Architectures

Distributed and Web Computing

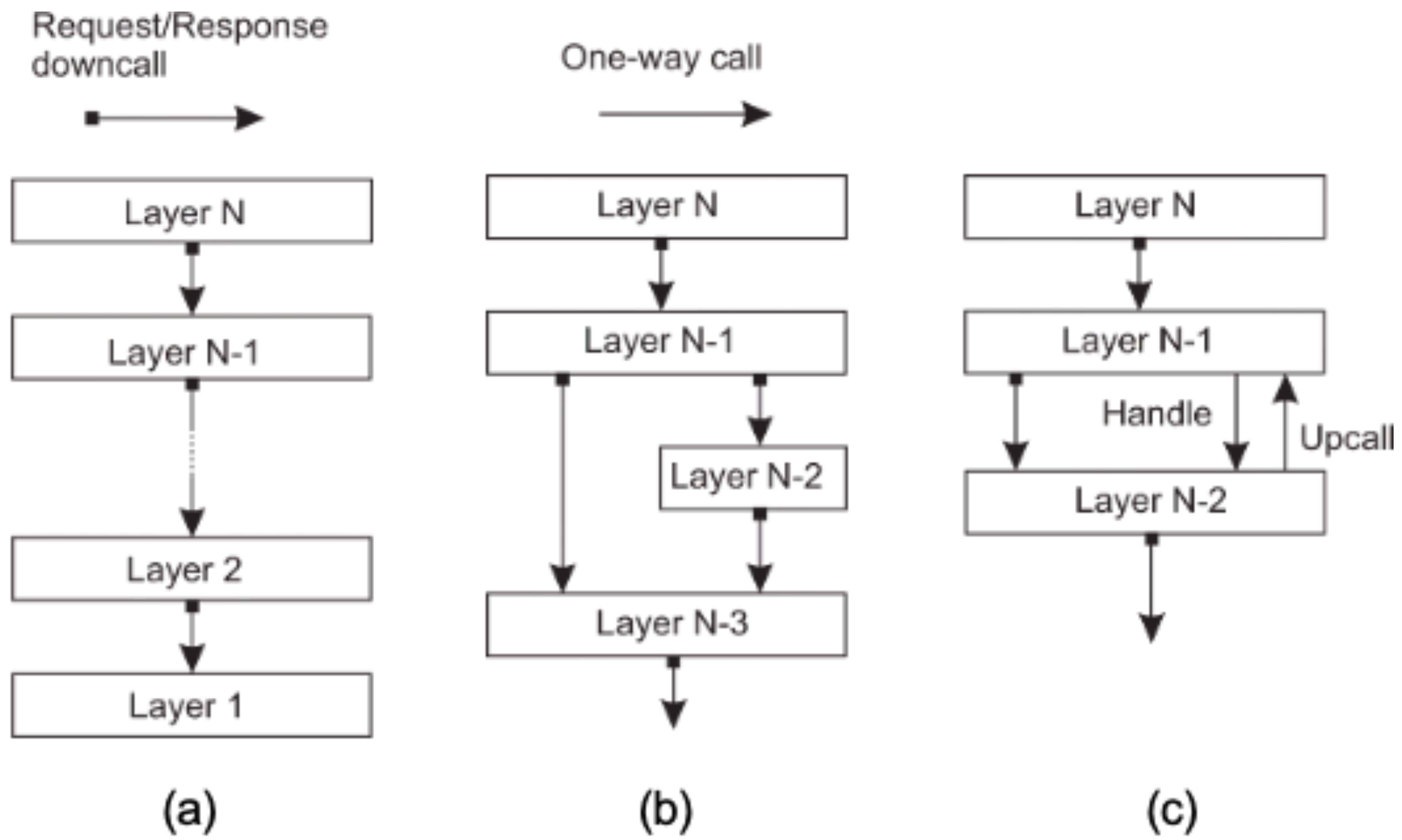
Architectural Styles

Basic idea

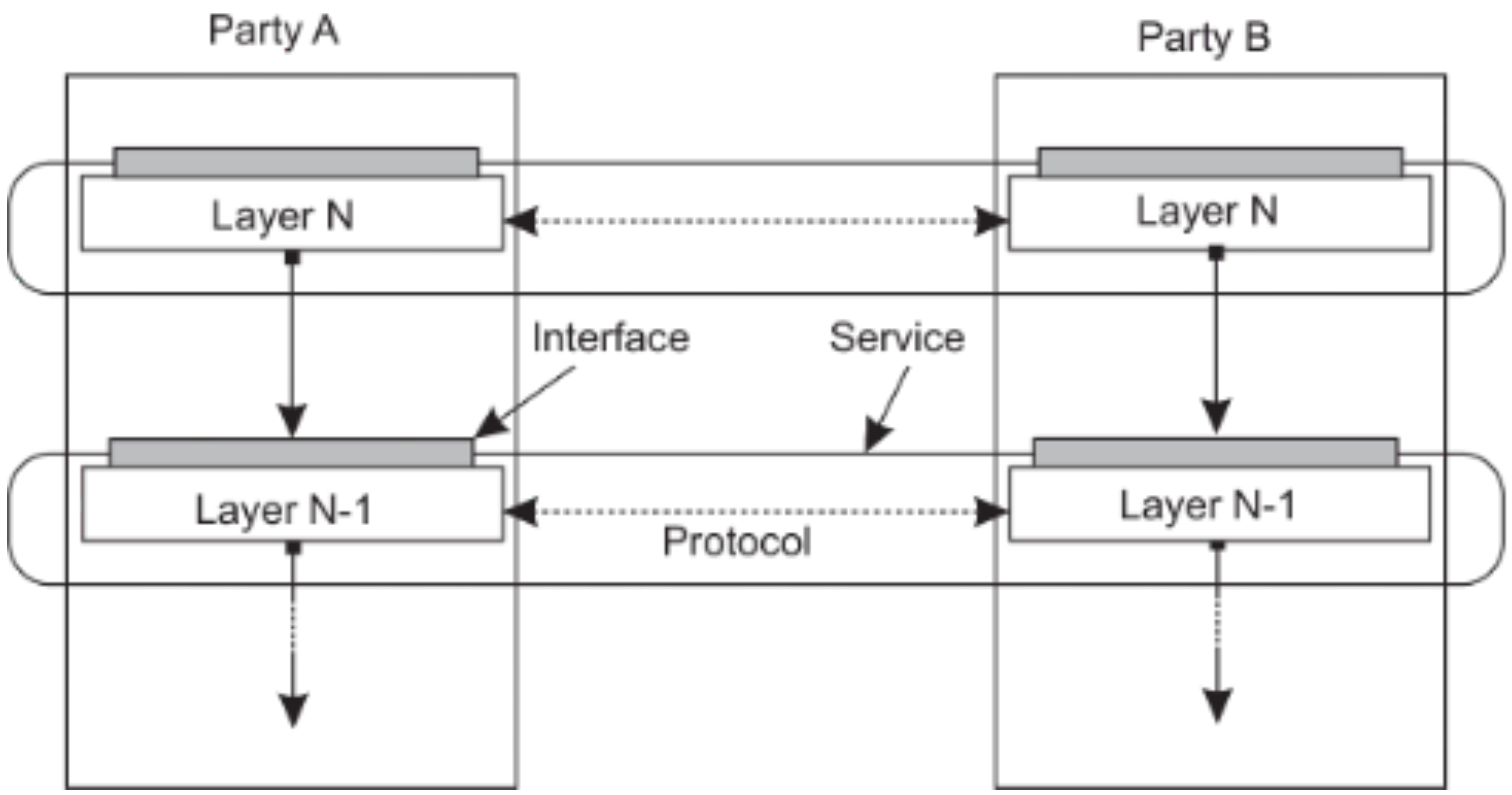
- A style is formulated in terms of
  - (replaceable) components with well-defined interfaces
  - the way that components are connected to each other
  - the data exchanged between components
  - how these components and connectors are jointly configured into a system.

- Connector
  - A mechanism that mediates communication, coordination, or cooperation among components. Example: facilities for (remote) procedure call, messaging, or streaming.

Layered Architecture



Example: Communication Protocols



Client Server Architecture

```
from socket import *
s = socket(AF_INET, SOCK_STREAM)
(conn, addr) = s.accept()
while True:
        data = conn.recv(1024)
        if not data: break
        msg = data.decode()+"*"
        conn.send(msg.encode())
conn.close()
```

```
from socket import *
s = socket(AF_INET, SOCK_STREAM)
s.connect((HOST, PORT))
msg = "Hello World"
s.send(msg.encode())
data = s.recv(1024)
print(data.decode())
s.close()
```

Explanation

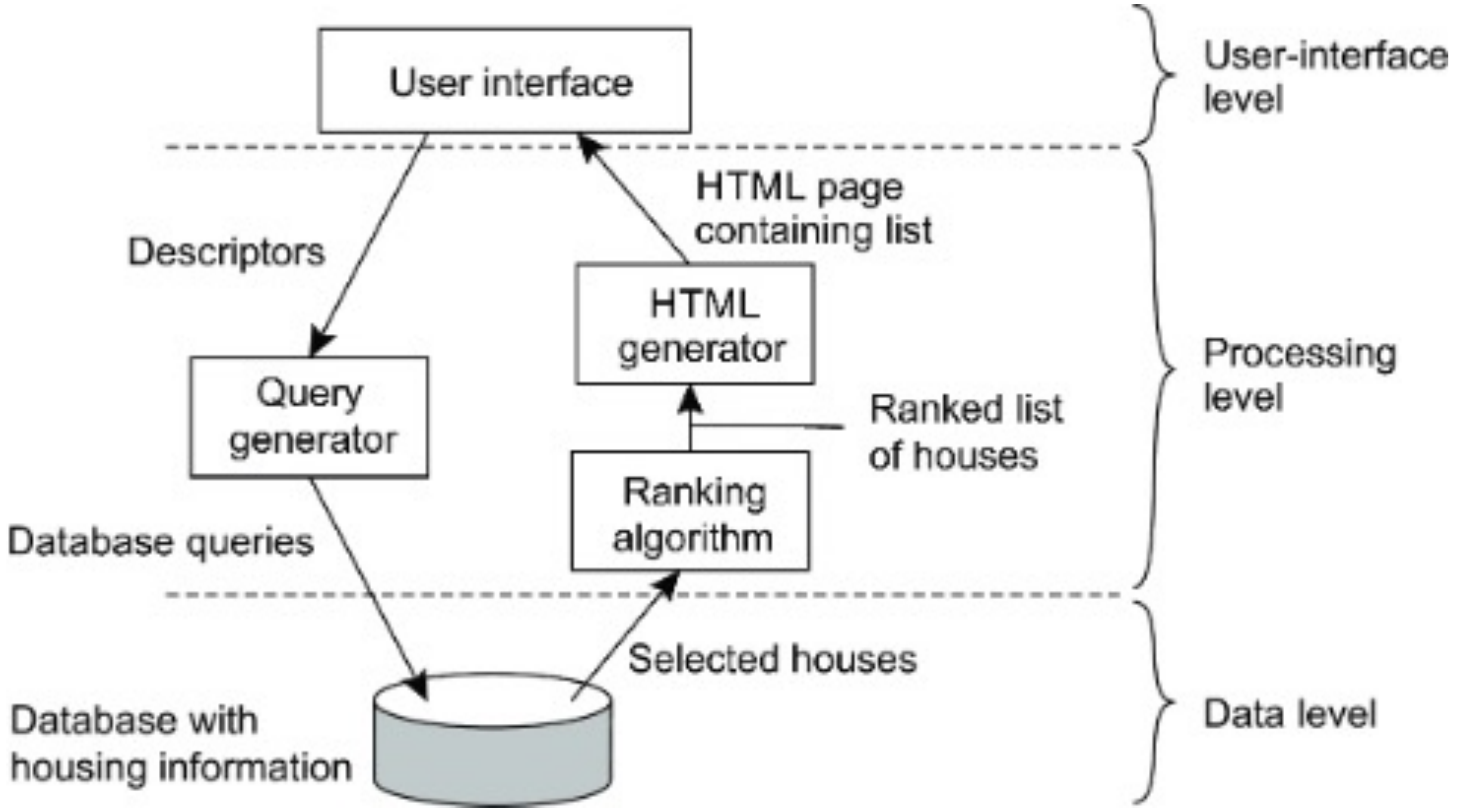
- There is a connection-oriented **service**
    - offered by the socket library

Application Layering

- Traditional three-layered view
    - Application-interface layer contains units for interfacing to users or external applications
    - Processing layer contains the functions of an application, i.e., without specific data
    - Data layer contains the data that a client wants to manipulate through the application components

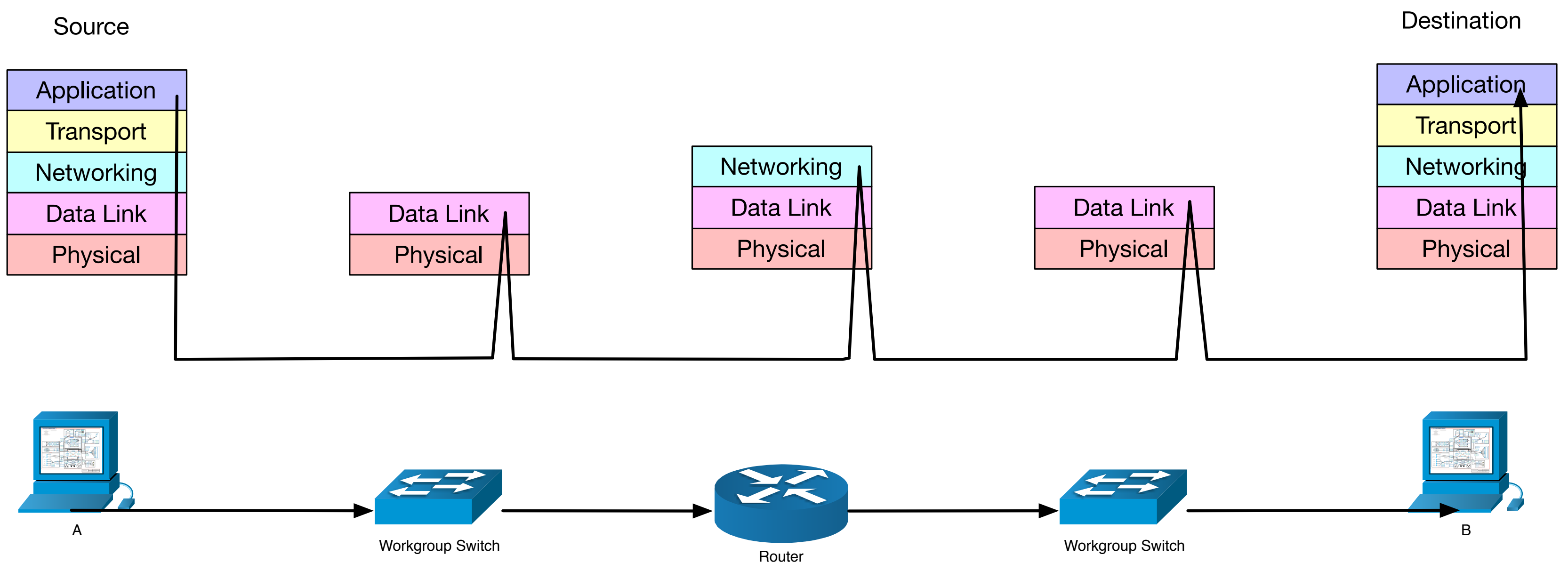
- Observation
    - This layering is found in many distributed information systems, using traditional database technology and accompanying applications.

