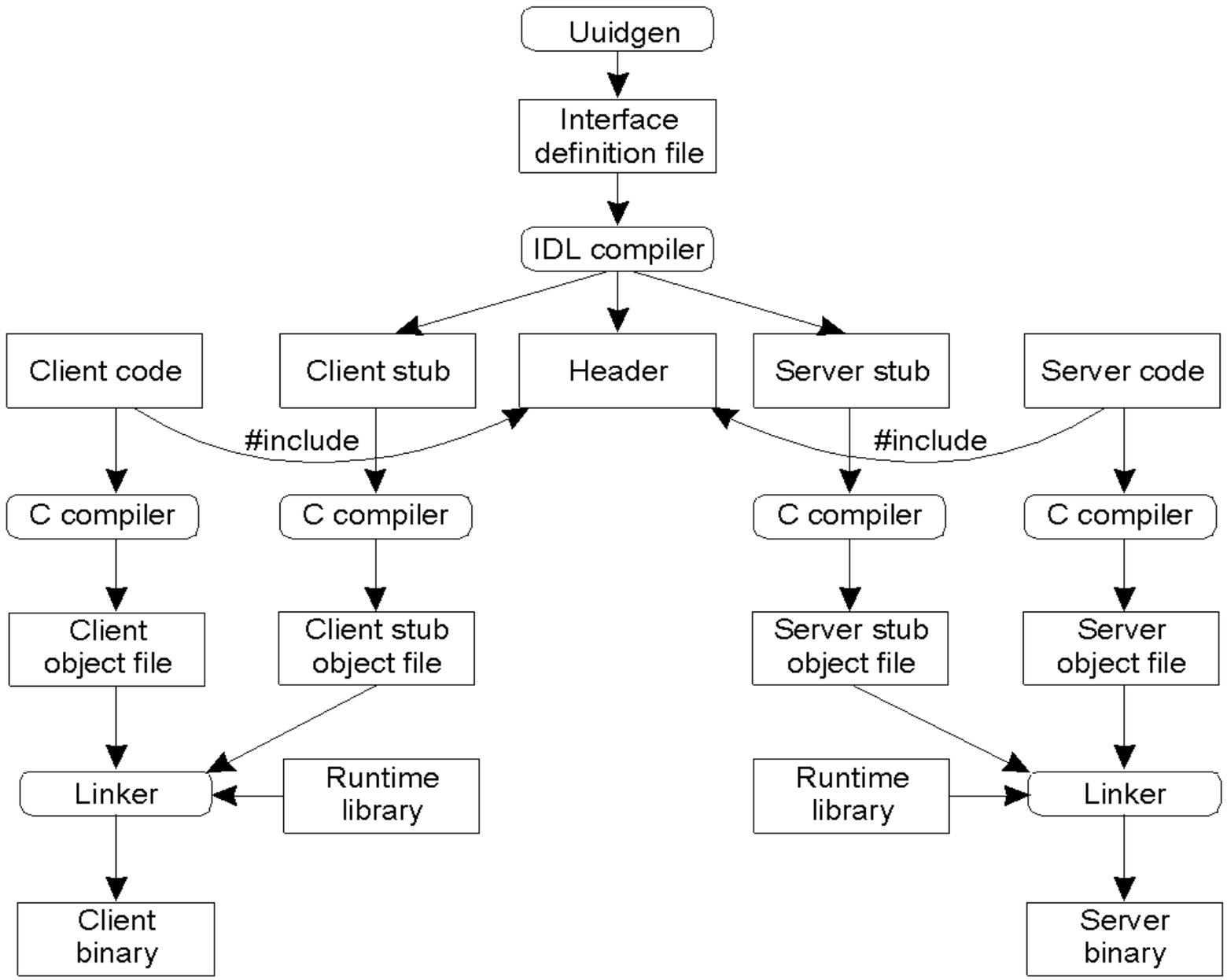
Where do stub procedures come from?



- In many systems, stub procedures are generated automatically by a special stub compiler
- Given a specification of the server procedure and the encoding rules, the message format is uniquely determined
- It is, thus, possible to have a compiler read the server specification and generate a client stub that packs its parameters into the officially approved message format
- Similarly, the compiler can also produce a server stub that unpacks the parameters and calls the server

Having both stub procedures generated from a single formal spec. of the server not only make life easier for the programmers, but reduces the chance of error and makes the system transparent with respect to differences in internal representation of data.

Writing Clients/Servers (using DCE RPC)



From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2nd edition

How does the client locate the server?

- Hardwire the server address into the client
  - ➡ Fast but inflexible!
- A better method is dynamic binding:
  - ➡ When the server begins executing, the call to initialize outside the main loop exports the server interface
  - ➡ This means that the server sends a message to a program called a binder, to make its existence known. This process is referred to as registering
  - ➡ To register, the server gives the binder its name, its version number, a unique identifier (32-bits), and a handle used to locate it
  - ➡ The handle is system dependent (e.g., Ethernet address, IP address, an X.500 address, …)

Binder interface

Call	Input	Output
Register	Name, version, handle, unique id	
Deregister	Name, version, unique id	

Dynamic Binding

- When the client calls one of the remote procedures for the first time, say, read:
  - ➡ The client stub sees that it is not yet bound to a server, so it sends a message to the binder asking to import version x of server interface
  - ➡ The binder checks to see if one or more servers have already exported an interface with this name and version number.
    - ✦ If no currently running server is willing to support this interface, the read call fails
    - ✦ If a suitable server exists, the binder gives its handle and unique identifier to the client stub
  - ➡ Client stub uses the handle as the address to send the request message to. The message contains the parameters and a unique identifier, which the server's kernel uses to direct incoming message to the correct server

Binder interface	Call	Input	Output
	Look up	Name, version	Handle, unique id

Dynamic Binding (2)

- Advantages
  - Flexibility
  - Can support multiple servers that support the same interface, e.g.:
    - Binder can spread the clients randomly over the servers to even load
    - Binder can poll the servers periodically, automatically deregistering the servers that fail to respond, to achieve a degree of fault tolerance
    - Binder can assist in authentication: For example, a server specifies a list of users that can use it; the binder will refuse to tell users not on the list about the server
  - The binder can verify that both client and server are using the same version of the interface
- Disadvantages
  - The extra overhead of exporting/importing interfaces costs time
  - The binder may become a bottleneck in a large distributed system

Message-Based Communication

- Lower-level interface to provide more flexibility
- Two (abstract) primitives are used to implement these
  - ➡ send
  - ➡ receive
- Issues:
  - ➡ Are primitives blocking or nonblocking (synchronous or asynchronous)?
  - ➡ Are primitives reliable or unreliable (persistent or transient)?

Synchronous/Asynchronous Messaging

- Synchronous
  - ➡ The sender is blocked until its message is stored in the local buffer at the receiving host or delivered to the receiver.
- Asynchronous
  - ➡ The sender continues immediately after executing a send
  - ➡ The message is stored in the local buffer at the sending host or at the first communication server.

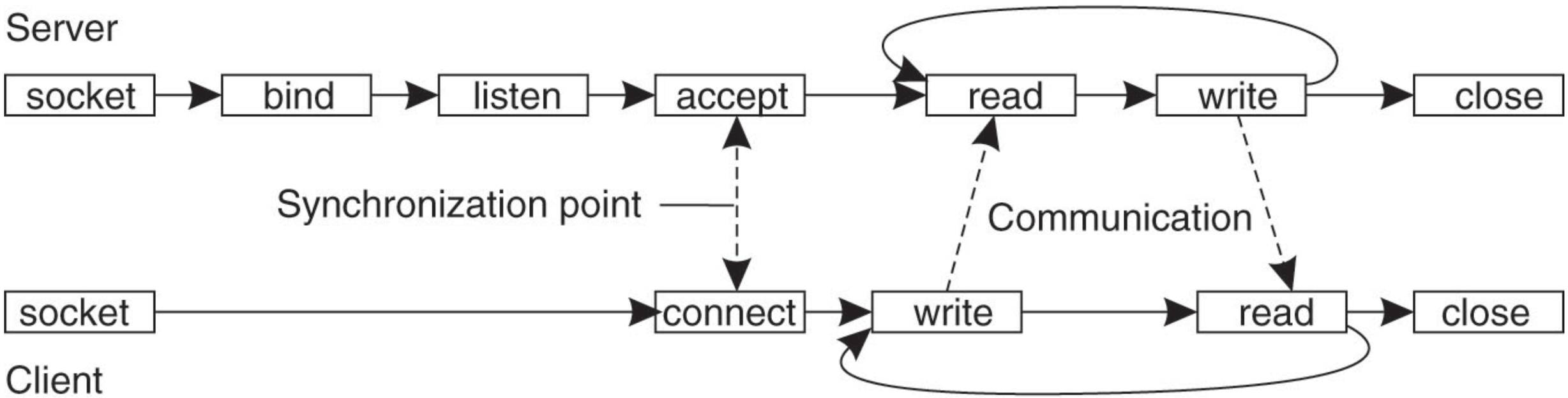
Transient/Persistent Messaging

- Transient
  - The sender puts the message on the net and if it cannot be delivered to the sender or to the next communication host, it is lost.
  - There can be different types depending on whether it is asynchronous or synchronous
- Persistent
  - The message is stored in the communication system as long as it takes to deliver the message to the receiver

Socket Primitives for TCP/IP

Primitive	Meaning
Socket	Create a new communication endpoint
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