Application Layering



Application Layering

- Layering works well for the communications stack
- TCP/IP stack
    - Application layer
    - Transport layer
    - Network layer
    - Physical layer



Application Layering

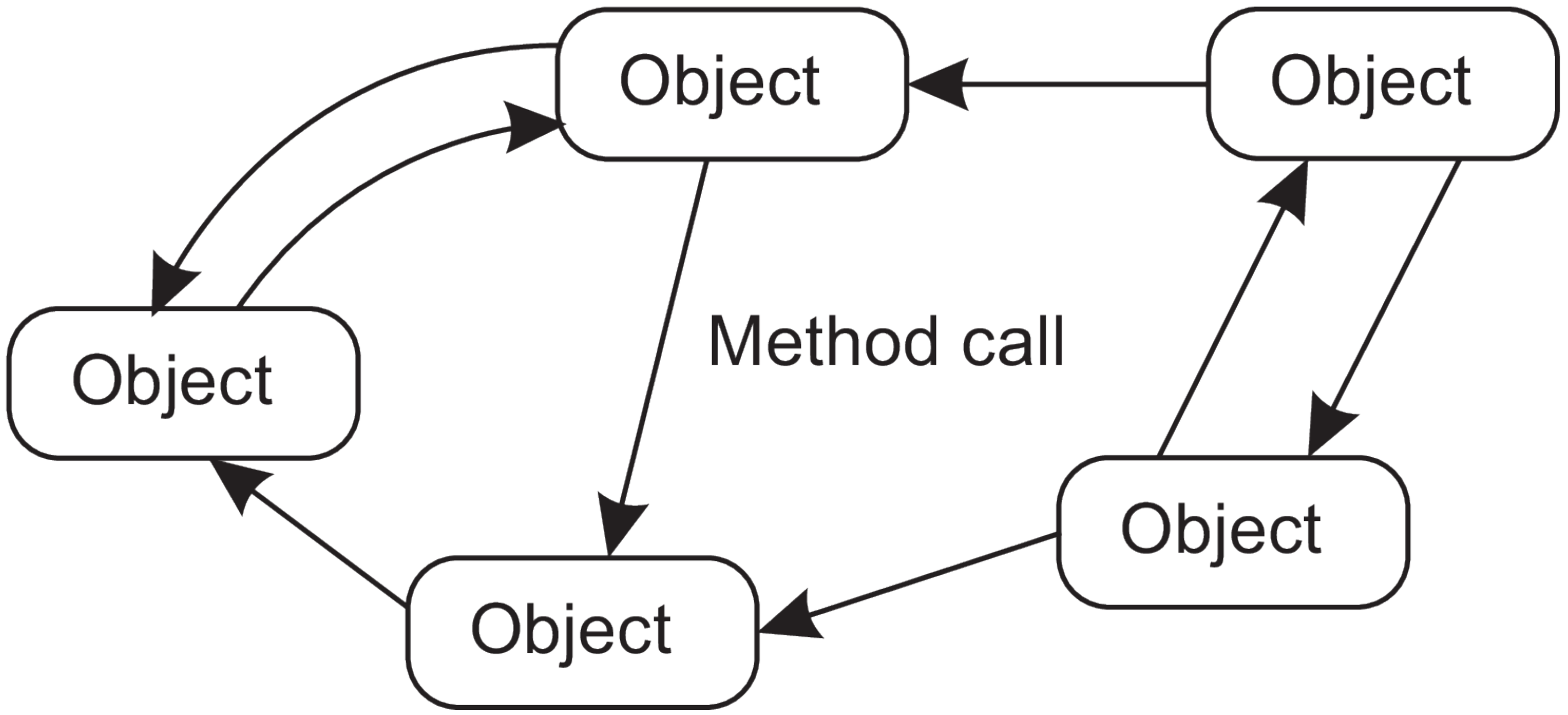
- Bad examples for application layering
  - Layers that are not scalable
  - Excessive copying (e.g. logins are excruciatingly slow)
  - Layers that introduce dependencies to components under outside control
    - Resulting in legal issues
    - Chaos if a developer withdraws the component
- Alternative: Looser coupling: Service oriented architectures

Service Oriented Architectures

- Based on
    - Objects
    - Services
    - Micro-services
- Each service is executed as its own thread

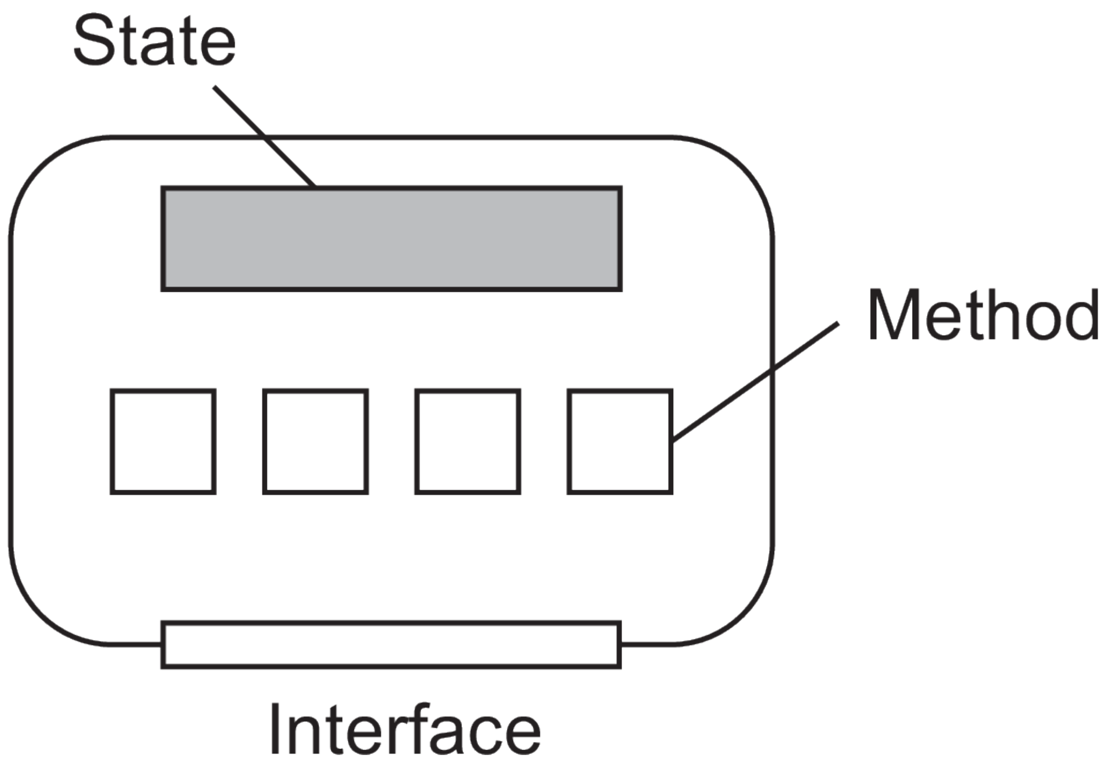
Service Oriented Architectures: Object-based architectures

- Objects interact with each other through method calls



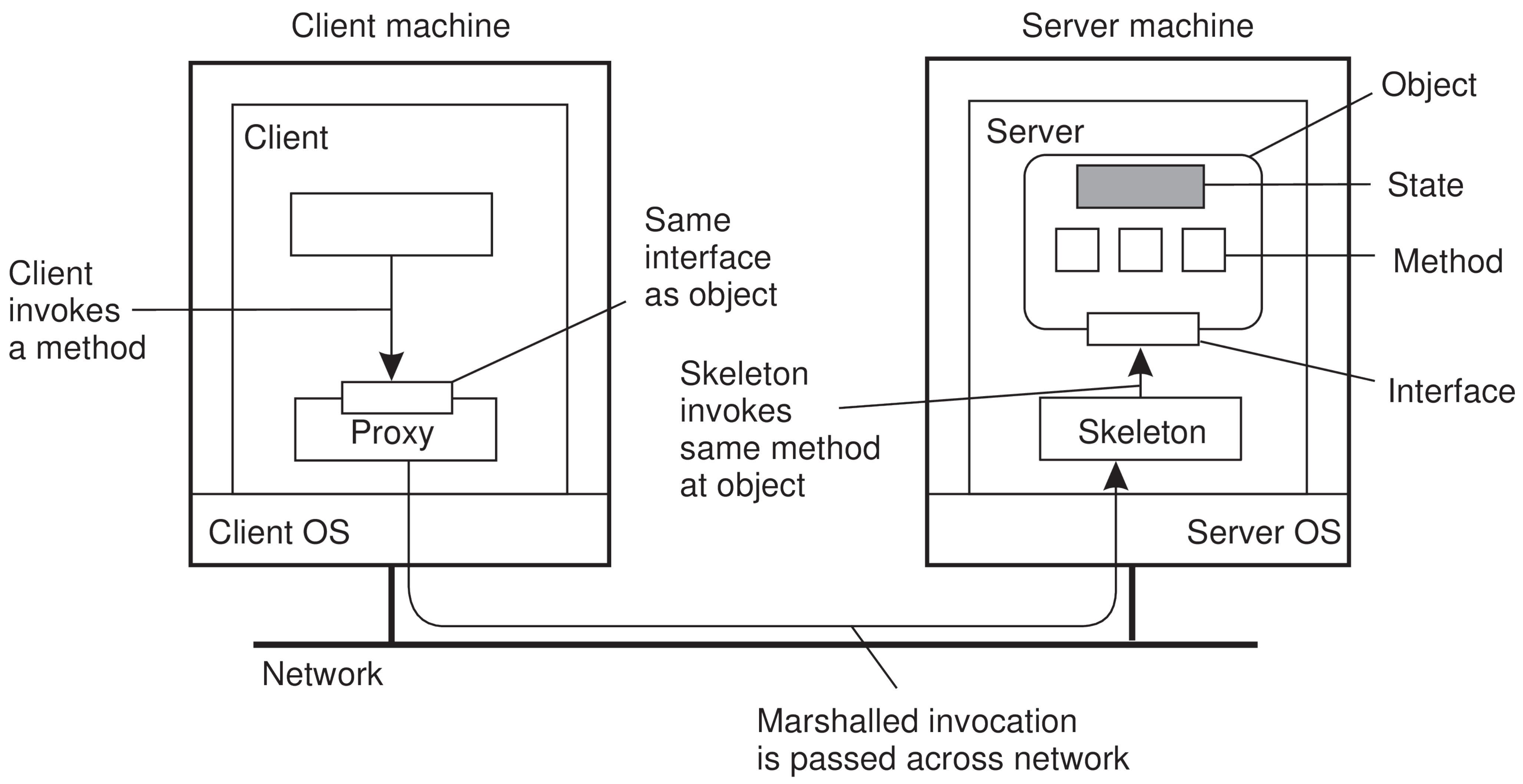
Service Oriented Architectures: Object-based architectures

- The *state* of an object encapsulates data
- *Methods* encapsulate the transformation of the data
- Interface conceals implementation details



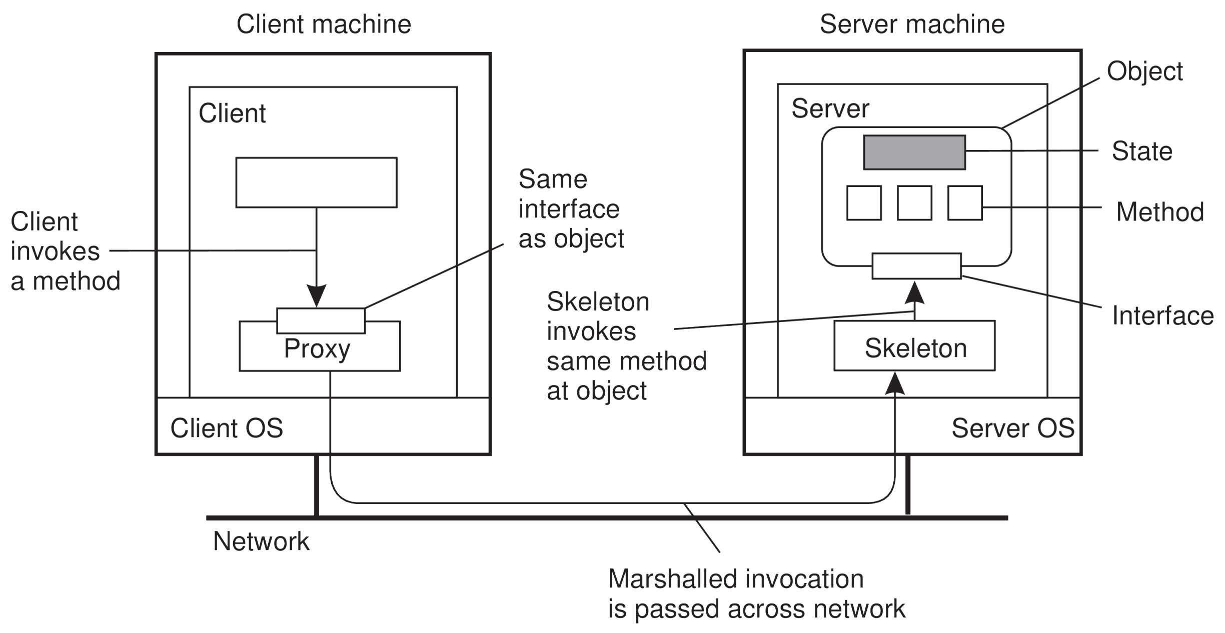
Service Oriented Architectures: Object-based architectures

- Implementing *remote objects*



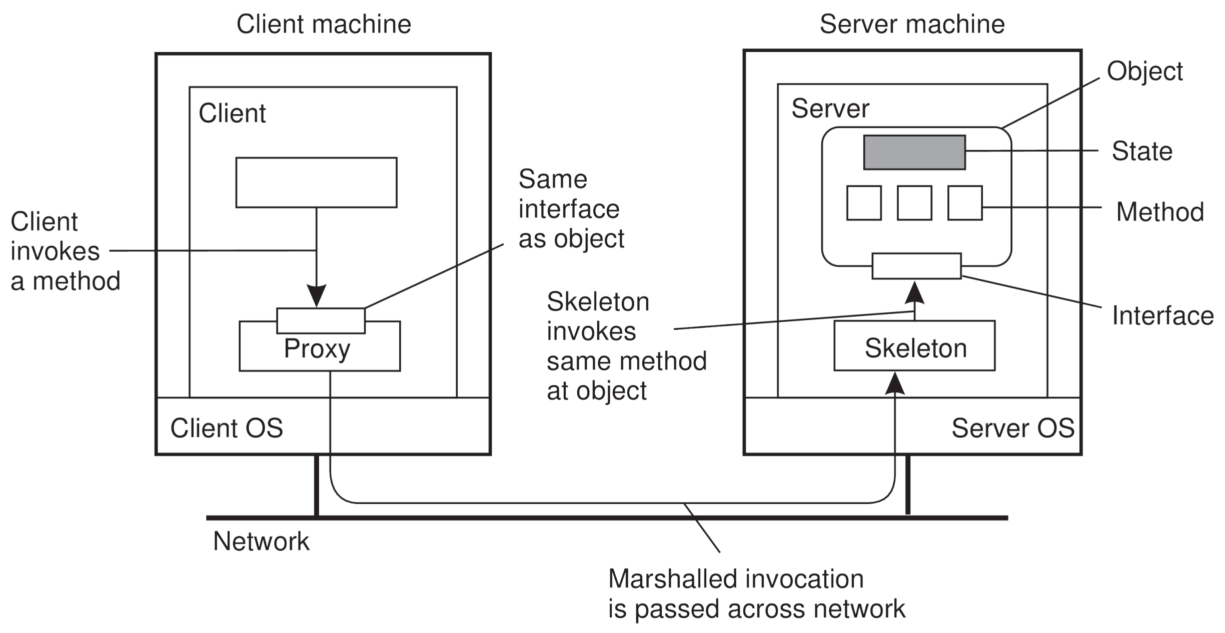
Service Oriented Architectures: Object-based architectures

- Client binds to a remote object by loading a *proxy* into its address space
  - (corresponding to a *stub* for **R**emote **P**rocedure **C**alls)
- The proxy implements the remote object's interface



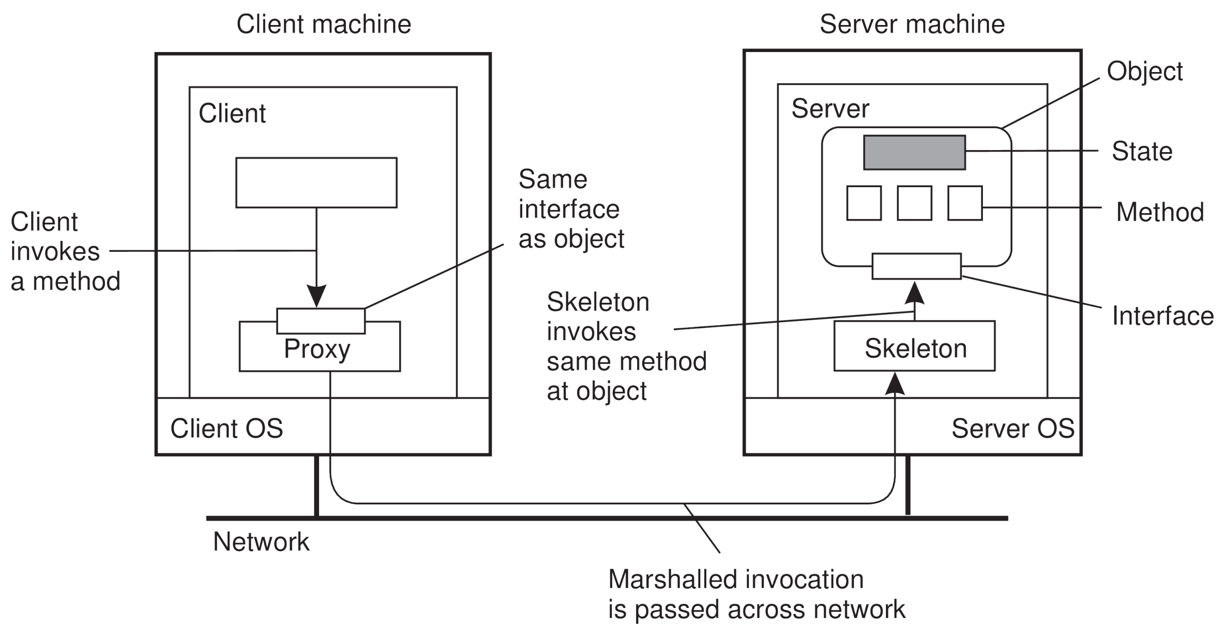
Service Oriented Architectures: Object-based architectures

- When a method of the remote object is invoked:
  - Parameters are passed to the Proxy
  - Proxy *marshals* parameters into a common format
  - Proxy sends the marshalled invocation to the server