Connection-Oriented Communication Using Sockets



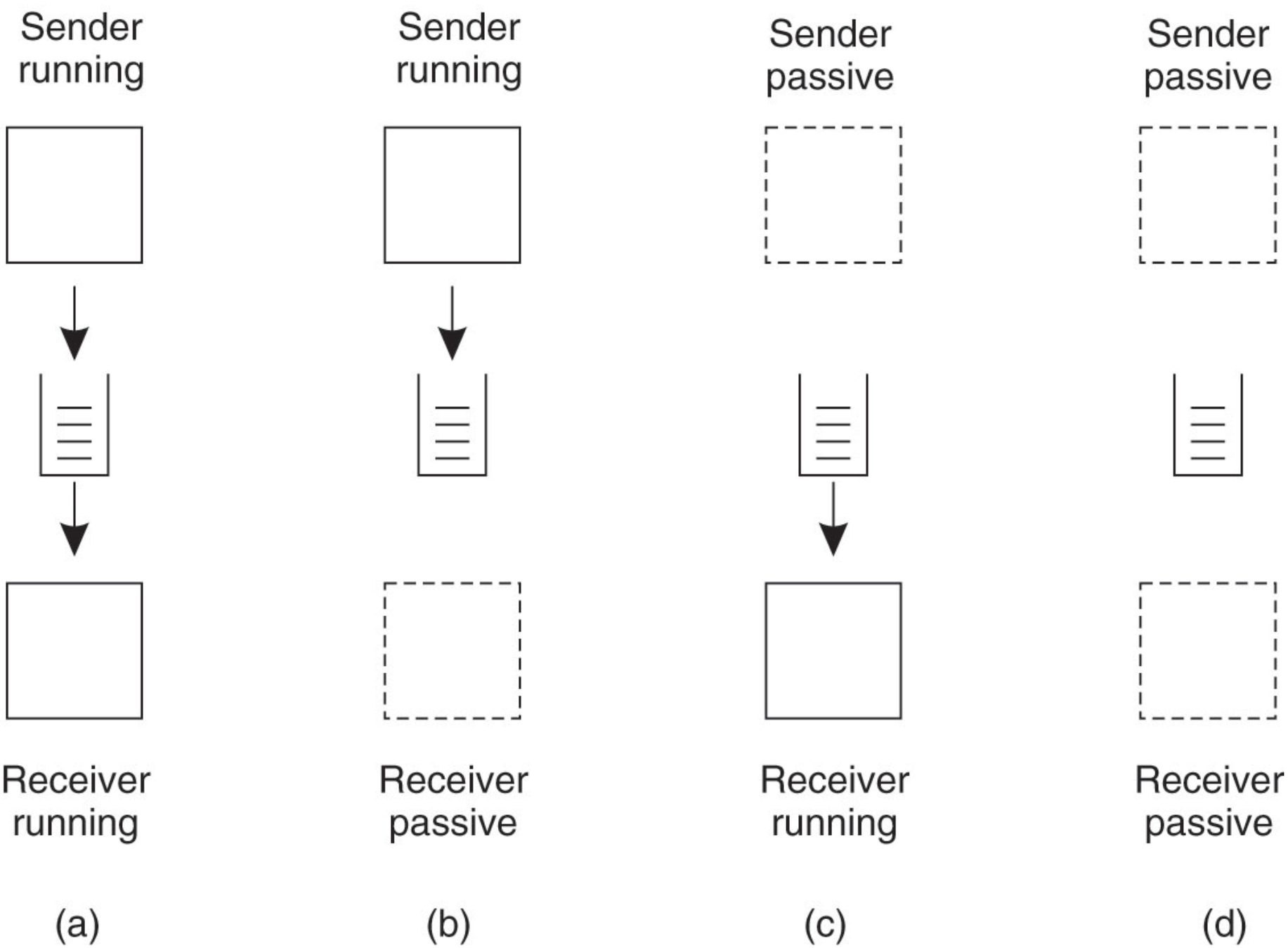
From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2nd edition

Persistent Messaging

- Usually called message-queuing system, since it involves queues at both ends
  - ➡ Sender application has a queue
  - ➡ Receiver application has a queue
  - ➡ Receiver does not have to be active when sender puts a message into sender queue
  - ➡ Sender does not have to be active when receiver picks up a message from its queue
- Basic interface:

Primitive	Meaning
Put	Append a message to a specified queue
Get	Block until the specified queue is nonempty, and remove the first message
Poll	Check a specified queue for messages, and remove the first. Never block
Notify	Install a handler to be called when a message is put into the specified queue

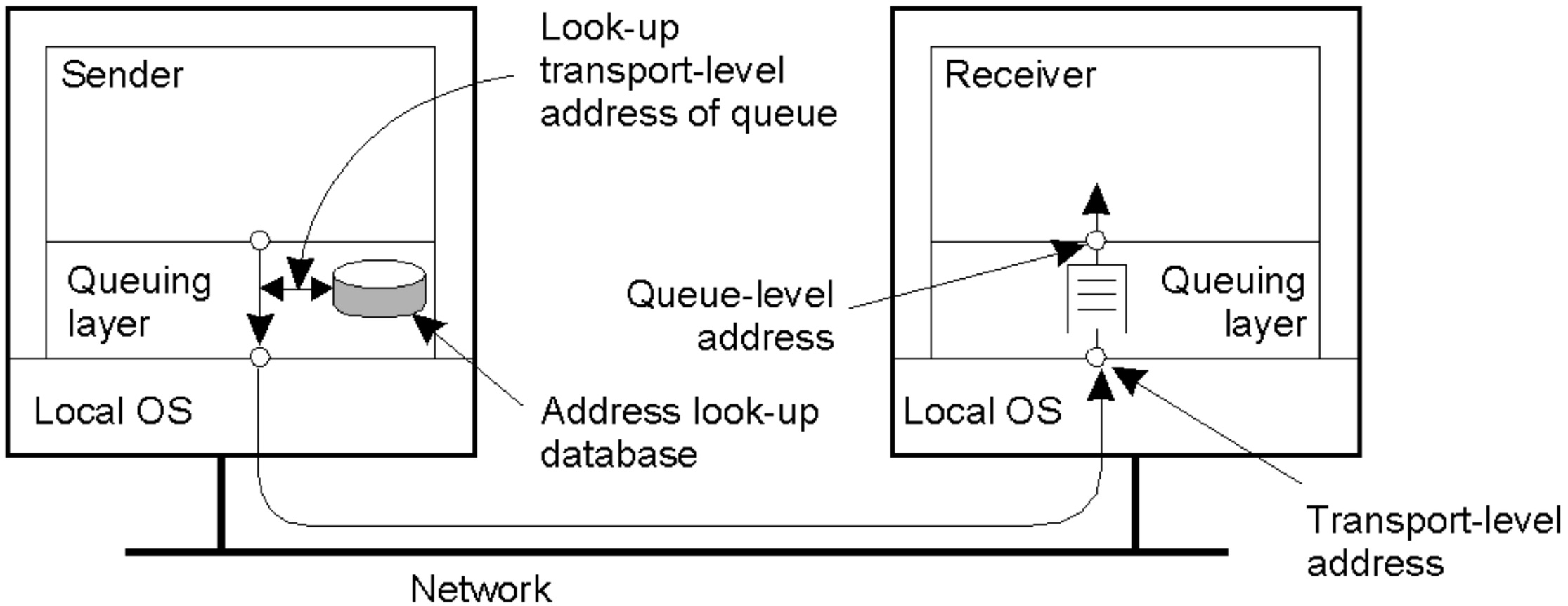
Different Message Queuing Models



From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2nd edition

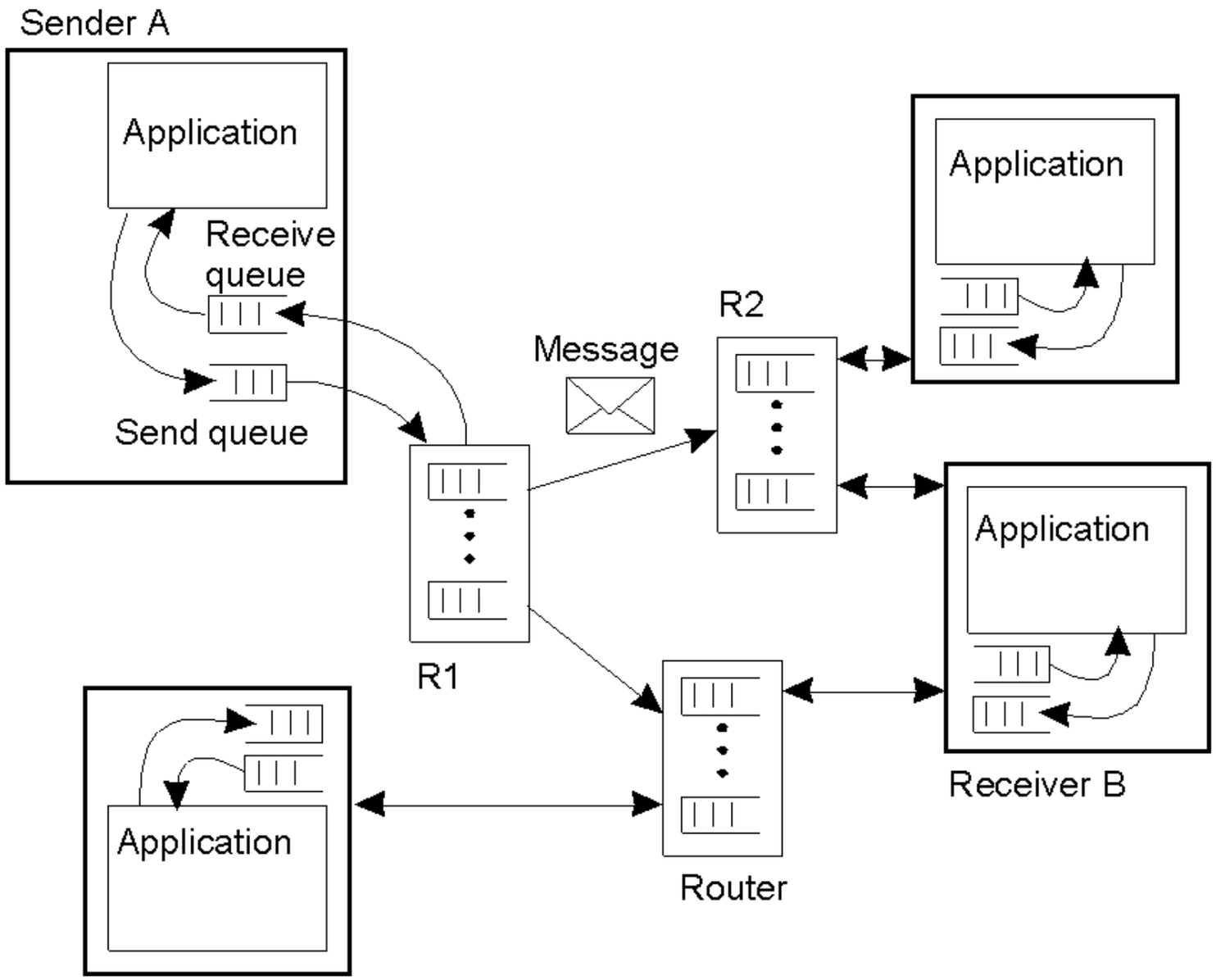
General Architecture of a Message-Queuing System (1)

- The relationship between queue-level addressing and network-level addressing.



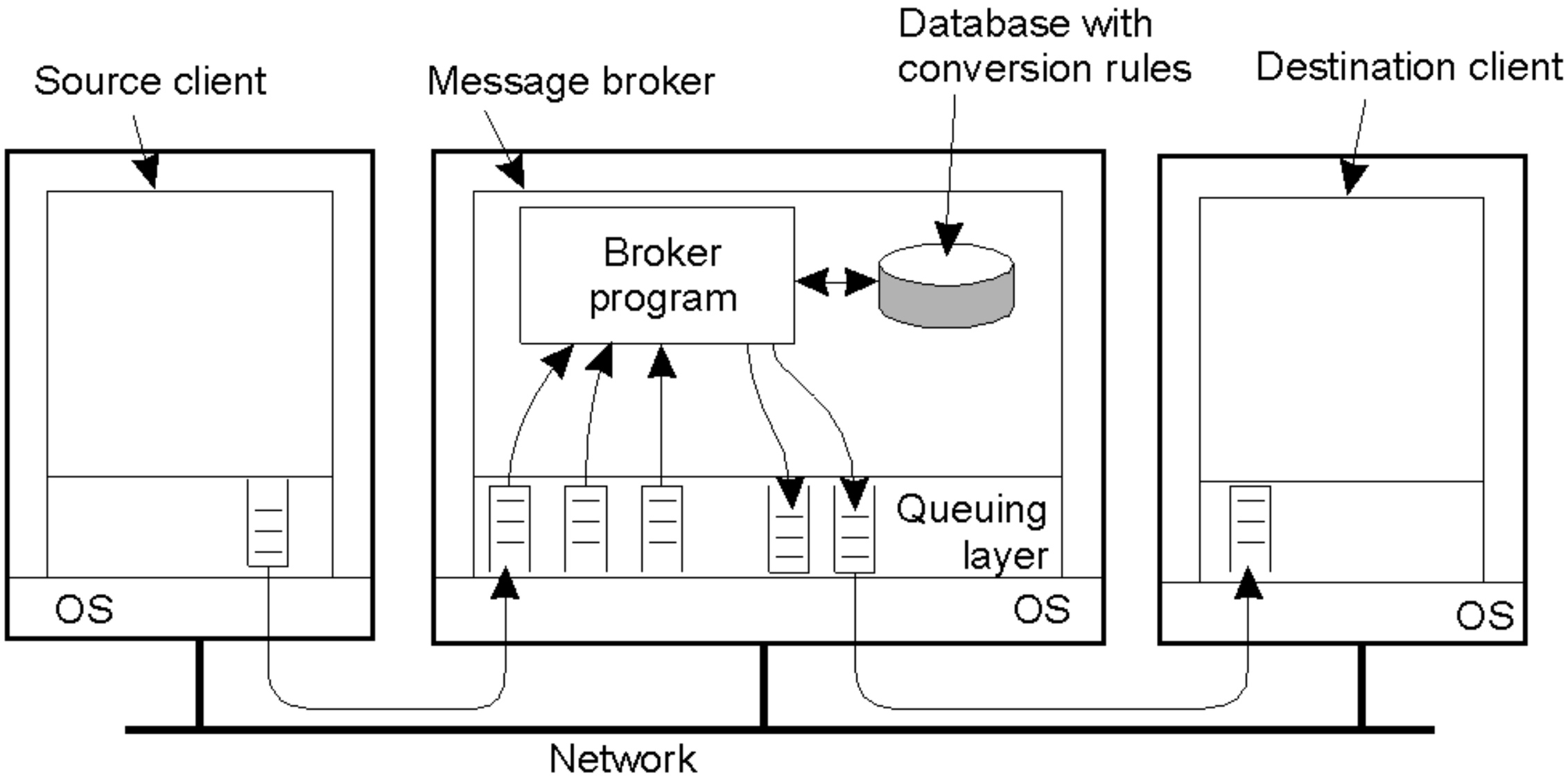
General Architecture of a Message-Queuing System (2)

- The general organization of a message-queuing system with routers.



From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2nd edition

Message Brokers

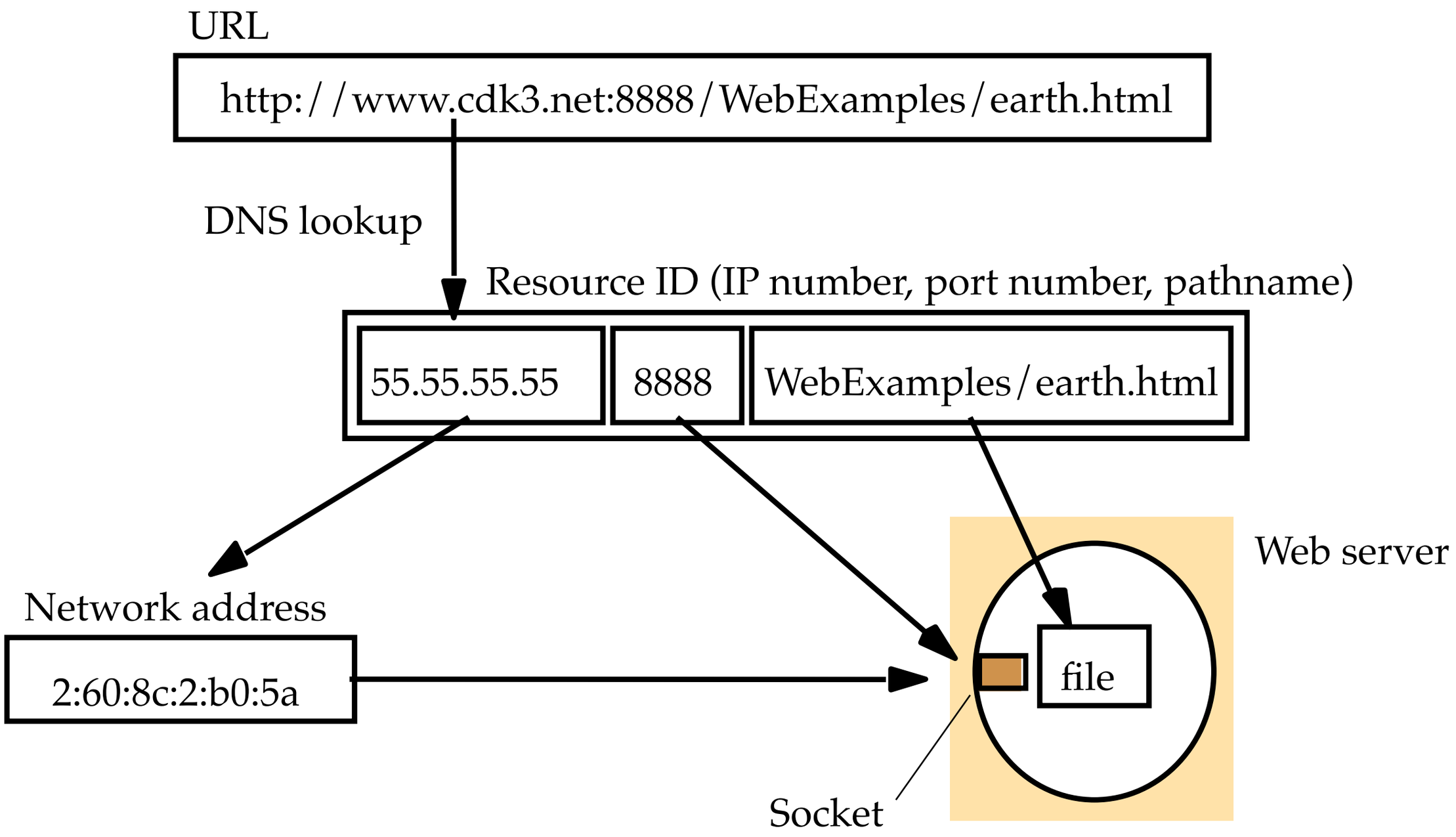


Distributed Naming

Names, Addresses, etc.