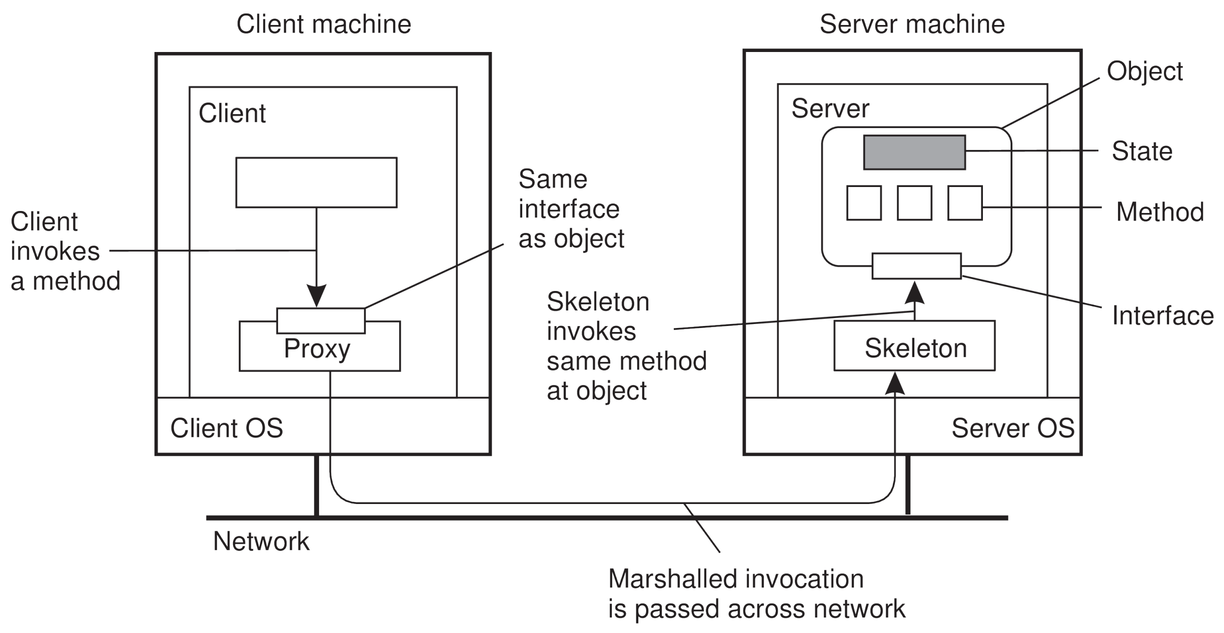
Service Oriented Architectures: Object-based architectures

- When a method of the remote object is invoked:
  - Message is passed on to the *Skeleton,* which unmarshalls the parameters
  - Skeleton then makes a method call to the remote object
  - Skeleton receives result



Service Oriented Architectures: Object-based architectures

- When a method of the remote object is invoked:
    - Skeleton marshals results into a common format
    - Marshalled results are sent to the Proxy
    - Proxy unmarshalls the results and passes them on to the client

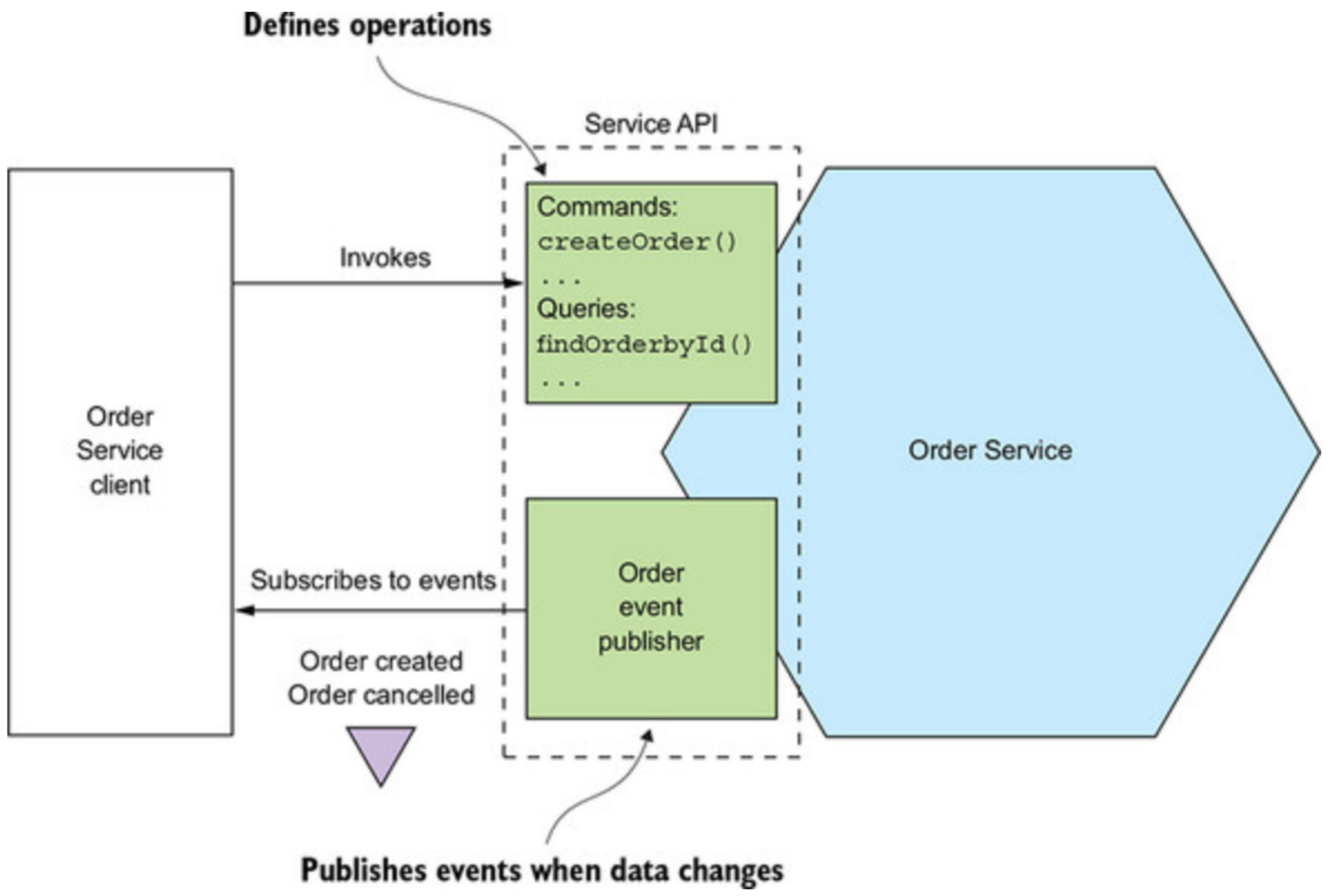


Service Oriented Architectures: Object-based architectures

- In this architecture:
    - The state of the distributed system is not distributed
    - It is made up of the states of the objects
    - But usually only one object matters, the one accessed by the user

Service Oriented Architectures: Micro-service architectures

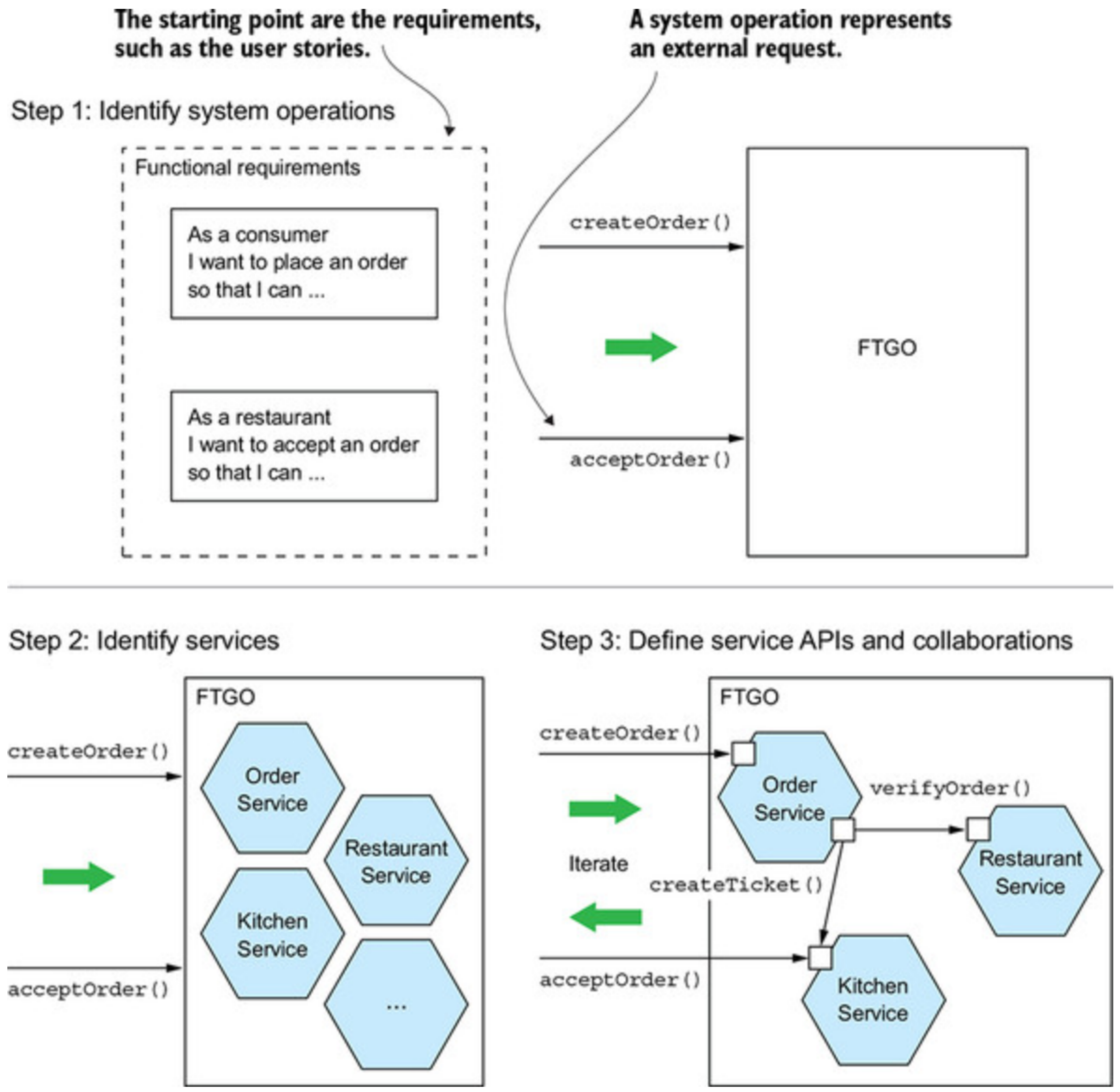
- A *service* is a standalone, independently deployable software component that implements some useful functionality.
  - Example:
  -



Service Oriented Architectures: Micro-service architectures

- Services are loosely coupled:
  - All interactions happen via an API
  - There is no common database that is accessed
    - Complicates data consistency and queries over several services
    - Eases contention for data access

Service Oriented Architectures: Micro-service architectures


The starting point are the requirements, such as the user stories.
A system operation represents an external request.
Step 1: Identify system operations
Functional requirements
As a consumer I want to place an order so that I can ...
As a restaurant I want to accept an order so that I can ...
createOrder()
acceptOrder()
FTGO
Step 2: Identify services
FTGO
createOrder()
acceptOrder()
Order Service
Restaurant Service
Kitchen Service
...
Step 3: Define service APIs and collaborations
FTGO
createOrder()
Order Service
verifyOrder()
Restaurant Service
Iterate
createTicket()
acceptOrder()
Kitchen Service

Resource based architectures

- Resource-based architecture
  - Views a distributed system as a huge collection of resources that are individually managed by component
  - Resources can be added and removed by remote applications
  - Resources can be modified and retrieved by remote applications

Resource based architectures

- **Re**presentational **S**tate **T**ransfer (REST) [Fielding 2000]
  - Resources are identified through a single naming scheme
  - All services offer the same interface, consisting of at most four operations: Put, Post, Get, Delete
  - Messages sent to or from a service are fully self-described
  - After executing an operation at a service, that component forgets everything about the caller

Resource based architectures

- REST operations

Operation	Description
`PUT`	Modify a resource by transferring a new state
`POST`	Create a new resource
`GET`	Retrieve the state of a resource in some representation
`DELETE`	Delete a resource

Resource based architectures

- HTTP actions
  - GET: Get a representation of this resource.
  - DELETE: Destroy this resource.
  - POST: Create a new resource underneath this one, based on the given representation.
  - PUT / PATCH: Replace this state of this resource with the one described in the given representation.
  - HEAD, OPTIONS, CONNECT, TRACE
- But REST $\neq$ HTTP, but HTTP is the application-level interface

Resource based architectures

- REST Constraints
- Client/Server - Client are separated from servers by a well-defined interface
- Stateless - A specific client does not consume server storage when it is "at rest"
- Cache - Responses indicate their own cacheability
- Uniform interface
- Layered system - A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way
- Code on demand (optional) - Servers are able to temporarily extend or customize the functionality of a client by transferring logic to the client that can be executed within a standard virtual machine

Resource based architectures

- REST Example: Amazon Simple Storage Service — S3
  - Two resources: objects (files) and buckets (directories)
    - Create an object / bucket:
      - Send HTTP PUT request with URI of object/bucket
    - To find all objects in a bucket:
      - Send HTTP GET request to bucket URI

Publish-Subscribe Architectures

- Coupling as loose as possible
- Separate processing and coordination

- System as a collection of autonomously operating processes

Publish-Subscribe Architectures

- Temporal coupling: Up at the same time
- Referential coupling: Knowing or not knowing each other
  - Examples of coupling:

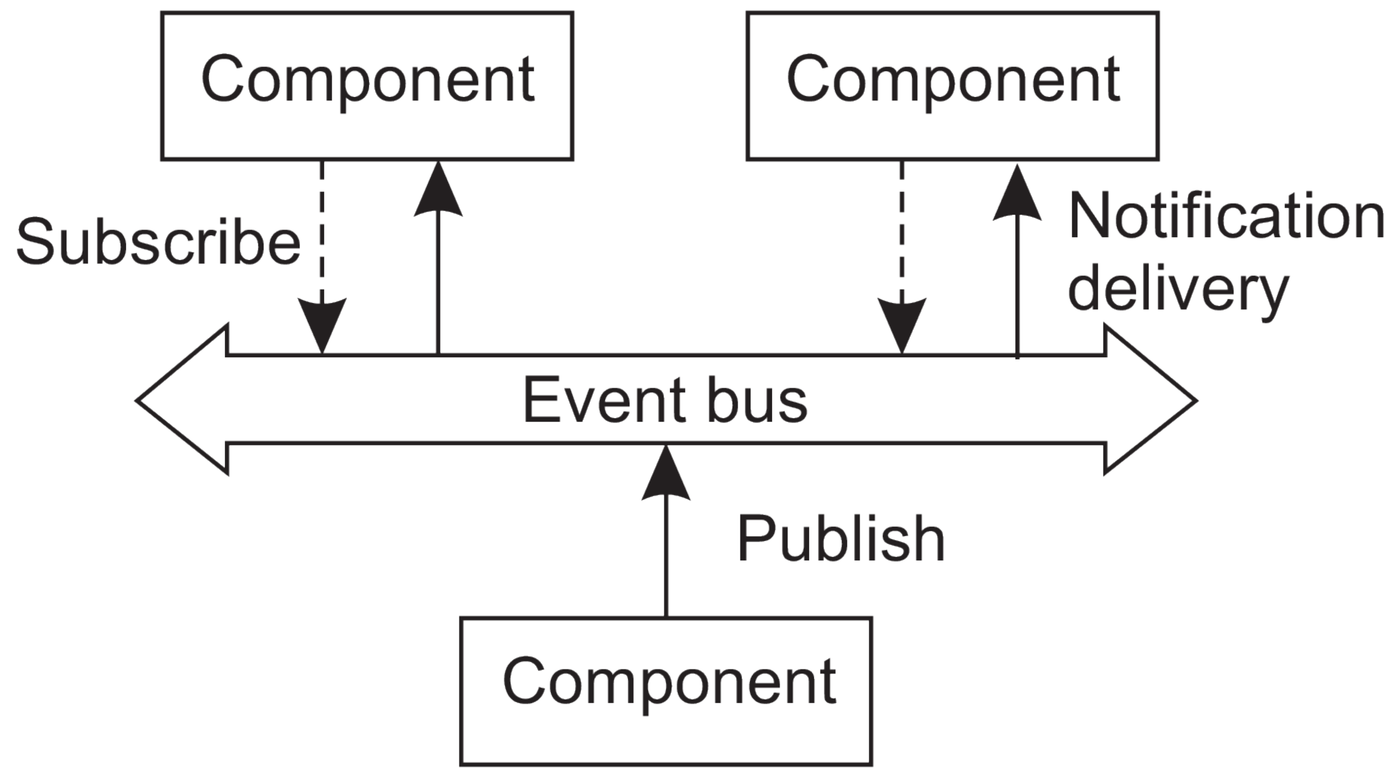
	Temporally coupled	**Temporally decoupled**
Referentially coupled	Direct	Mailbox
Referentially decoupled	Event-based	Shared data space