- Names
  - Identification of objects
    - Resource sharing: Internet domain names
    - Communication: domain name part of email address
  - How much information about an object is in a name?
    - Pure names: uninterpreted bit patterns
    - Non-pure names: contain information about the object, e. g., its location
- Name Services
  - Entries of the form <name, attributes>, where attributes are typically network addresses
  - Type of lookup queries
    - name ➔ attribute values
    - also called name resolution
- Directory Services
  - <name, attributes> entries
  - Types of lookup queries
    - name ➔ attribute values
    - attribute values ➔names

Composed Naming Domains



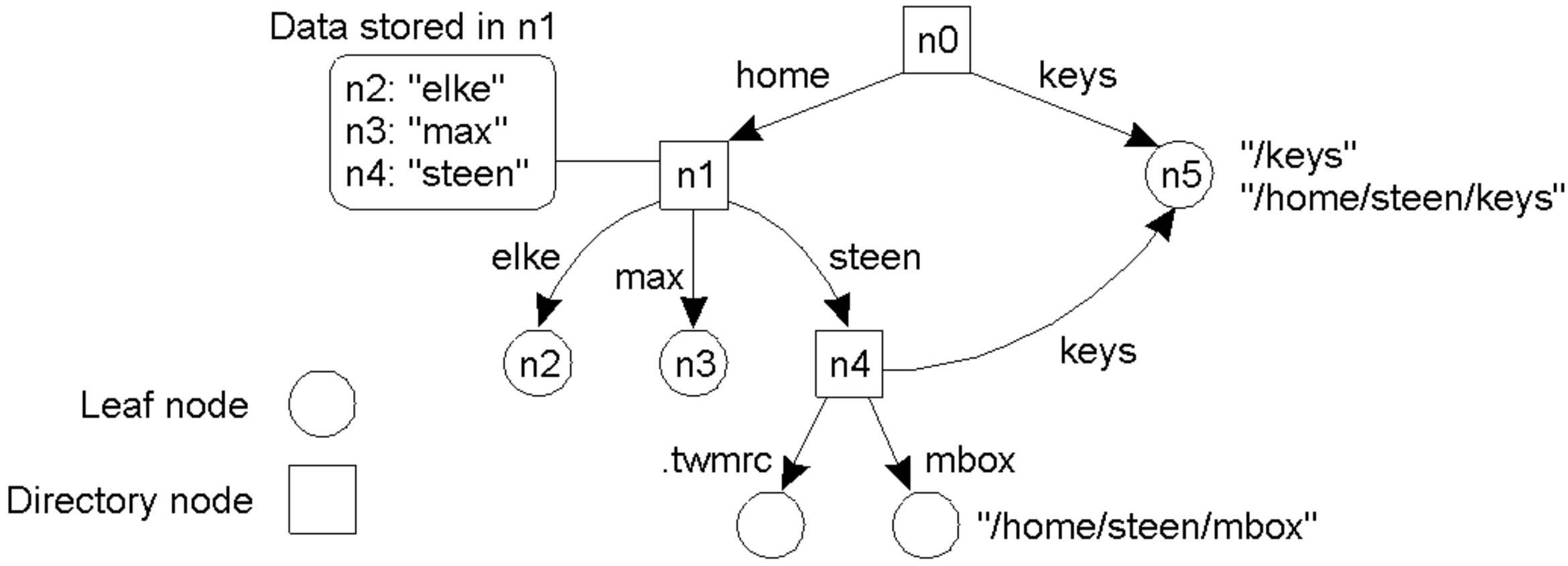
Requirements on Name Services

- Usage of a convention for unique global naming
  - ➡ Enables sharing
  - ➡ It is often not easily predictable which services will eventually share resources
- Scalability
  - ➡ Naming directories tend to grow very fast, in particular in Internet
- Consistency
  - ➡ Short and mid-term inconsistencies tolerable
  - ➡ In the long term, system should converge towards a consistent state
- Performance and availability
  - ➡ Speed and availability of lookup operations
  - ➡ Name services are at the heart of many distributed applications
- Adaptability to change
  - ➡ Organizations frequently change structure during lifetime
- Fault isolation
  - ➡ System should tolerate failure of some of its servers

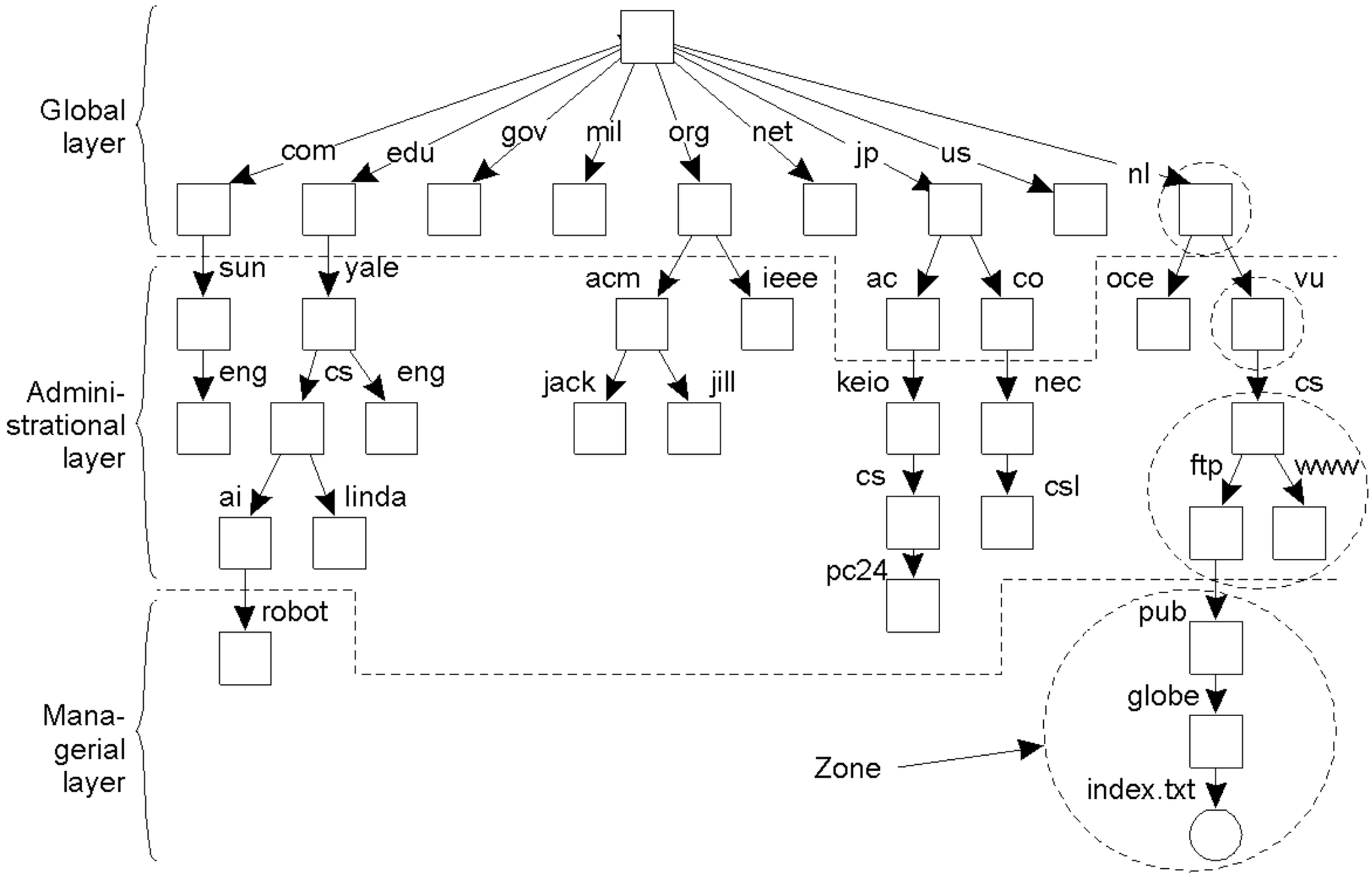
Name Spaces

- Set of all valid names to be used in a certain context, e. g., all valid URLs in WWW
- Can be described using a generative grammar (e. g., BNF for URLs)
- Internal structure
  - ➡ Flat set of numeric or symbolic identifiers
  - ➡ Hierarchy representing position (e. g., UNIX file system)
  - ➡ Hierarchy representing organizational structure (e. g., Internet domains)
- Potentially infinite
  - ➡ Holds only for hierarchic name spaces
  - ➡ Flat name spaces finite size induced by max. name length
- Aliases
  - ➡ In general, allows a convenient name to be substituted for a more complicated one
- Naming domain
  - ➡ Name space for which there exist a single administrative authority for assigning names within it

Naming Graph with Single Root



Name Space Distribution (1)



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

Name Space Distribution (2)