Publish-Subscribe Architectures

- Mailbox:
    - Processes can send data to a mail-box type queue

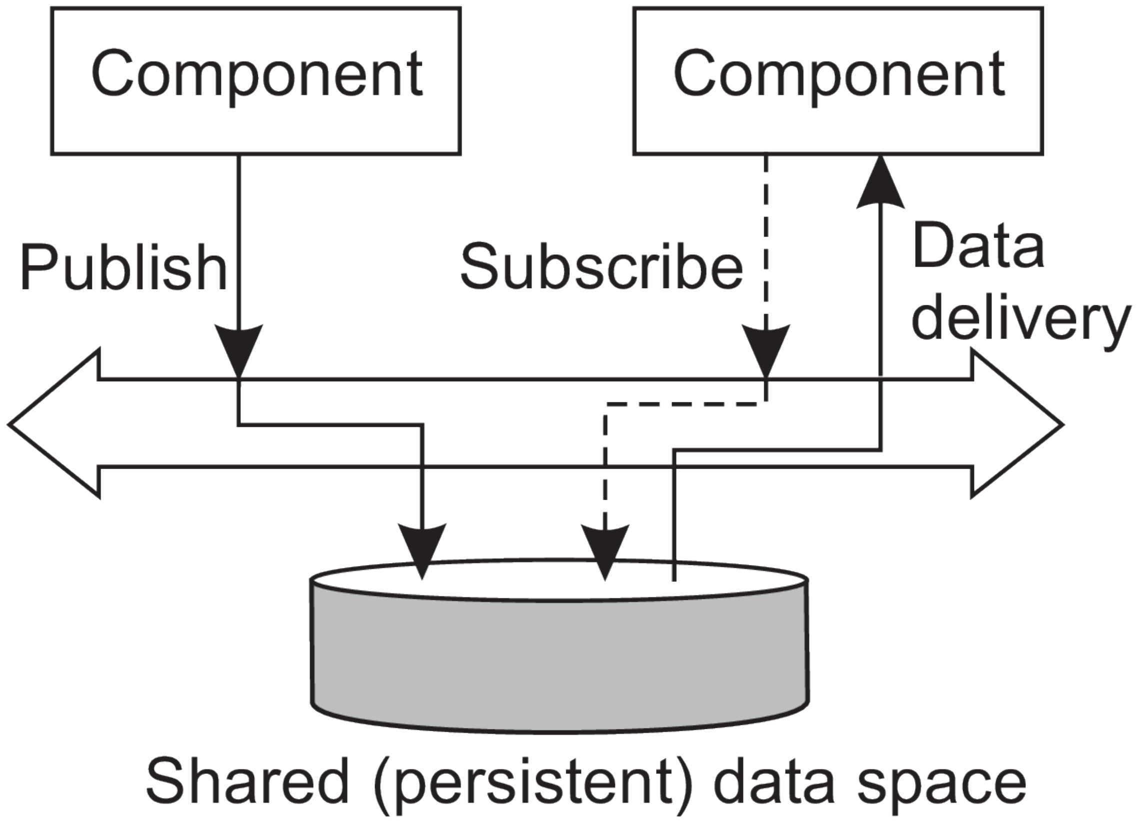
Publish-Subscribe Architectures

- Event-based coordination
  - Process can publish notifications
  - Processes can subscribe to certain notifications



Publish-Subscribe Architectures

- Shared Data Space:
  - Processes communicate via tuples
    - Like the rows of a relational database table



Publish-Subscribe Architectures

- Example: Linda Tuple Space (1989)
  - **OUT** primitive for storing tuples
  - **RD** primitive for reading stored tuple
  - **IN** primitive for reading and deleting stored tuple
- in and rd are **blocking**. If there are no matching tuple, the requesting process blocks

Publish-Subscribe Architectures

- Linda example continued:
  - Dining philosopher problems:
    - A bunch of philosophers sit around a table and think
    - Sometimes, they want to eat, for which they need two shared forks

Publish-Subscribe Architectures

```
phil(i)
   int i;
{
   while(1) {
    think();
    in("room ticket");
    in("chopstick", i);
    in("chopstick", (i+1)%Num);
    eat();
    out("chopstick", i);
    out("chopstick", (i+1)%Num);
    out("room ticket");
   }
}

initialize()
{
   int i;
   for (i = 0; i < Num; i++) {
    out("chopstick", i);
    eval(phil(i));
    if (i < (Num-1)) out("room ticket");
   }
}
```

Publish-Subscribe Architectures

```
blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]

blog._out(("bob","distsys","I am studying chap 2"))
blog._out(("bob","distsys","The linda example's pretty simple"))
blog._out(("bob","gtcn","Cool book!"))
```

(a) Bob's code for creating a microblog and posting three messages.

```
blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]

blog._out(("alice","gtcn","This graph theory stuff is not easy"))
blog._out(("alice","distsys","I like systems more than graphs"))
```

(b) Alice's code for creating a microblog and posting two messages.

```
blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]

t1 = blog._rd(("bob","distsys",str))
t2 = blog._rd(("alice","gtcn",str))
t3 = blog._rd(("bob","gtcn",str))
```

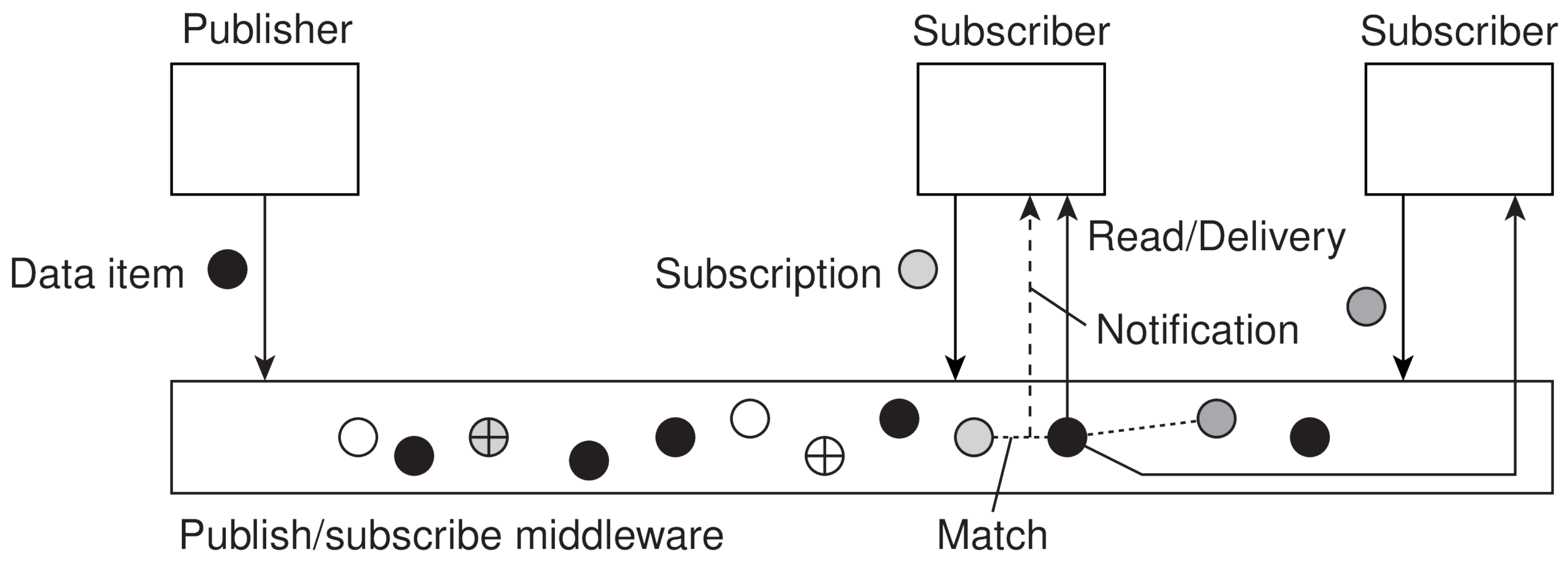
(c) Chuck reading a message from Bob's and Alice's microblog.

Publish-Subscribe Architectures

- Subscribe needs to describe the items of interest
    - This means *naming*
- Topic-based Publish-Subscribe Architecture:
    - Items are described
        - In the form of key-value pairs
- Content-based Publish-Subscribe Architecture
    - Items are directly compared to the search criteria

Publish-Subscribe Architectures

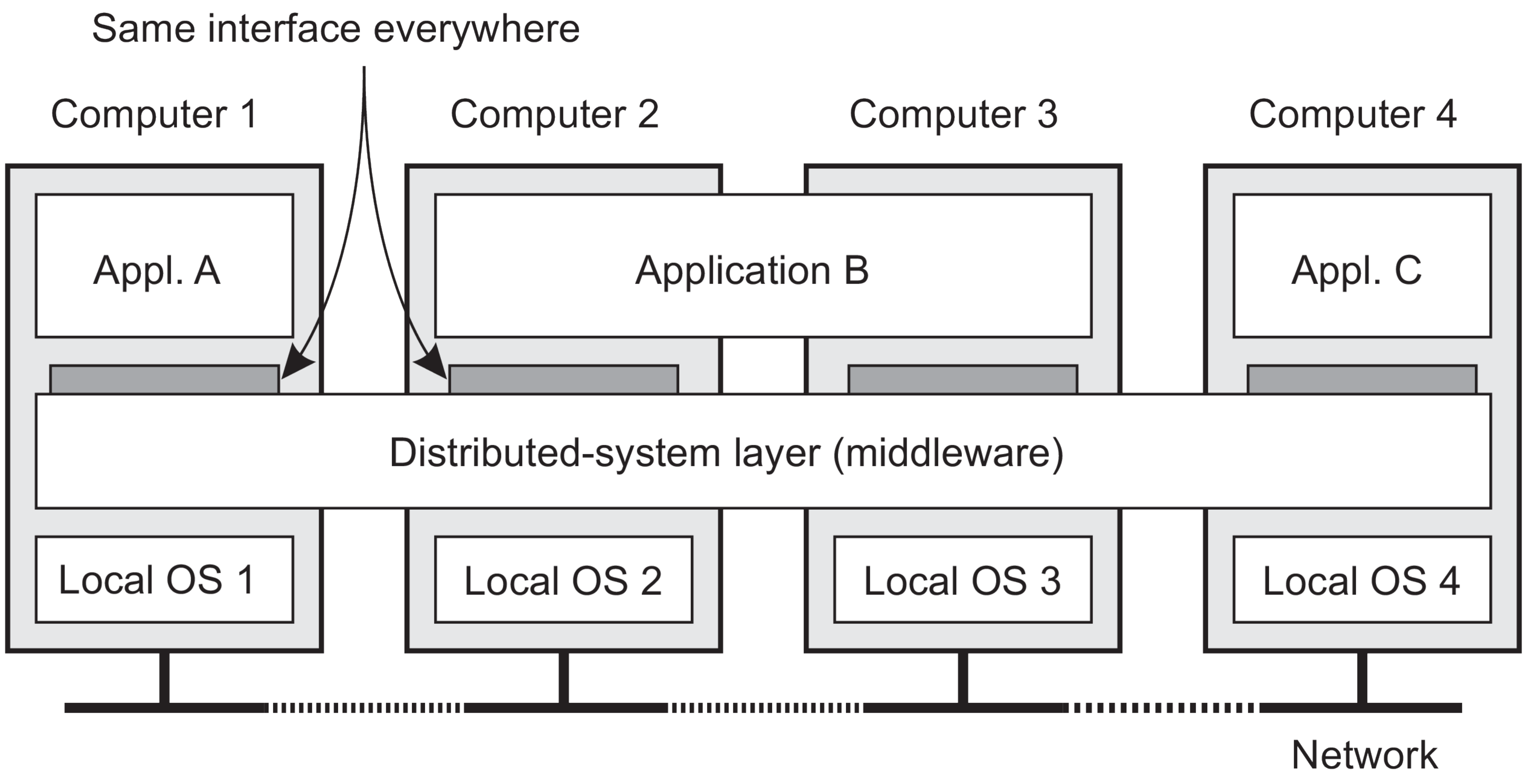
- Need a middleware layer that allows the matching and forwarding
    - Middleware can notify or can forward



Middleware and Distributed Systems

Middleware

- Classical Middleware



Middleware

- Middleware: manager of resources to share and deploy resources across a network
- Typical middleware services
    - Facilities for inter-application communication.
    - Security services.
    - Accounting services.
    - Masking of and recovery from failures.

Middleware

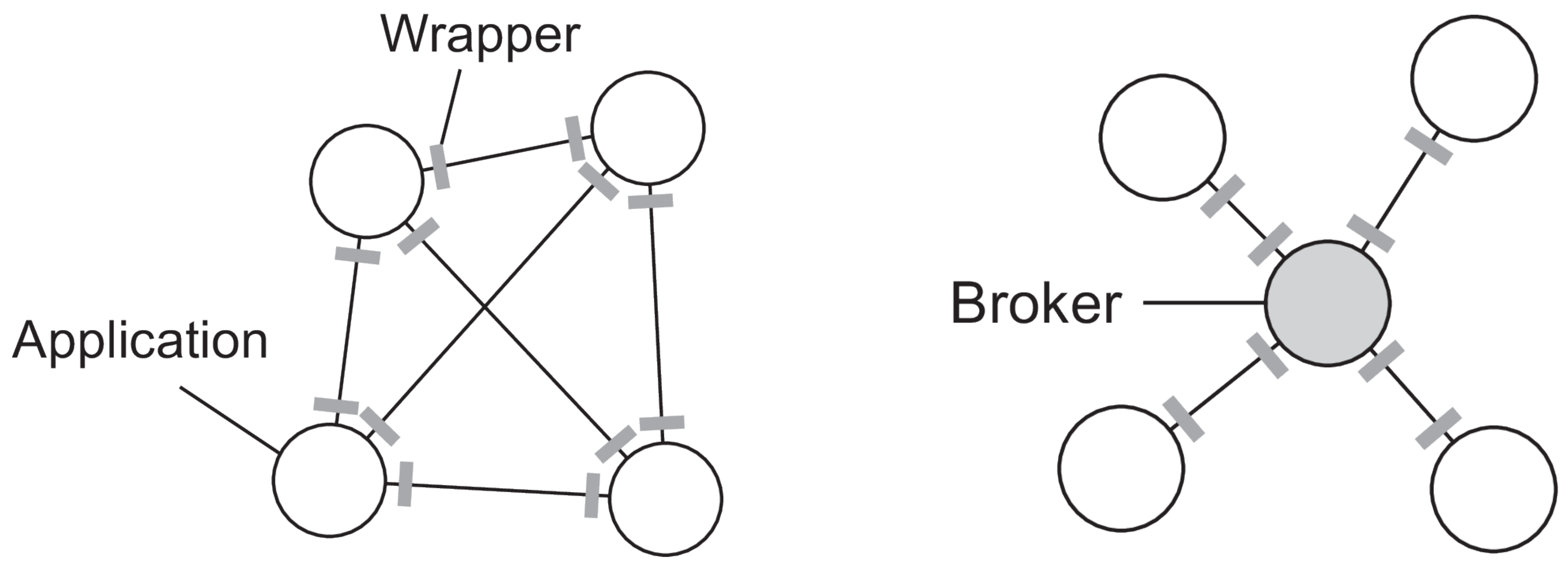
- Problem
  - The interfaces offered by a legacy component are most likely not suitable for all applications.
- Solution
  - A wrapper or adapter offers an interface acceptable to a client application. Its functions are transformed into those available at the component.

Middleware Design Pattern Wrappers

- A wrapper or adapter
  - Special component that offers an interface acceptable to a client application, of which the functions are transformed into those available at the component.

Middleware Design Pattern Wrappers

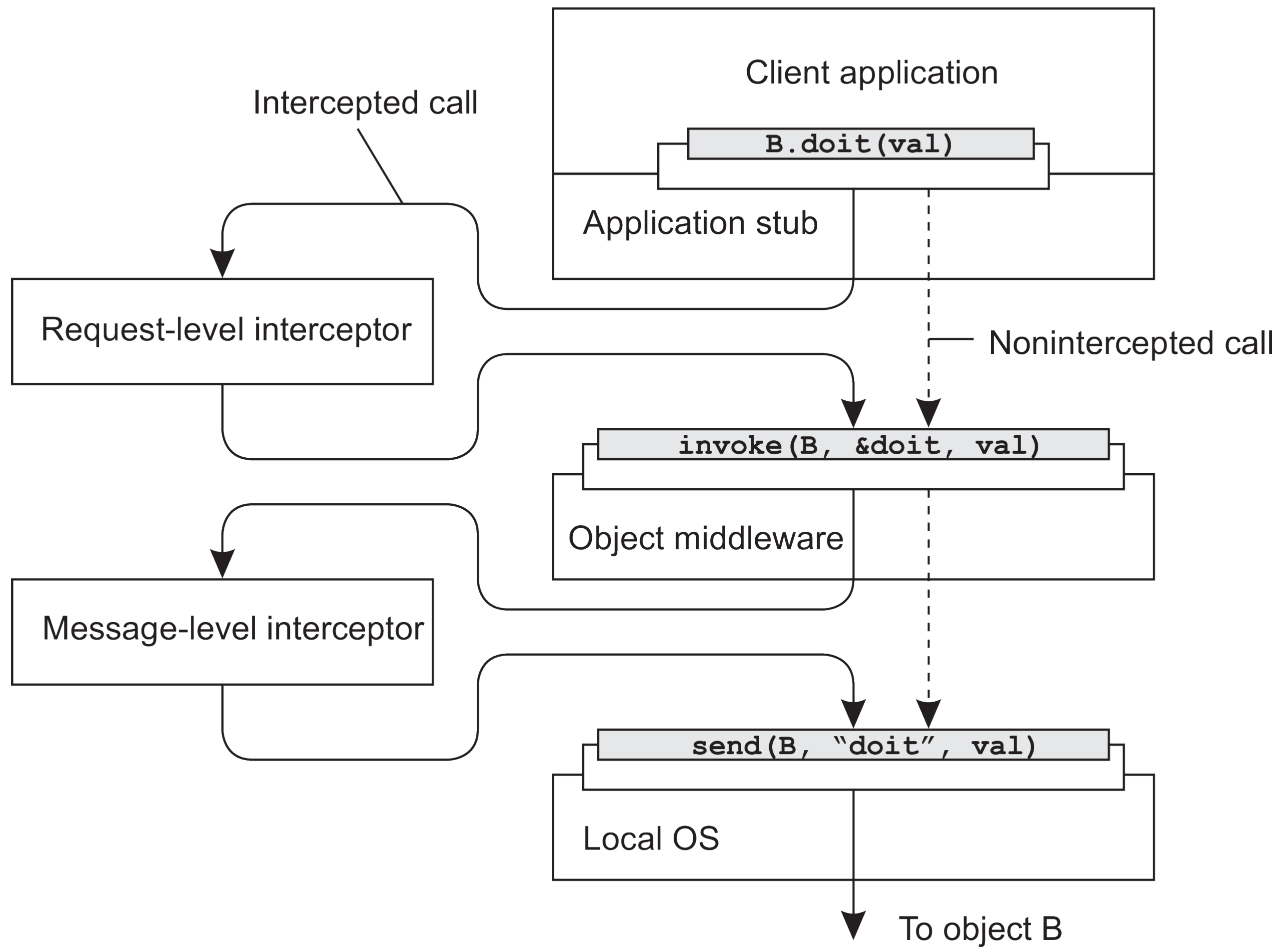
- Number of wrappers can become huge
  - N systems could need $N(N-1)$ wrappers
  - Can use a broker:



Middleware Design Pattern Interceptors

- Interceptor — Software construct that breaks the normal flow of control and allows other code to be executed
- Object A calls a method of object B
  - Object A is offered a local interface that is the same as the interface offered by object B. A calls the method available in that interface.
  - The call by A is transformed into a generic object invocation, made possible through a general object-invocation interface offered by the middleware at the machine where A resides.
  - Finally, the generic object invocation is transformed into a message that is sent through the transport-level network interface as offered by A’s local operating system.
  -

Middleware Design Pattern Interceptors



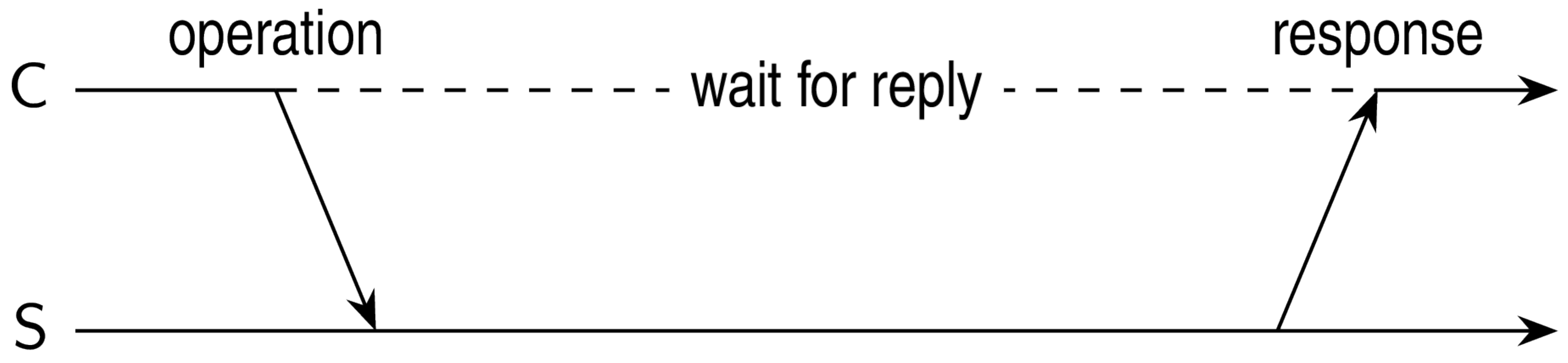
Modifiable Middleware

- Modifiable Middleware — adaptive software
- Problem
  - Middleware contains solutions that are good for most applications
    - ⇒ you may want to adapt its behavior for specific applications.
- Interceptors are the basic solution

Centralized System Architectures

Centralized System Architectures

- Basic Client–Server Model
  - Characteristics:
    - There are processes offering services (servers)
    - There are processes that use services (clients)
    - Clients and servers can be on different machines
    - Clients follow request/reply model regarding using services

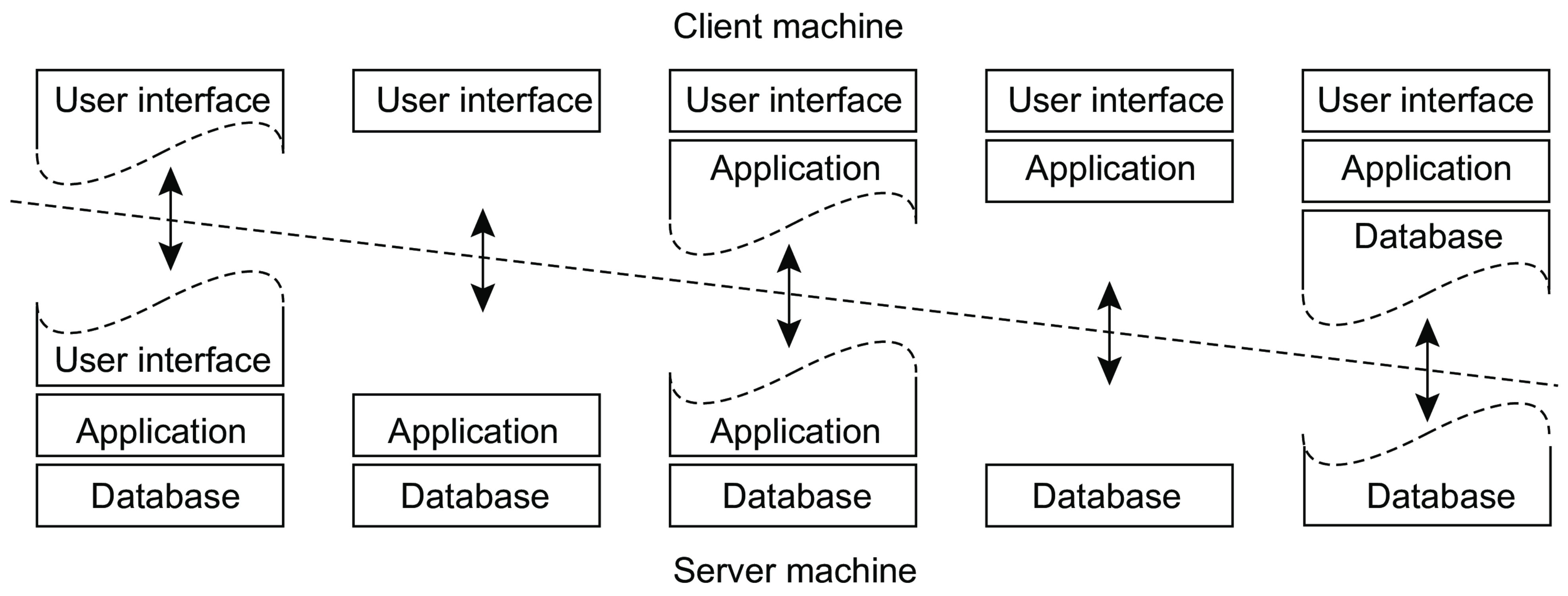


Centralized System Architectures

- Multi-tiered centralized system architectures
  - Some traditional organizations
    - Single-tiered: dumb terminal/mainframe configuration
    - Two-tiered: client/single server configuration
    - Three-tiered: each layer on separate machine

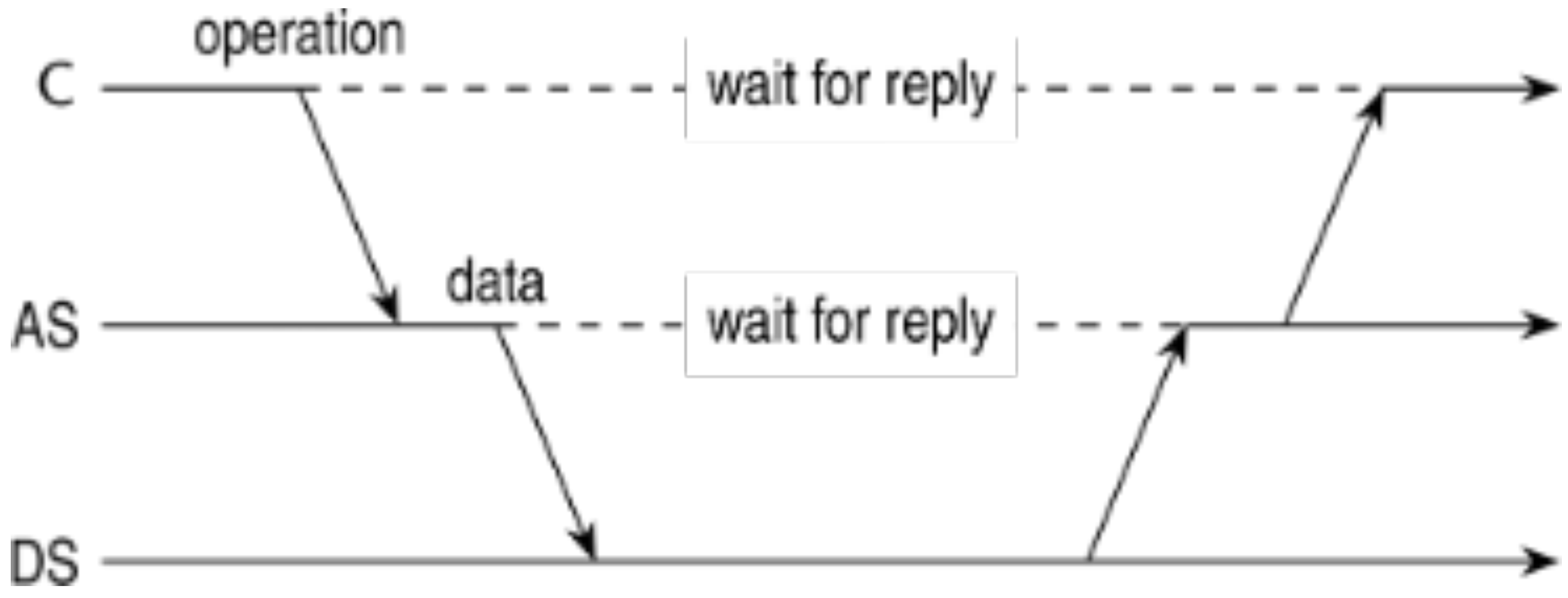
Centralized System Architectures

- Two-Tiered Systems



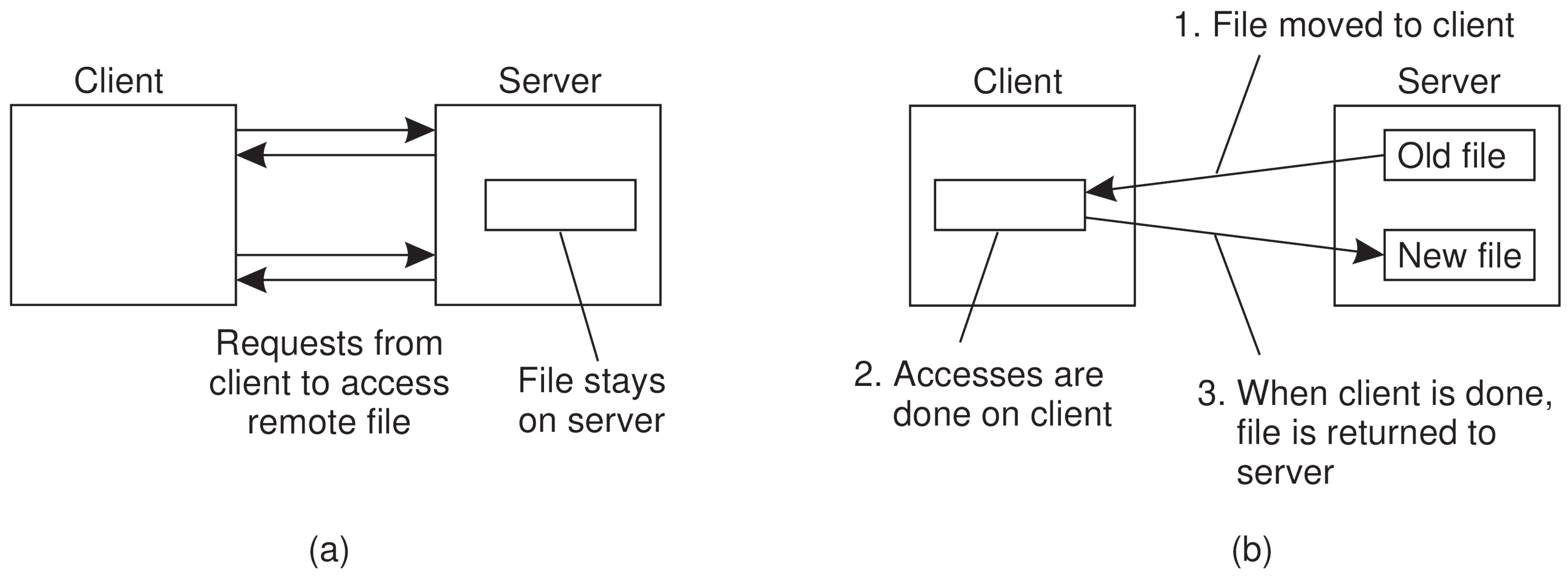
Centralized System Architectures

- Component can be client and server at the same time

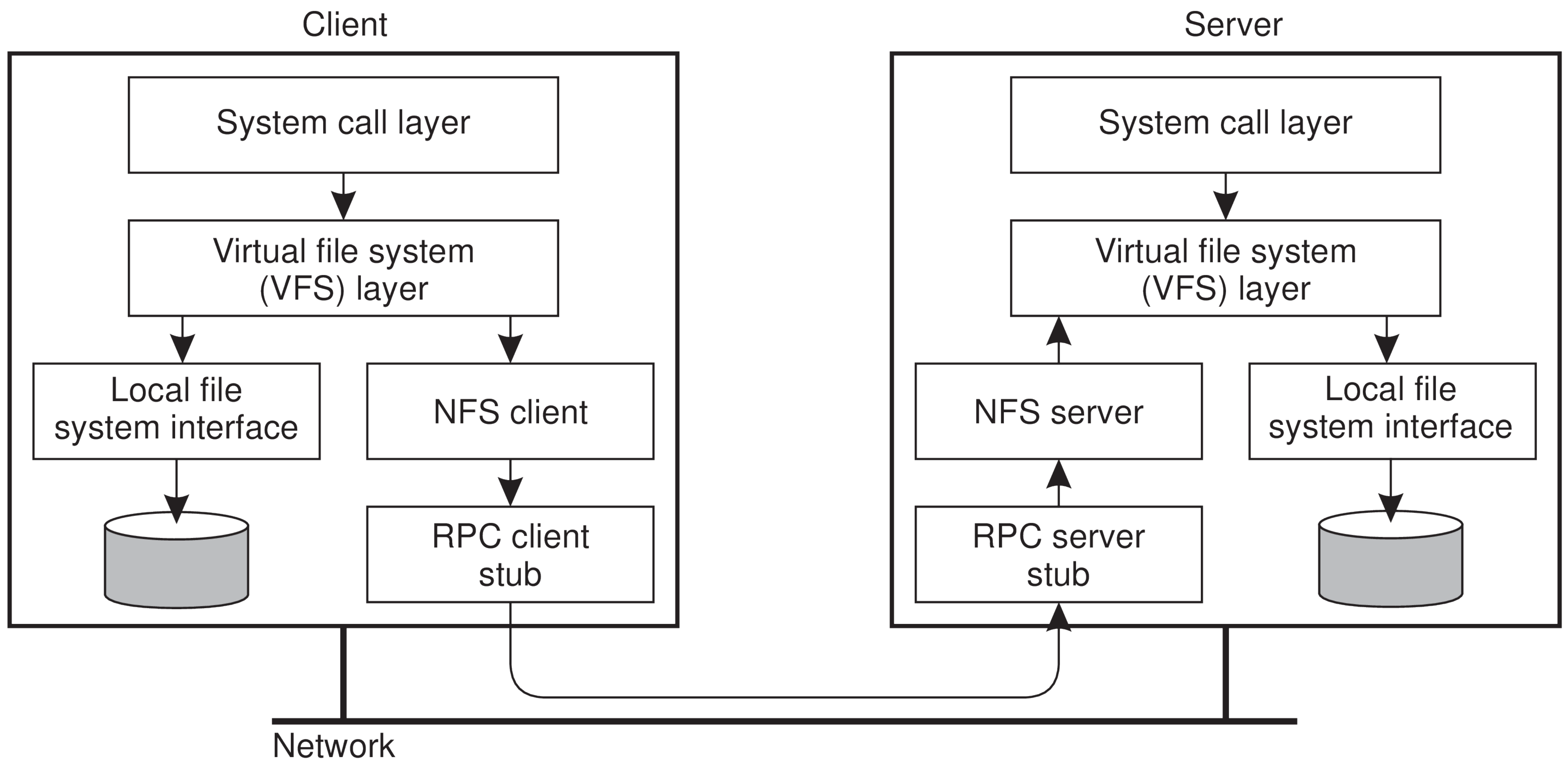


Example: Network File System

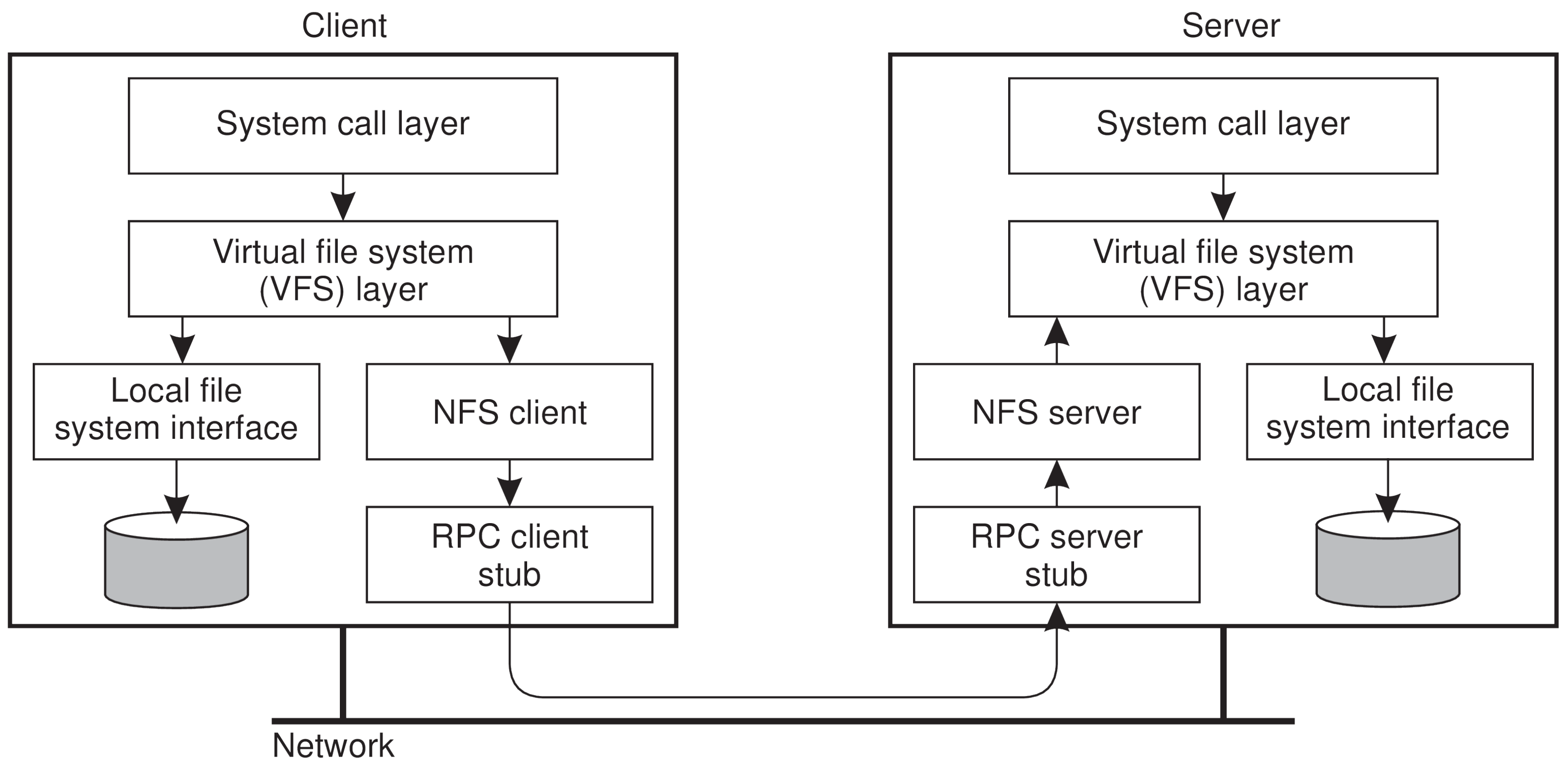
- Foundations
  - Each NFS server provides a standardized view of its local file system: each server supports the same model, regardless the implementation of the file system.
- The NFS remote access model (left) and the upload download alternative (right


Client
Server
Requests from client to access remote file
File stays on server
(a)
1. File moved to client
Client
Server
Old file
New file
2. Accesses are done on client
3. When client is done, file is returned to server
(b)

Example: Network File System


Client
System call layer
Virtual file system (VFS) layer
Local file system interface
NFS client
RPC client stub
Server
System call layer
Virtual file system (VFS) layer
NFS server
Local file system interface
RPC server stub
Network

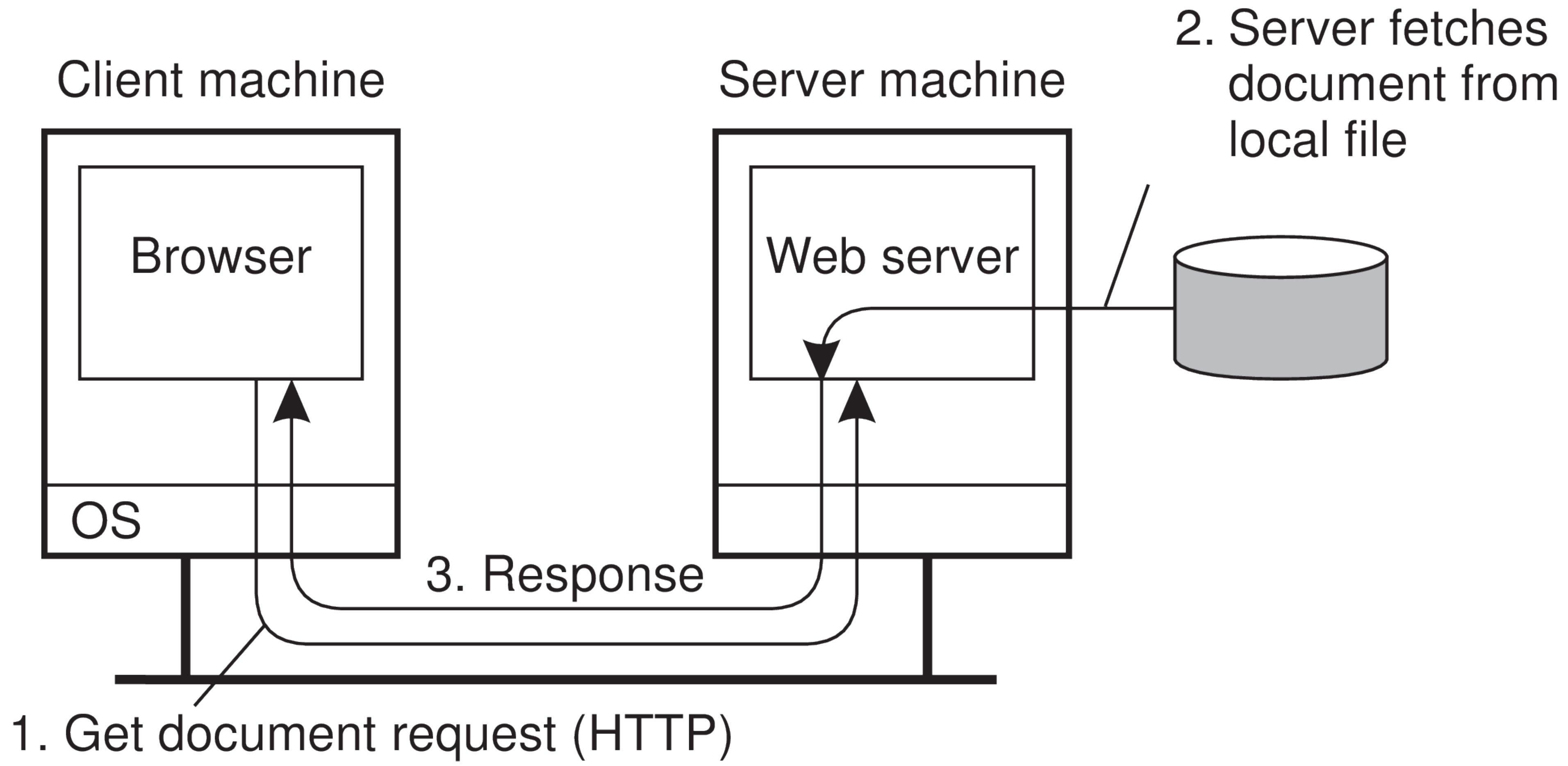
Example: Network File System


Client
System call layer
Virtual file system (VFS) layer
Local file system interface
NFS client
RPC client stub
Server
System call layer
Virtual file system (VFS) layer
NFS server
Local file system interface
RPC server stub
Network

- VFS replaces the interface of the Client’s file system
- Client-server communication uses Remote Procedure Calls

Example: The Web

- Simple Web Servers



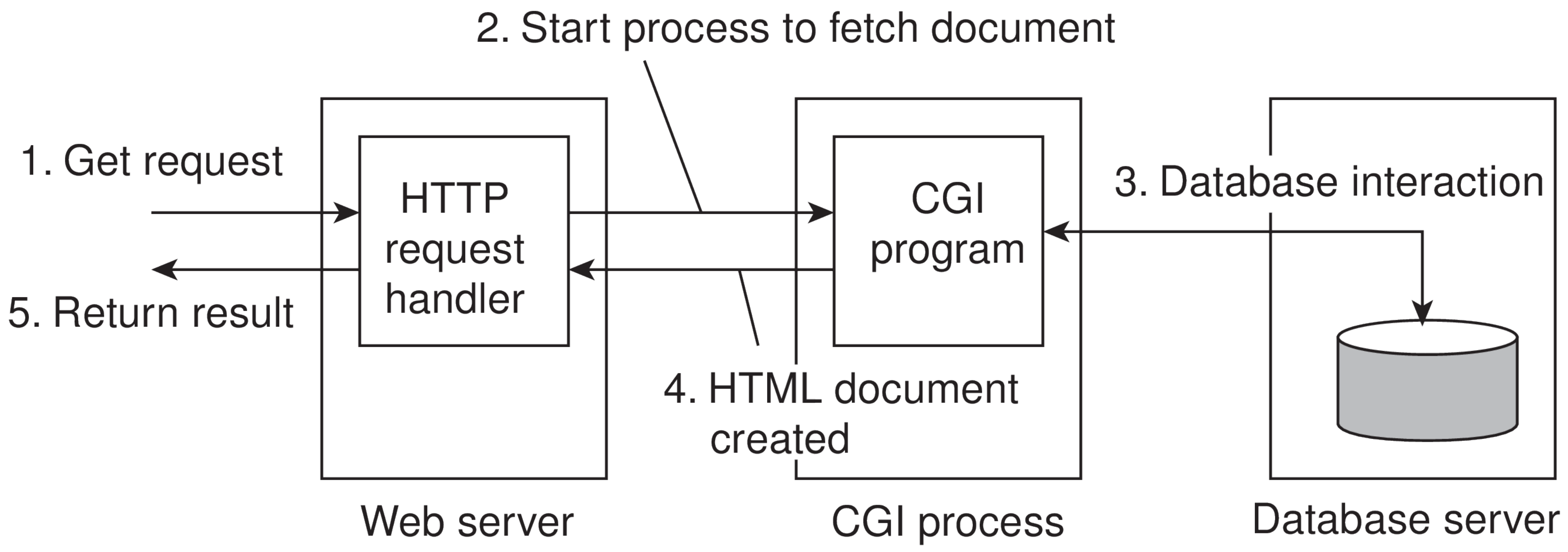
2. Server fetches document from local file
Client machine
Server machine
Browser
Web server
OS
3. Response
1. Get document request (HTTP)

Example: The Web

- Documents
  - Simple documents
  - Marked up with HyperText Markup Language (HTML)
  - Common Gateway Interface (CGI)
    - Web server executes a program based on user data
    - After processing, result is served as a document to the client

Example: The Web

- CGI uses two-tiered server



Example: The Web

- Documents
  - Simple documents
  - Marked up with HyperText Markup Language (HTML)
  - Common Gateway Interface (CGI)
  - Server Side Scripts
    - Server executes a script embedded in the document
    - Document gets altered
      - `<strong> <?php echo $_SERVER['REMOTE_ADDR']; ?> </strong>`

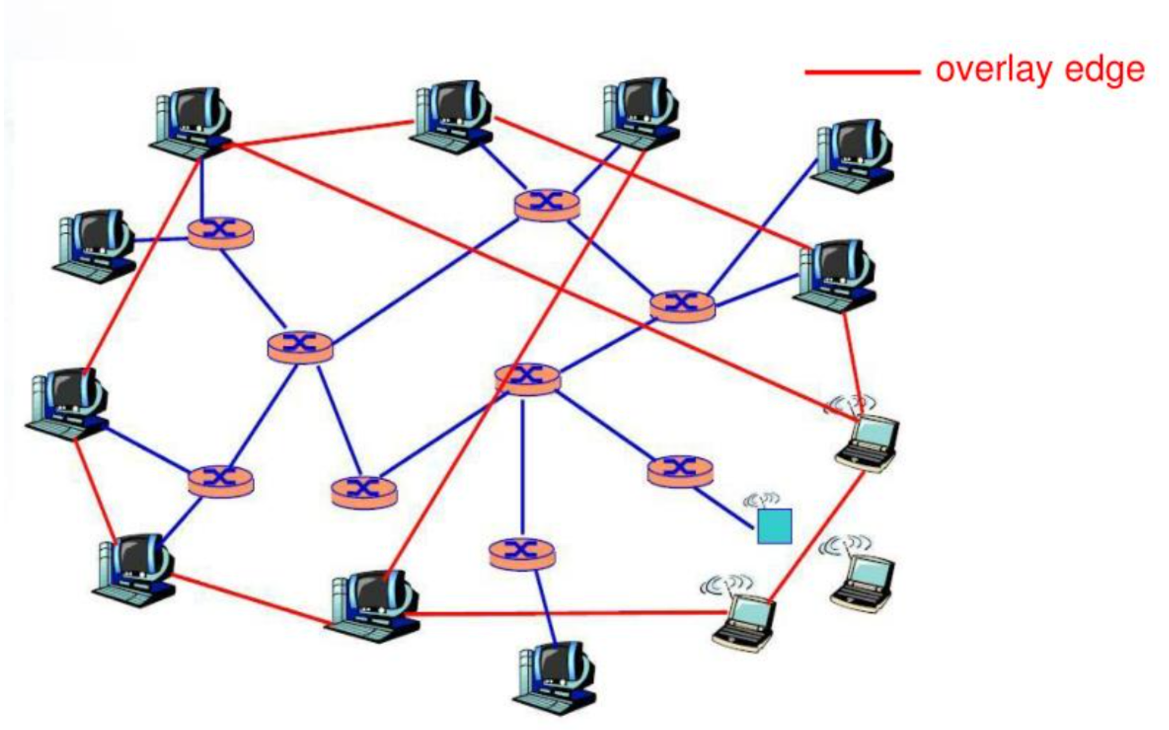
Symmetrically distributed system architectures

Symmetrically distributed system architectures

- Vertical distribution:
  - Multi-tiered architecture
- Horizontal distribution:
  - client or server physically split up into logically equivalent parts
  - each part is operating on its own share of the complete data set
  - E.g. P2P systems

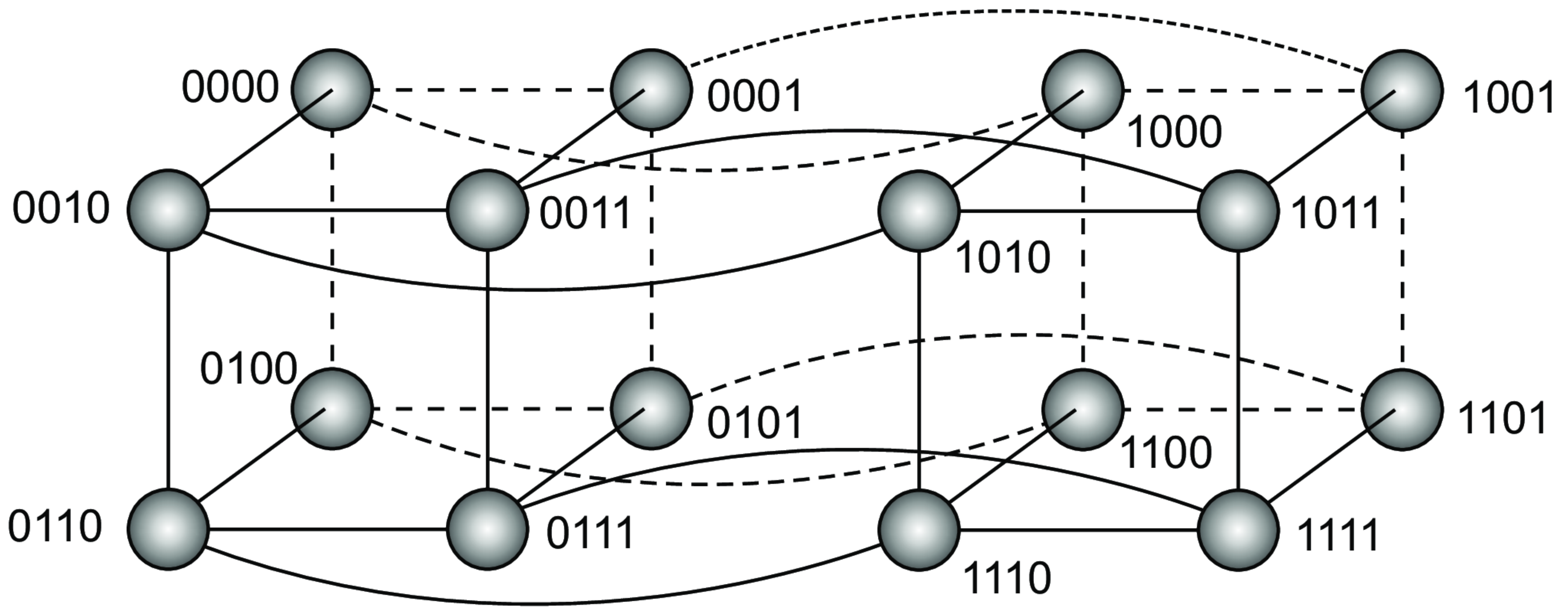
P2P

- Peers are (assumed to be) equal:
  - Processes acts as clients and servers
- P2P use an overlay network
  - Communication channels established over a network
  - Often TCP