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

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

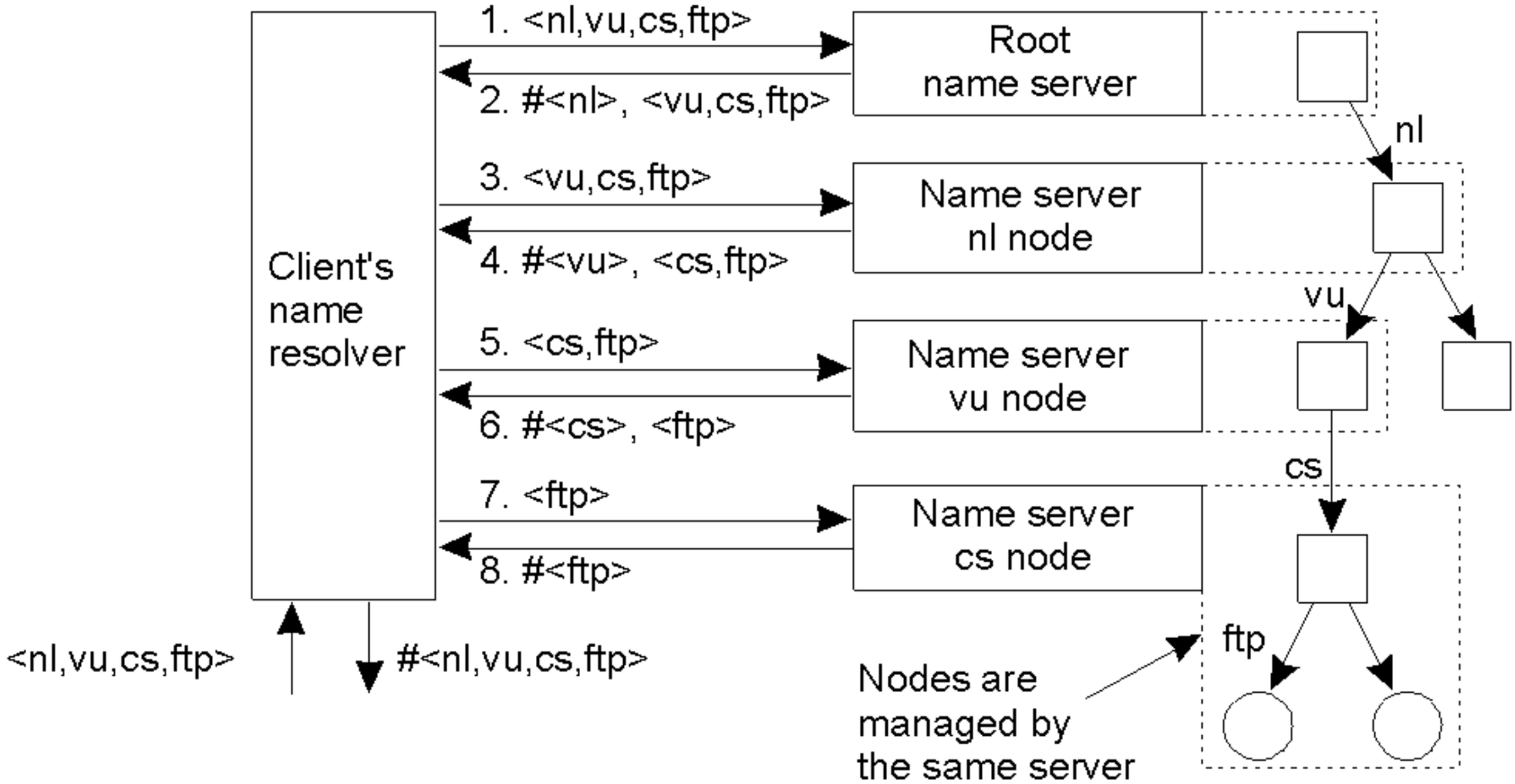
Issues in Name & Directory Services

- Partitioning
  - No one name server can hold all name and attribute entries for entire network, in particular in Internet
  - Name server data partitioned according to domain
- Replication
  - A domain has usually more than one name server
  - Availability and performance are enhanced
- Caching
  - Servers may cache name resolutions performed on other servers
    - Avoids repeatedly contacting the same name server to look up identical names
  - Client lookup software may equally cache results of previous requests

Name Resolution

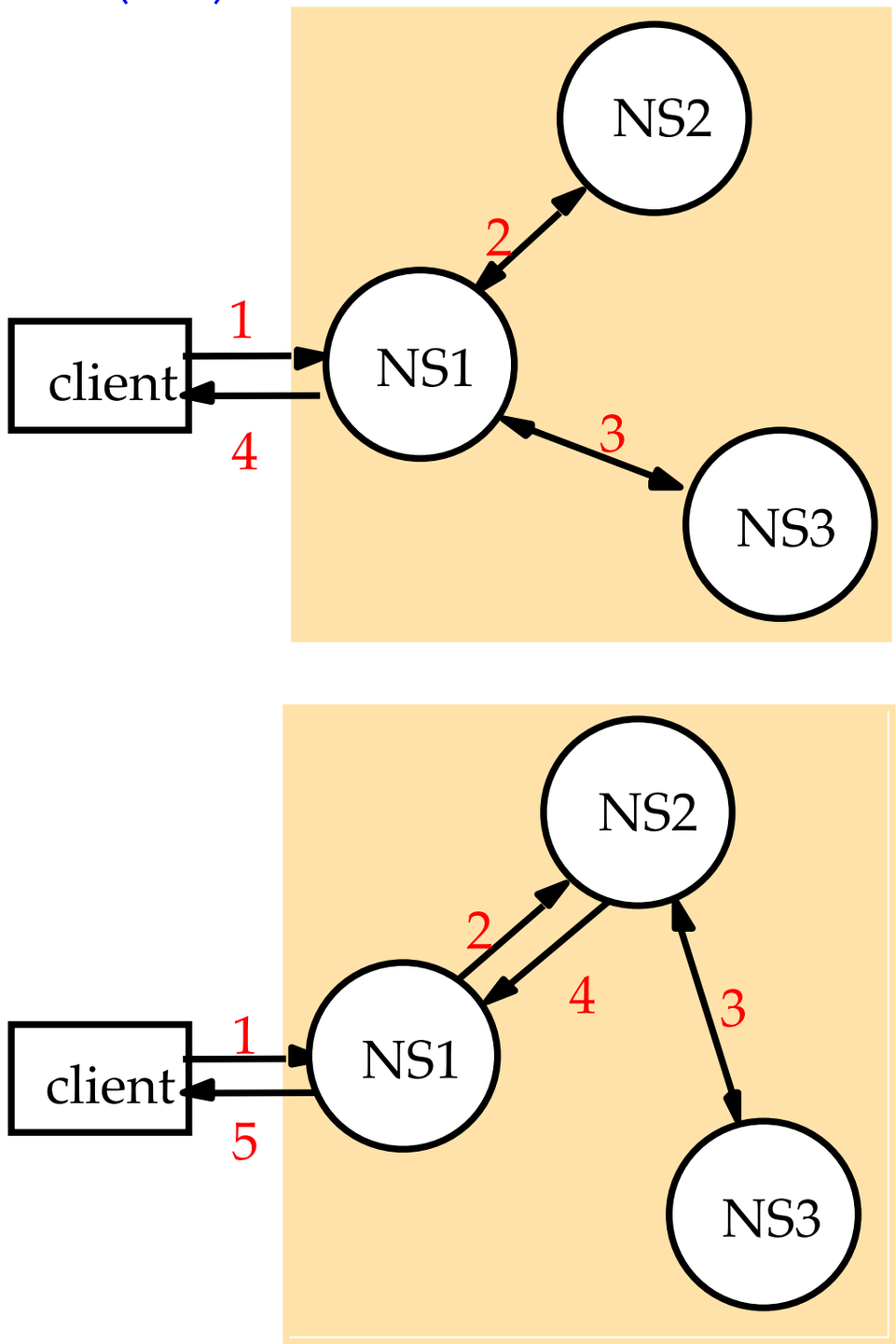
- Translation of a name into the related primitive attribute
- Often, an iterative process
  - ➡ Name service returns attributes if the resolution can be performed in it's naming context
  - ➡ Name service refers query to another context if name can't be resolved in own context
- Deal with cyclic alias references, if present
  - ➡ Abort resolution after a predefined number of attempts, if no result obtained

Iterative Navigation Example

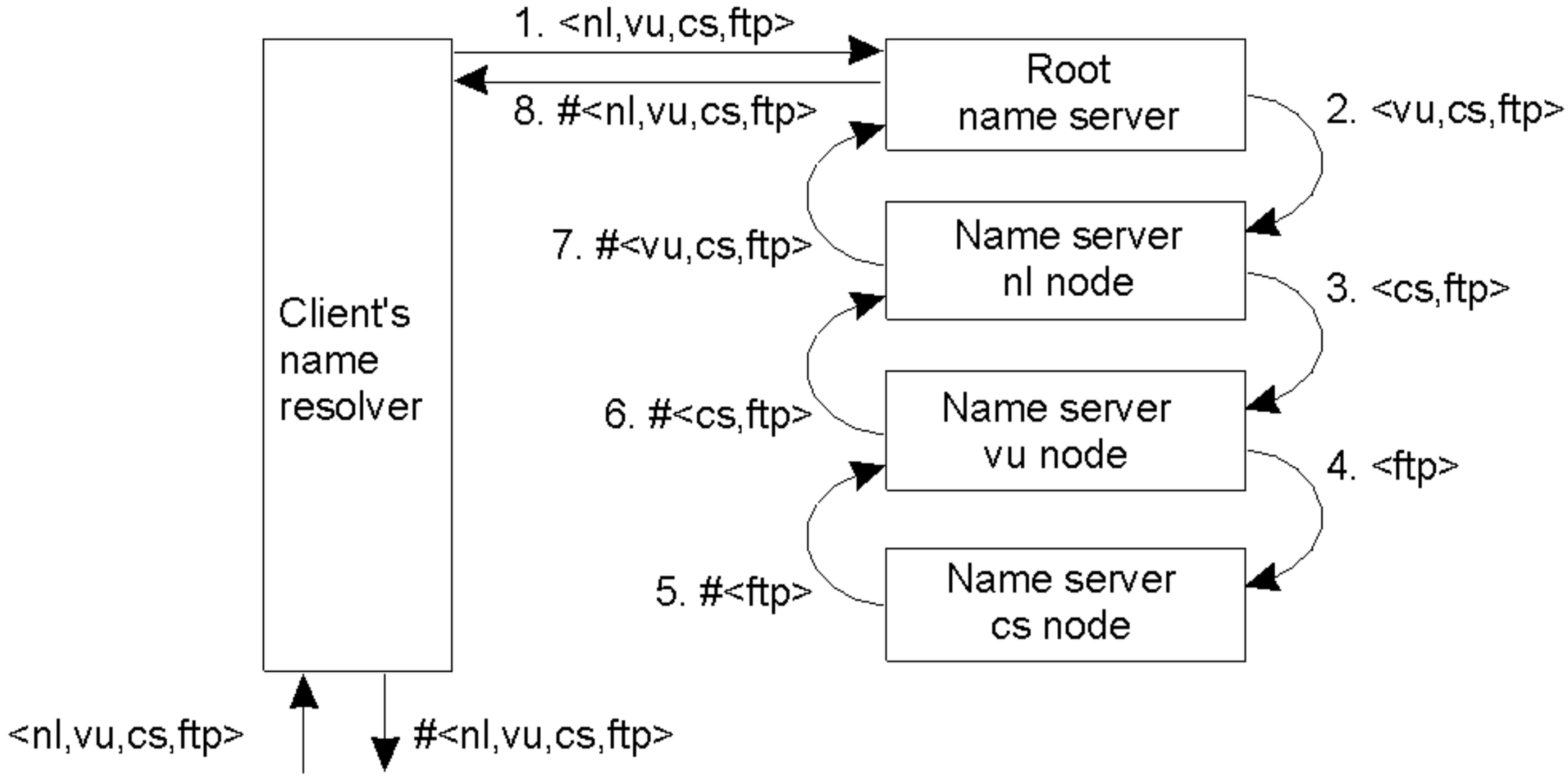


Navigation (2)

- Non-recursive, server-controlled
  - ➡ Server contacts peers if it cannot resolve name itself
    - ✦ by multicast or iteratively by direct contact
- Recursive, server-controlled
  - ➡ If name cannot be resolved, server contacts superior server responsible for a larger prefix of the name space
    - ✦ recursively applied until name resolved
    - ✦ can be used when clients and low-level servers are not entitled to directly contact high-level servers