Structured P2P

- Nodes in overlay network follow a given topology
  - Example: Hypercube



Structured P2P

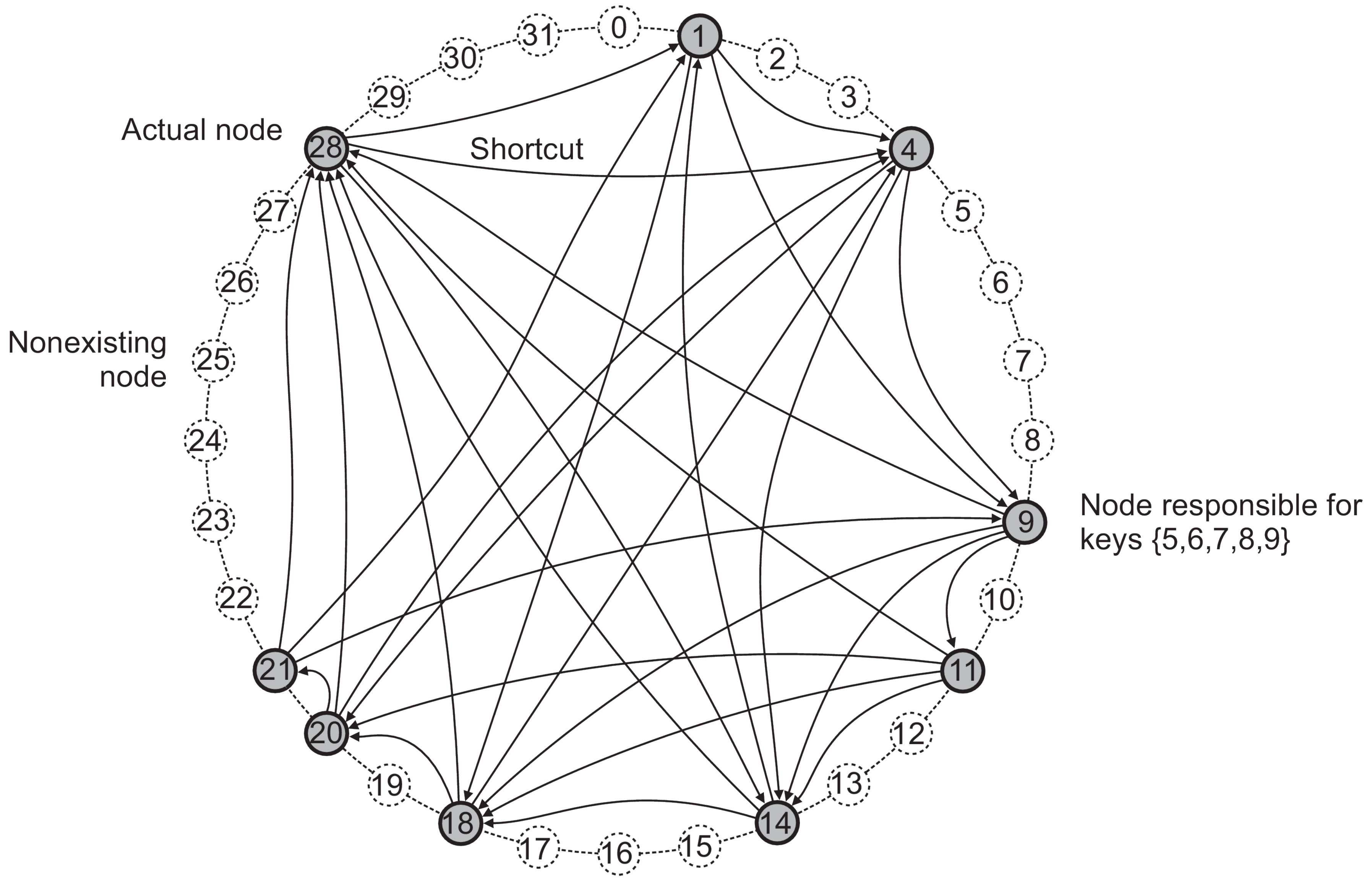
- Topology is used to look up data
  - P2P system implements a Distributed Hash Table
    - Stores key-value pairs
  - Key-Value pairs where the key is the hash of the value

Structured P2P

- Example: Hypercube
  - We are 0011 and look for item with
    - Hash 10010100…01
  - Located at 1001
    - Send message to 1011, which forwards to 1001
    - Document gets returned on the same route

Structured P2P

- Chord: Peers arranged in a single virtual circle



Structured P2P

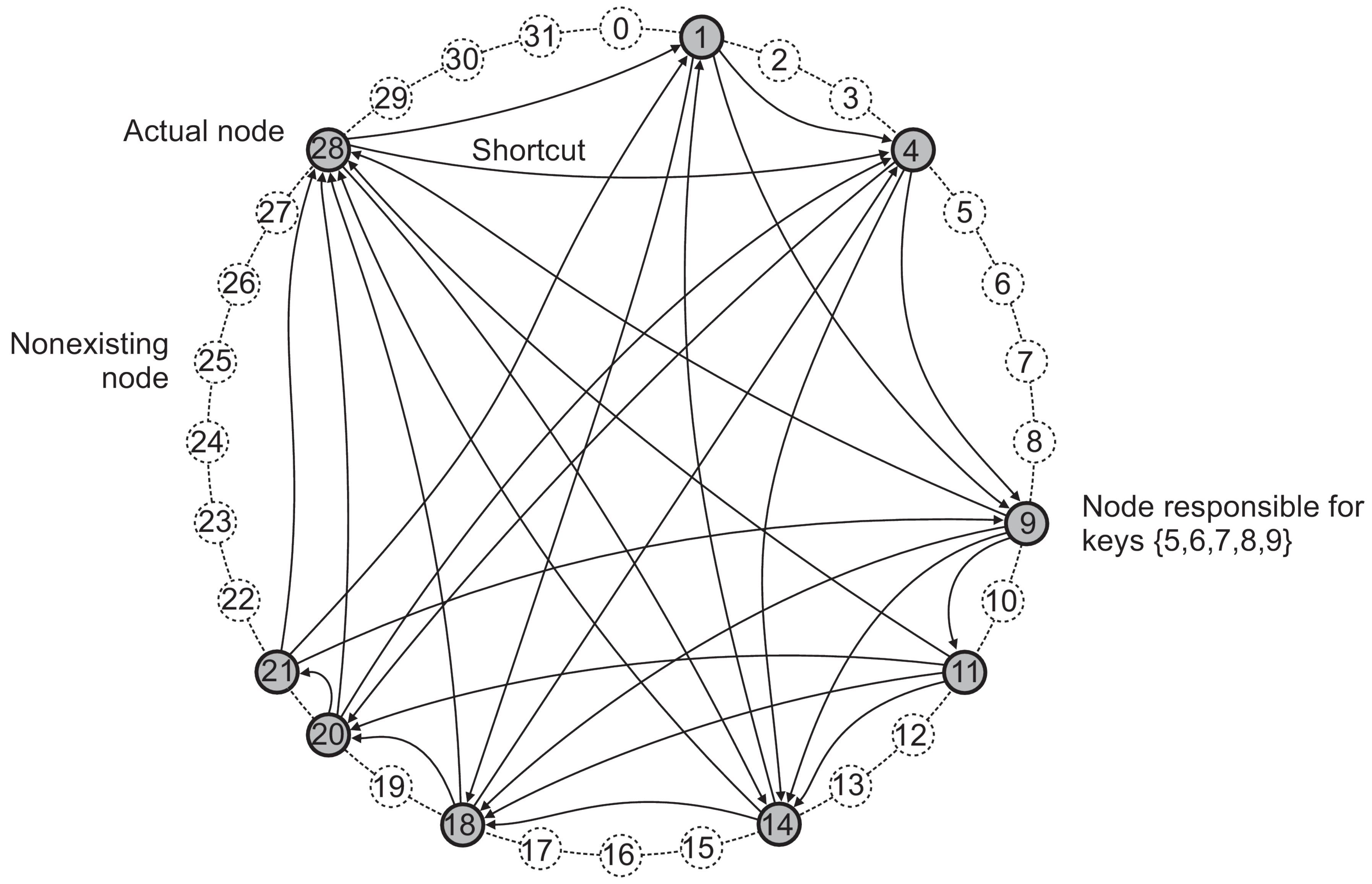
- Chord
  - A joining peer gets a random binary identifier
  - Needs to be able to access one peer
  - Finds predecessor and successor
  - Predecessor sets the next link to the new peer
  - New peer sets its next link to the successor
  - Peer stores all items from successor with hashes $\leq$ it’s id

Structured P2P

- Chord:
  - To find item with a given hash:
    - Send request along the next links until it reaches the correct peer
  - To speed up searches:
    - Peers maintain short-cut links
      - Which might or might not work

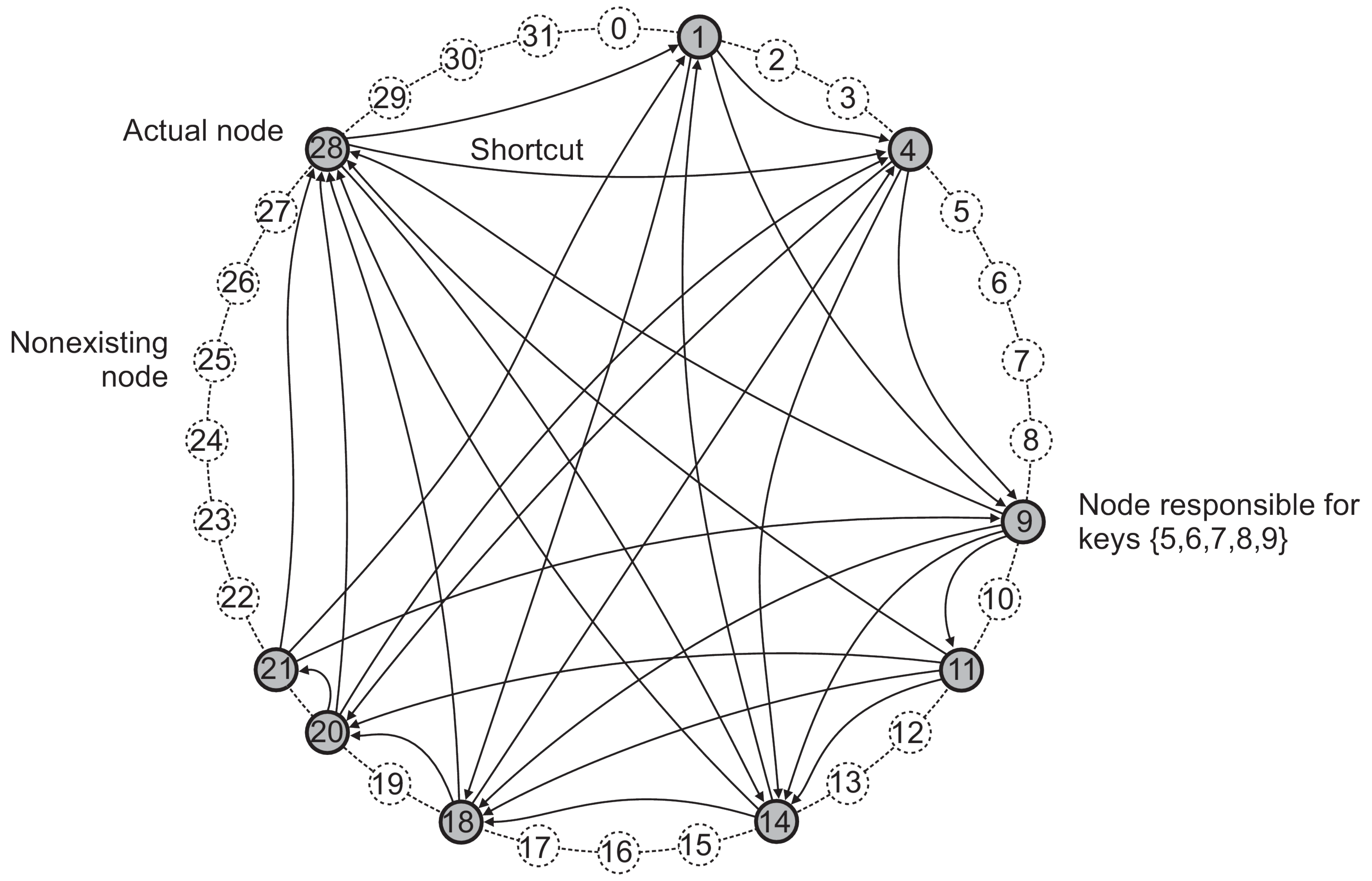
Structured P2P

- Chord



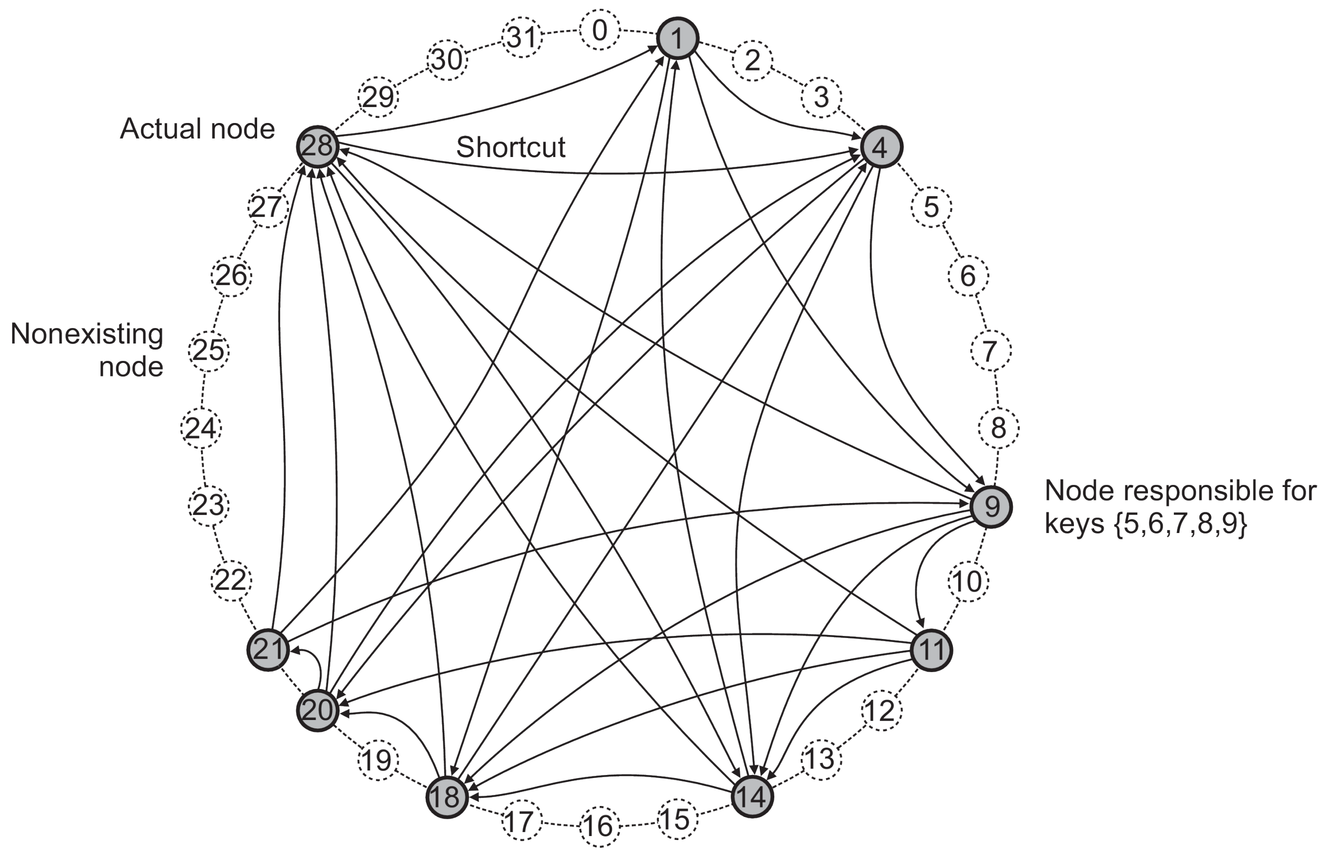
Structured P2P

- Chord: We are in 28. Find item with hash 24?



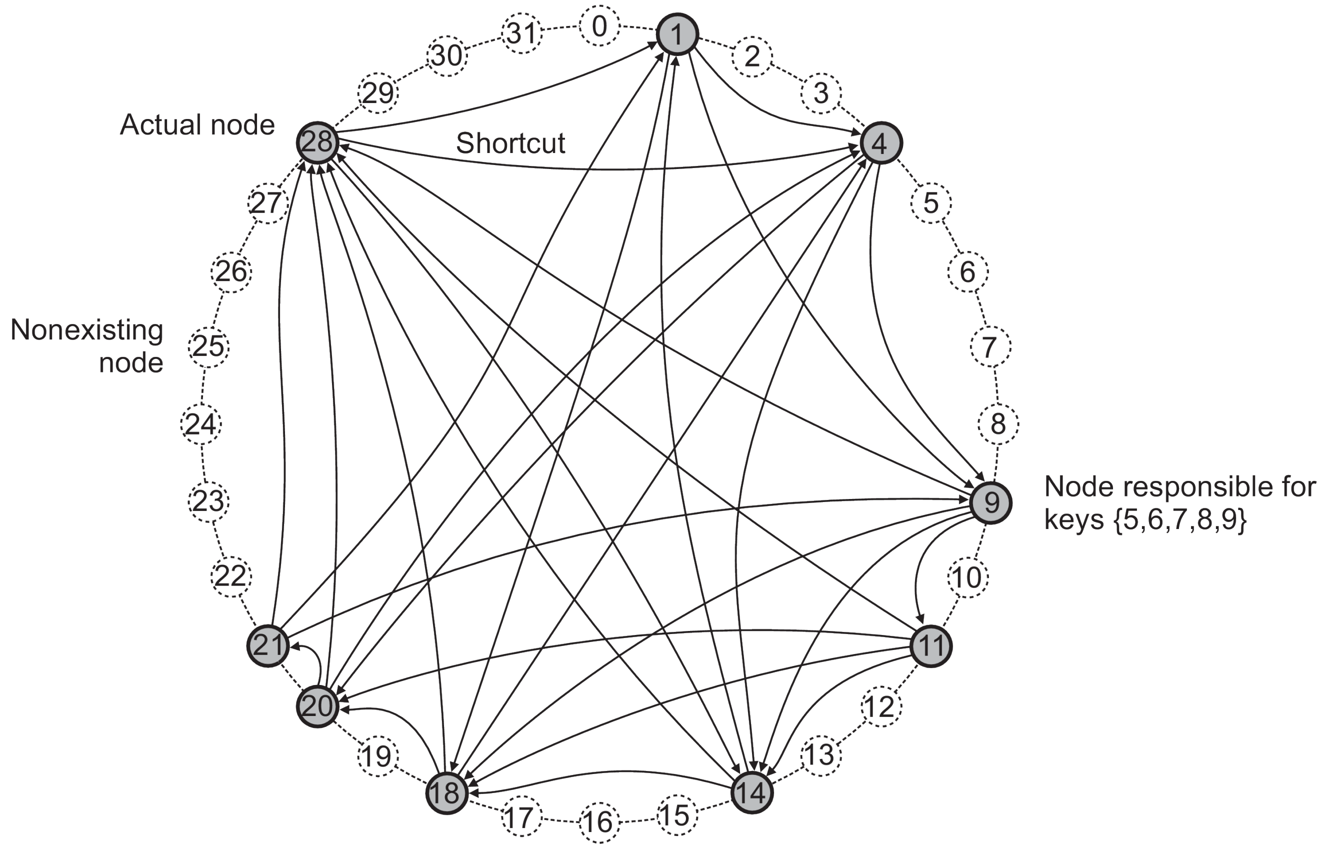
Structured P2P

- Chord: We are in 28. Find item with hash 24?
    - We have shortcuts to 4, 14.