Recursive Navigation Example

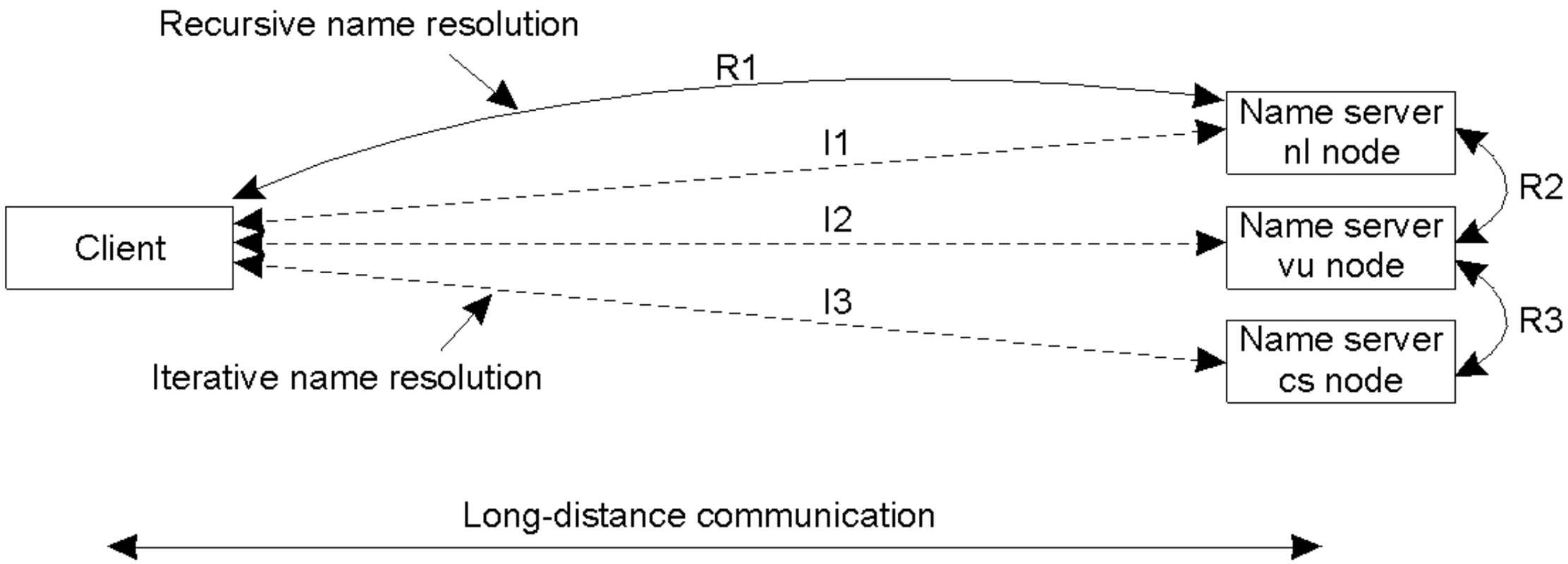


Recursive Name Resolution

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>

- Recursive name resolution of *<nl, vu, cs, ftp>*. Name servers cache intermediate results for subsequent lookups.

Communication Cost Comparison of Iterative vs Recursive



Name & Directory Services

- Domain Name System (DNS)
  - ➡ Name service used across the Internet
- X. 500
  - ➡ ITU - standardized directory service
- LDAP
  - ➡ Directory service
  - ➡ Lightweight implementation of X.500
  - ➡ Often used in intranets
- Jini
  - ➡ Discovery service used in spontaneous networking
  - ➡ Contains directory service component

Domain Name System (DNS)

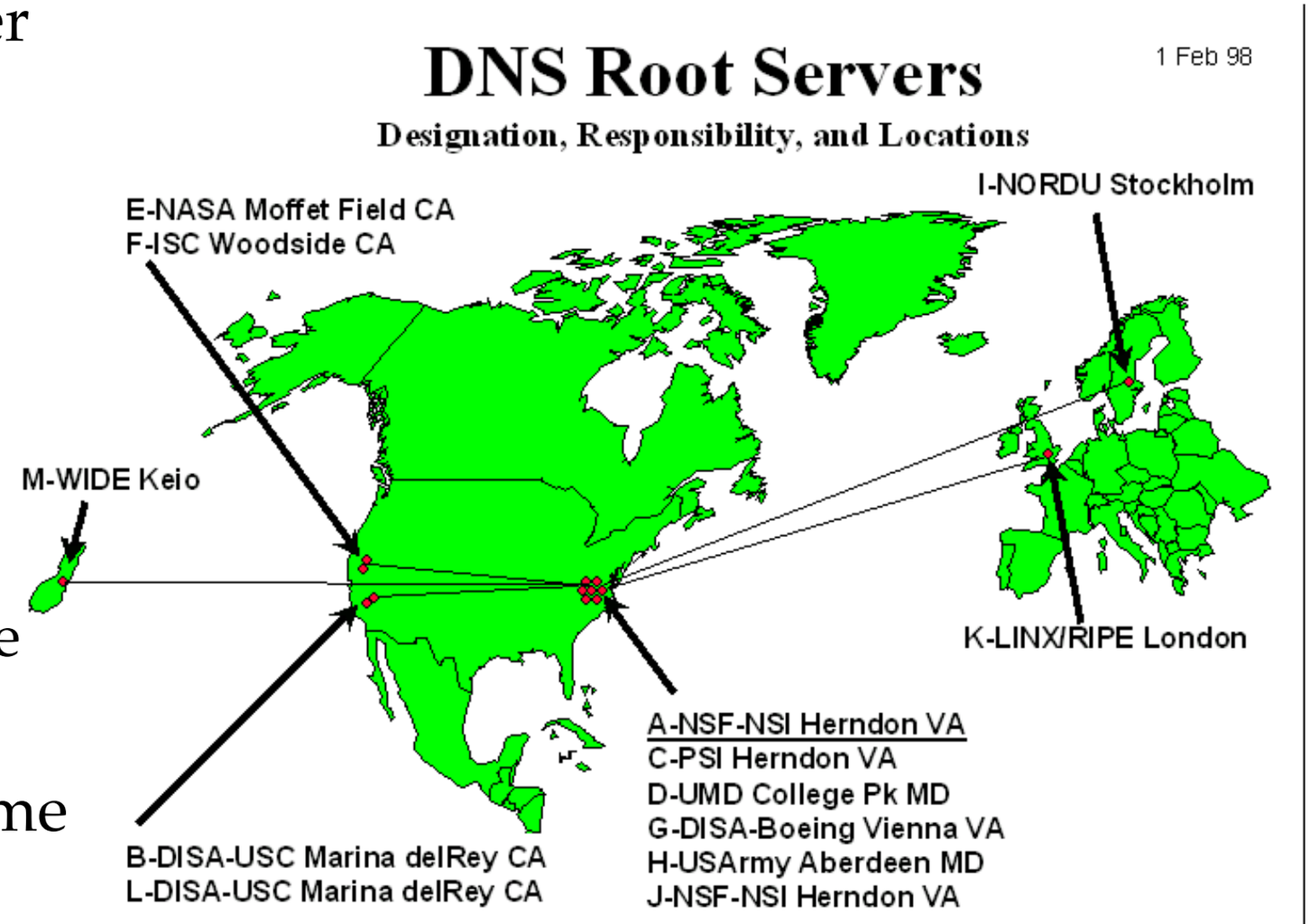
- Performs name-to-IP mapping
- *Distributed database*
  - ➡ implemented in hierarchy of many name servers
- *Application-layer protocol*
  - ➡ host, routers, name servers to communicate to resolve names (address/name translation)
- Why not centralize DNS?
  - ➡ single point of failure
  - ➡ traffic volume
  - ➡ distant centralized database
  - ➡ maintenance
  - ➡ doesn't scale!

DNS Types

- No server has all name-to-IP address mappings
- Local name servers
  - ➡ Each ISP, company has local (default) name server
  - ➡ Host DNS query first goes to local name server
- Authoritative name server
  - ➡ For a host: stores that host's IP address, name
  - ➡ Can perform name-to-address translation for that host's name
- Root name server
  - ➡ Establishes a rooted tree of DNSs

DNS: Root Name Servers

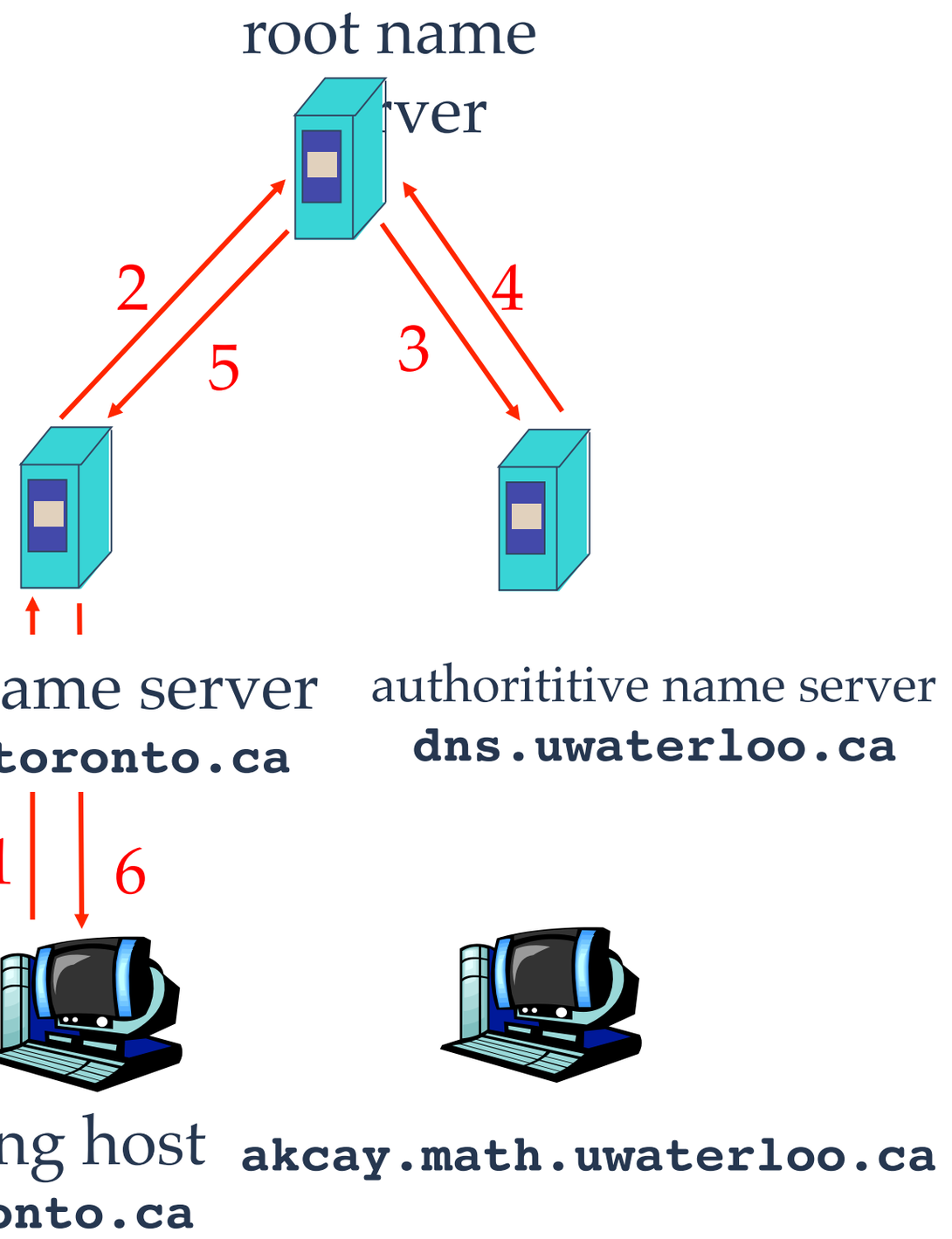
- Contacted by local name server that can not resolve name
- Root name server:
  - ➡ contacts authoritative name server if name mapping not known
  - ➡ gets mapping
  - ➡ returns mapping to local name server
- Approximately dozen root name servers worldwide



Simple DNS Example

Host **`db.utoronto.ca`** wants IP address of **`akcay.math.uwaterloo.ca`**

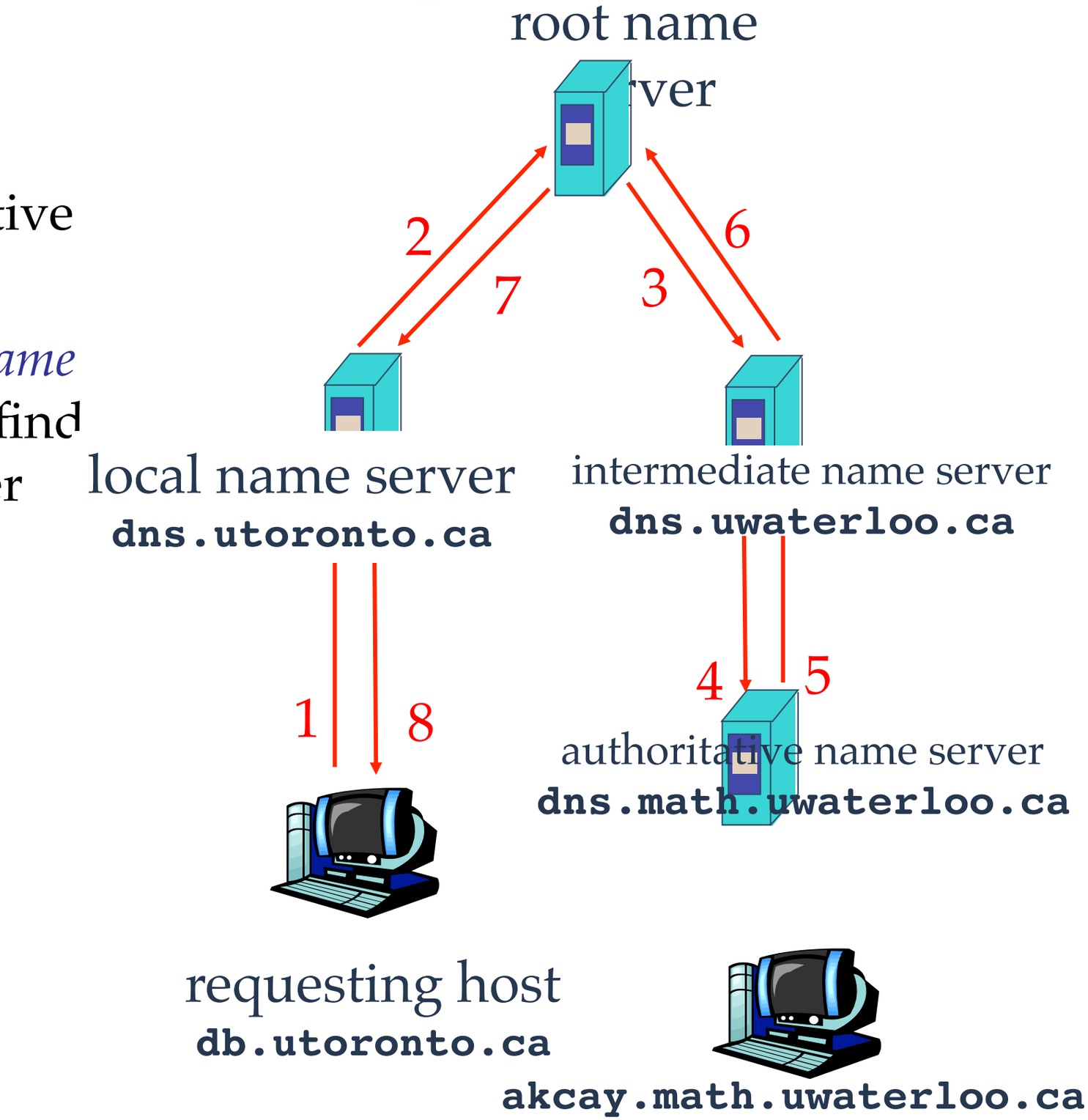
1. Contacts its local DNS server, **`dns.utoronto.ca`**
2. **`dns.utoronto.ca`** contacts root name server, if necessary
3. root name server contacts authoritative name server, **`dns.uwaterloo.ca`,** if necessary



DNS Example

Root name server:

- may not know authoritative name server
- may know *intermediate name server:* who to contact to find authoritative name server



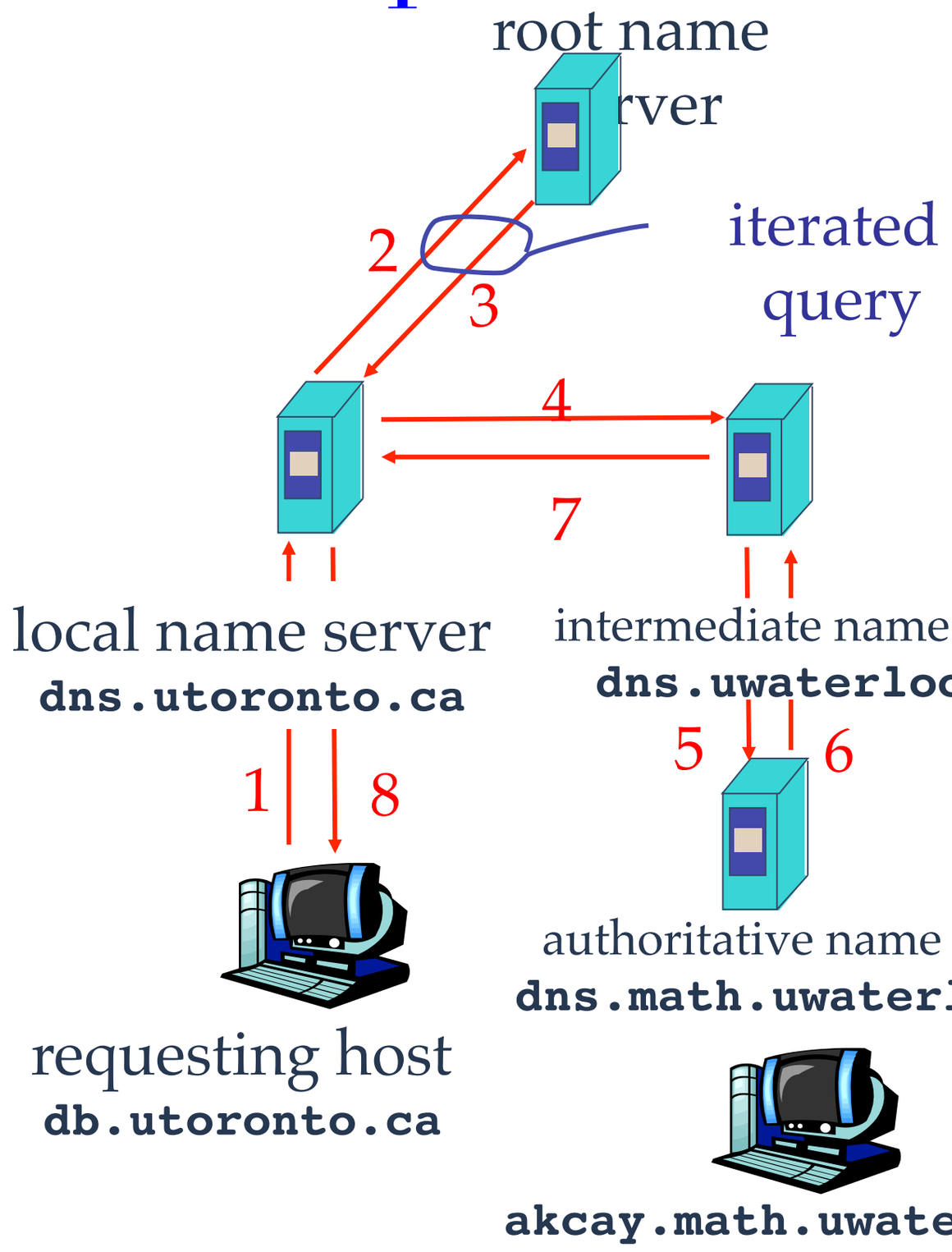
DNS: iterated queries

Recursive query:

- Puts burden of name resolution on contacted name server
- Heavy load?

Iterated query:

- Contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



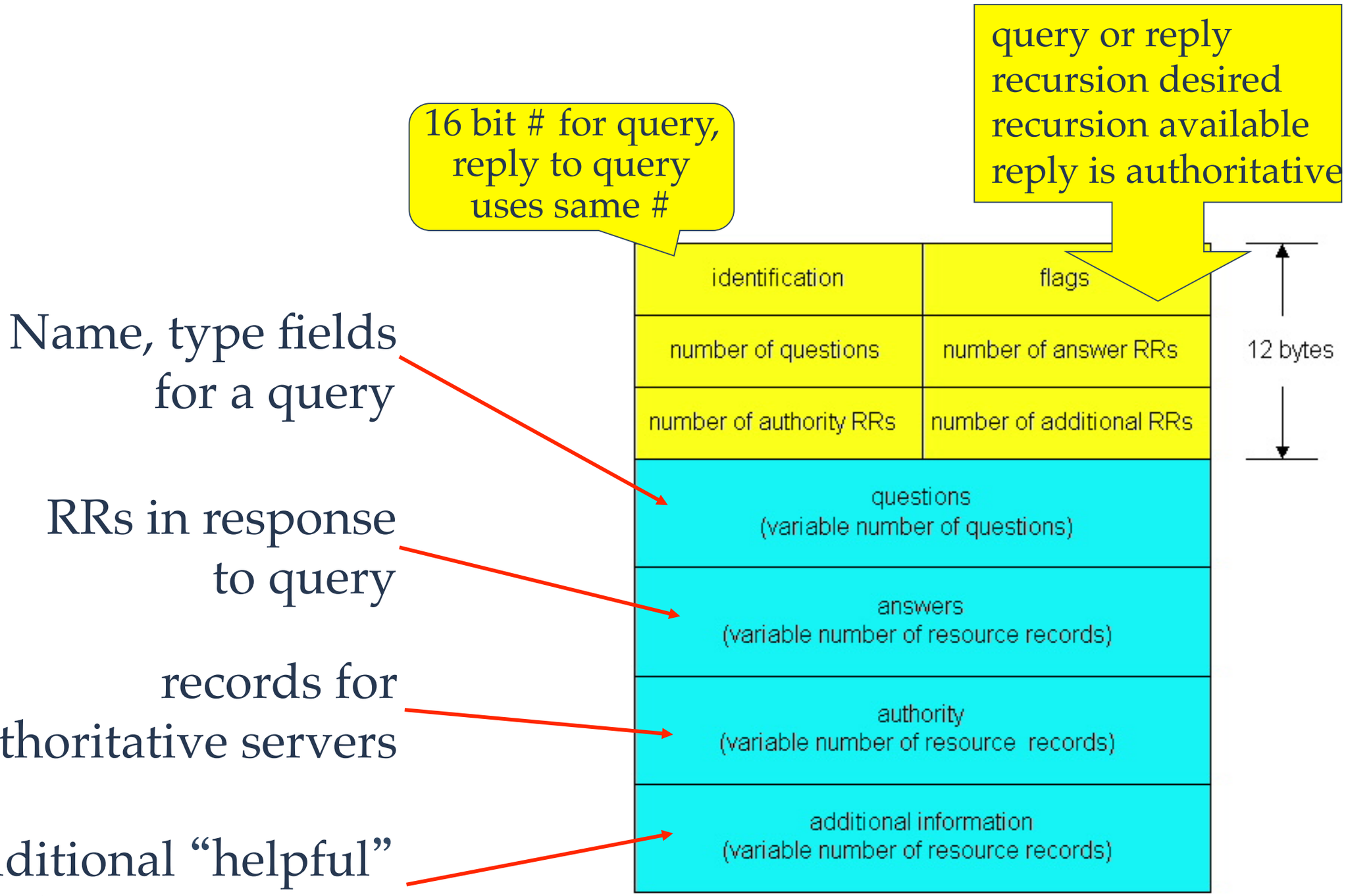
DNS Records

- DNS holds resource records (RR)
  - RR format: **`(name, value, type,ttl)`**
- Example records:
  - Type=A
    - **`name`** is hostname
    - **`value`** is IP address
  - Type=NS
    - **`name`** is domain (e.g. foo.com)
    - **`value`** is IP address of authoritative name server for this domain
  - Type=CNAME
    - **`name`** is an alias name for some "canonical" (the real) name
    - **`value`** is canonical name
  - Type=MX
    - **`value`** is hostname of mail server associated with **`name`**

DNS Record Types

Record type	*Meaning*	*Main contents*
A	A computer address	IP number
NS	An authoritative name server	Domain name for server
CNAME	The canonical name for an alias	Domain name for alias
SOA	Marks the start of data for a zone	Parameters governing the zone
WKS	A well-known service description	List of service names and protocols
PTR	Domain name pointer (reverse lookups)	Domain name
HINFO	Host information	Machine architecture and operating system
MX	Mail exchange	List of *preference, host* pairs
TXT	Text string	Arbitrary text

DNS Protocol & Messages



Discovery Services

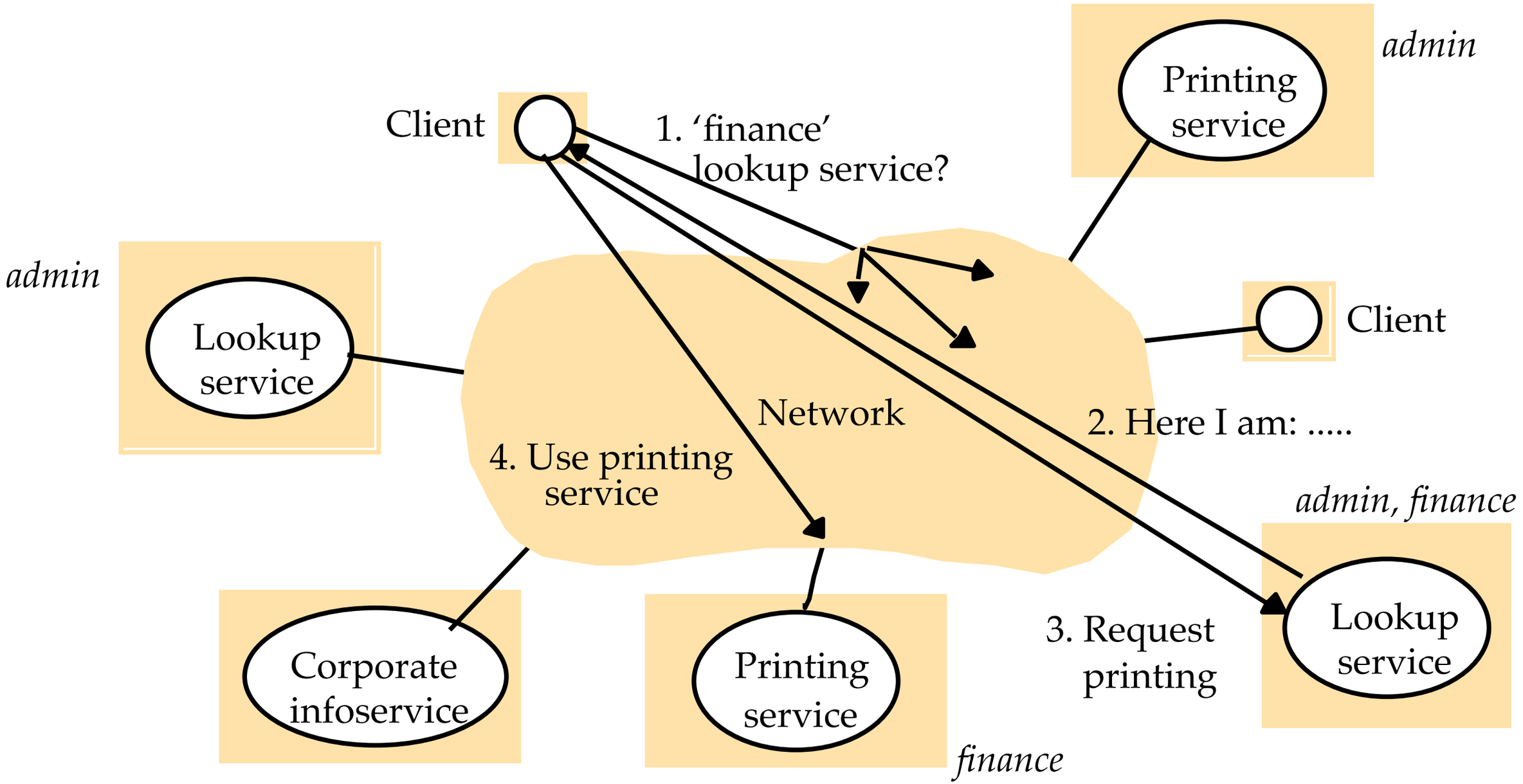
- Directory services that allow clients to query available services in a spontaneous networking environment
  - e. g., which are the available color printers
  - services enter their data using registration interfaces
  - structure usually rather flat, since scope limited to (wireless) LAN
- JINI (http://www.sun.com/jini/)
  - JAVA-based discovery service
    - clients and servers run JVMs
      - communication via Java RMI
      - dynamic loading of code

JINI Discovery Related Services

- Lookup service
  - ➡ Holds information regarding available services
- Query of lookup service by Jini client
  - ➡ Match request
  - ➡ Download object providing service from lookup service
- Registration of a Jini client or Jini service with lookup service
  - ➡ Send message to well-known IP multicast address, identical to all Jini instances
  - ➡ Limit multicast to LAN using time-to-live attribute
  - ➡ Use of leases that need to be renewed periodically for registering Jini services

JINI Discovery Scenario

- new client wishes to print on a printer belonging to the `finance` group



From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.