Structured P2P

- Chord: We are in 28. Find item with hash 24?
  - We have shortcuts to 4, 14.
  - Pick the highest one ≤ 24



Unstructured P2P

- E.g Gnutella
  - Each node has a list of some peers
  - Upon joining, a peer get’s a list of peers from the contacted peer
  - Peers constantly update their lists of peers
    - by randomly dropping
    - and by randomly adding discovered peers

Unstructured P2P

- To find something:
  - Flooding
    - A peer contact its contacts
    - Contacts contact their contacts
    - … up to a certain number of iterations
      - Use Time To Live field

Unstructured P2P

- Random walk
  - Searching peer select one or n neighbor and sends request
  - While a peer receives the request:
    - If it has resource, then sends it to originator
    - Otherwise: selects a single neighbor and forward request
  - Use TTL or contact originator to check whether the walk should be stopped.

Unstructured P2P

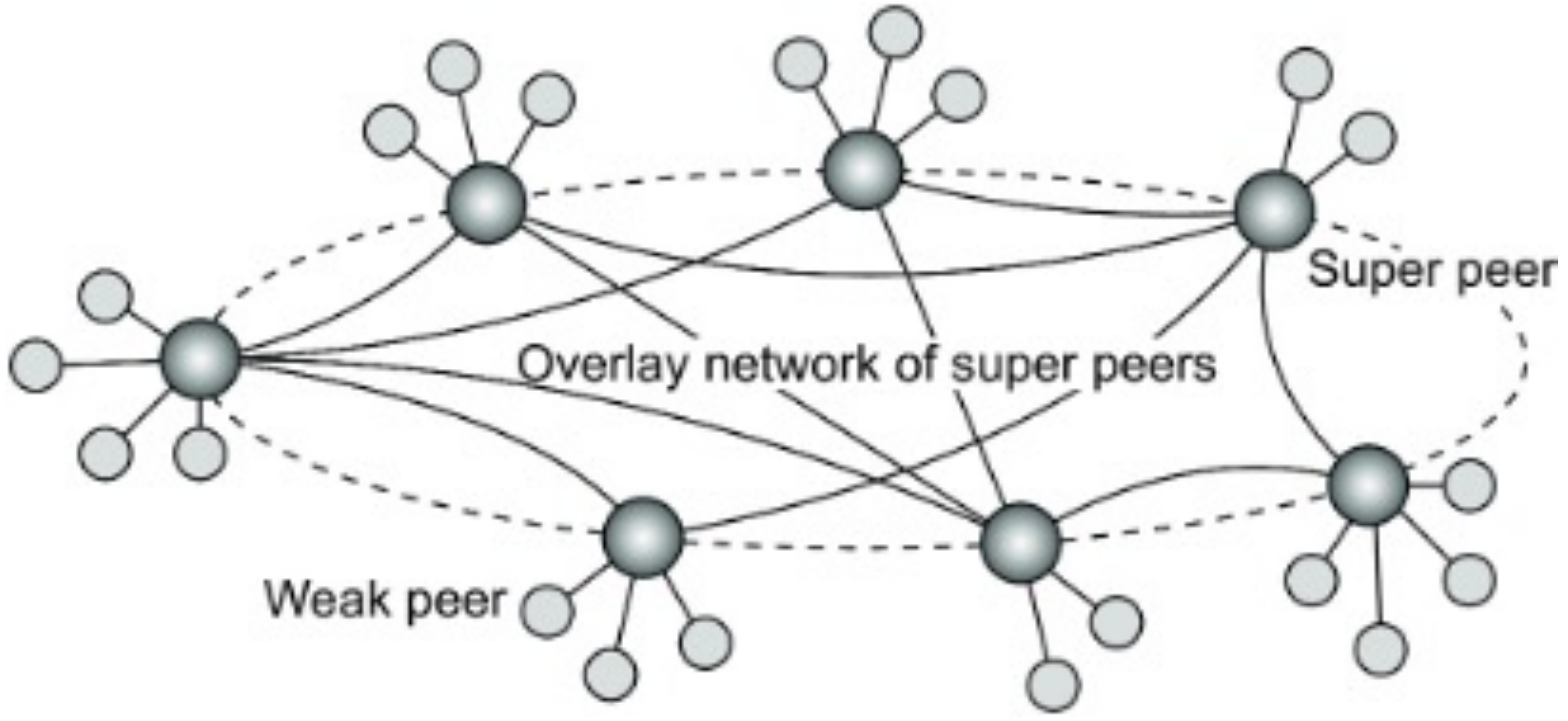
- Policy-based searches in UP2P
  - Examples:
    - A peer updates its contact list based on responsiveness to queries
    - Prefer contacts with large lists of neighbors for flooding,

Hierarchically Structured P2P

- Use special nodes that keep an index of available resources
- Use special nodes that act as brokers
  - Collect data on capacity and use
  - Allows to allocate resources to the best suited peers

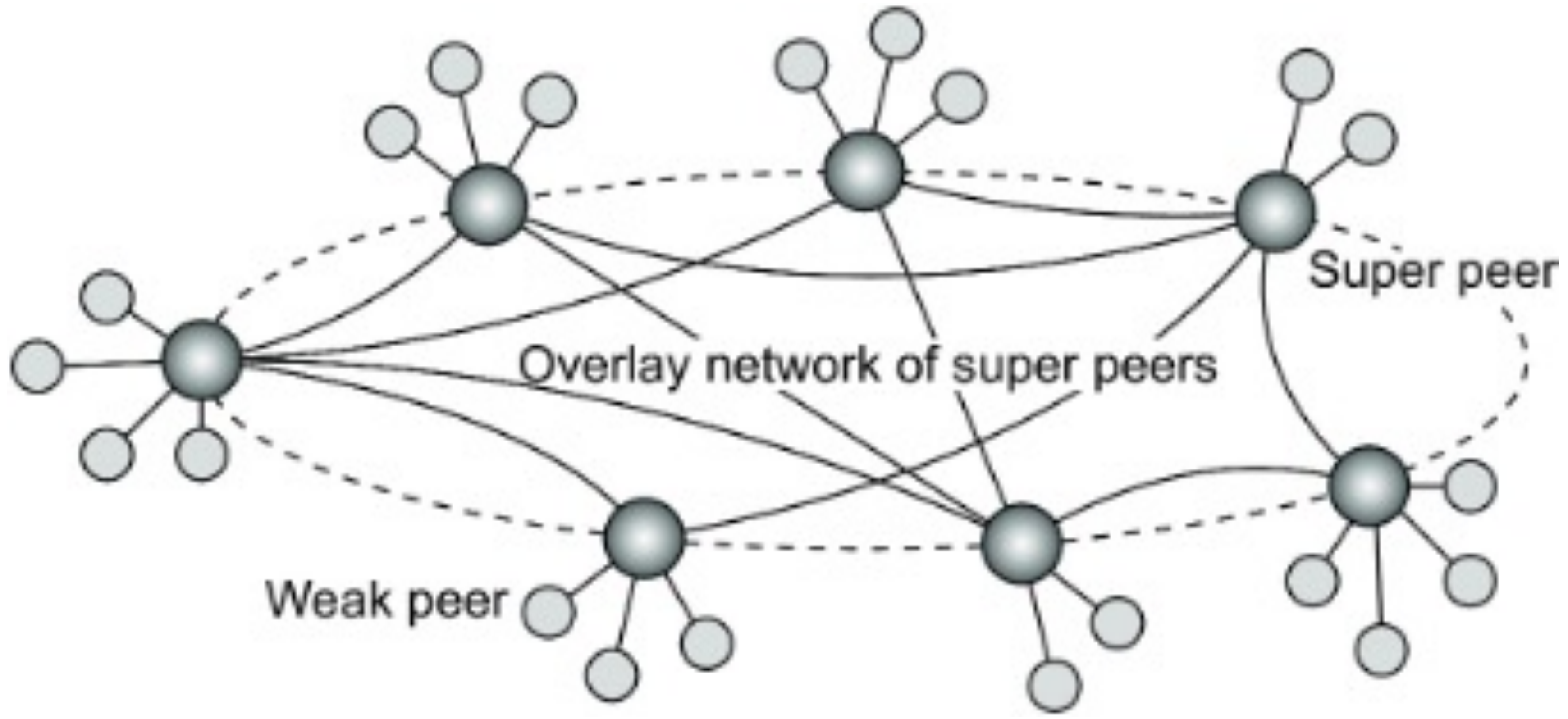
Hierarchically Structured P2P

- Introduces a distinction between peers
    - E.g. super-peers and weak pears



Hierarchically Structured P2P

- A weak peer does not have to be attached to the same super-peer, but can change based on past experience



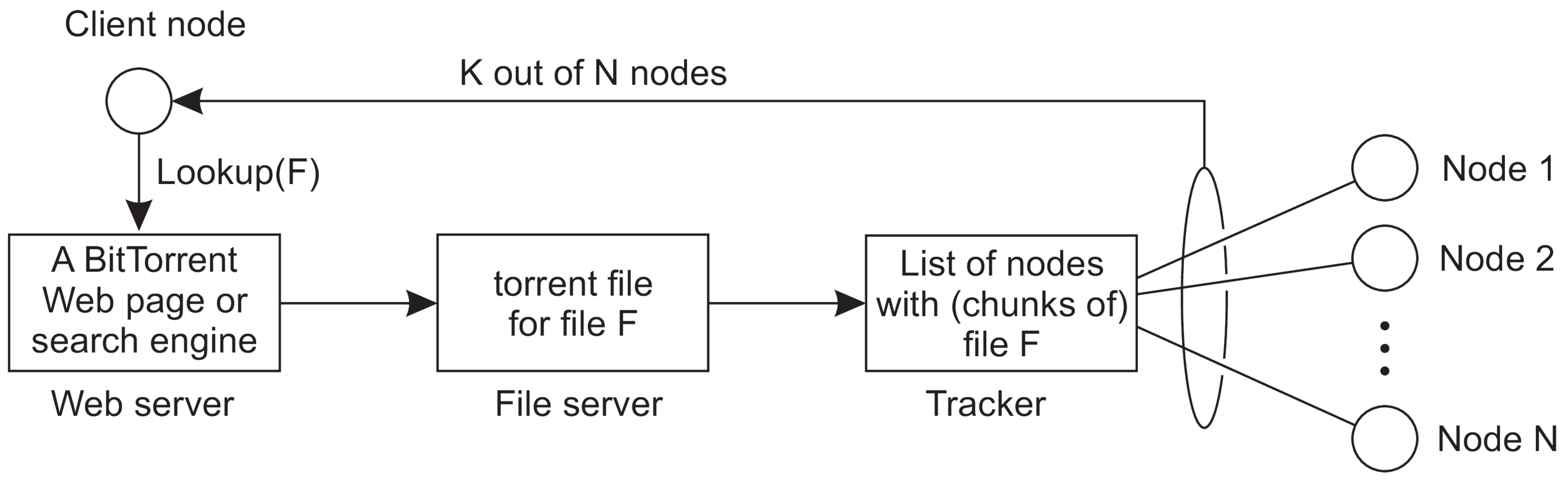
- Super-peer selection is the next problem

Example: Bit Torrent

- Bit Torrent: File-downloading system
  - Force downloaders to be active participants
  - Prevent *“free riding”*

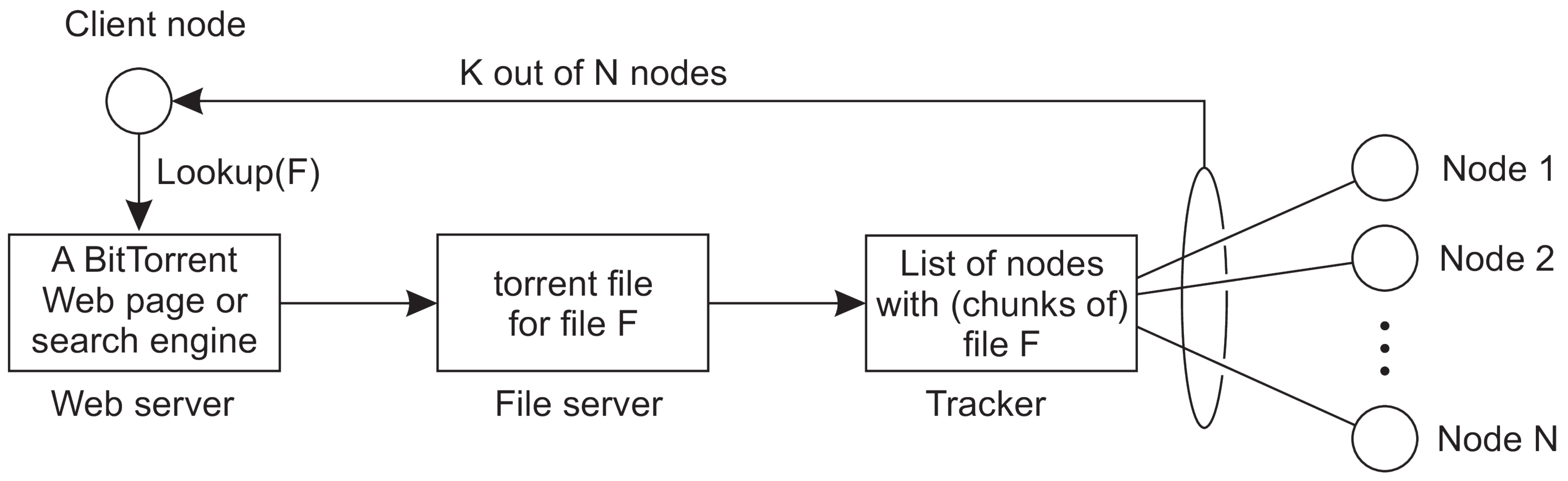
Example: Bit Torrent

- Client looks up file at a Bit Torrent web page
  - This is a global directory
- Lookup result is a “*Torrent File*”
  - Torrent file has a link to a *tracker*



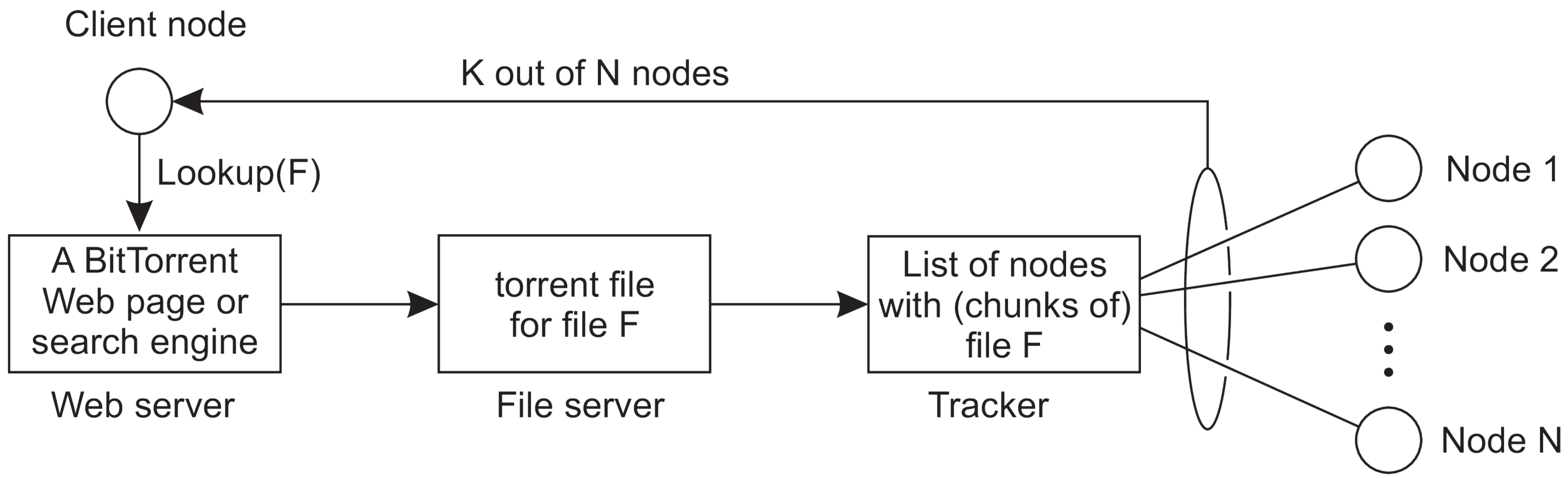
Example: Bit Torrent

- Trackers has a list of active servers with chunks of the desired file
    - Usually a single tracker per file
- Tracker adds requesting node to its list
- Requesting node joins a *swarm*



Example: Bit Torrent

- Gets a chunk from a peer in the swarm
- Then trades the chunk for another chunk with a peer in the swarm



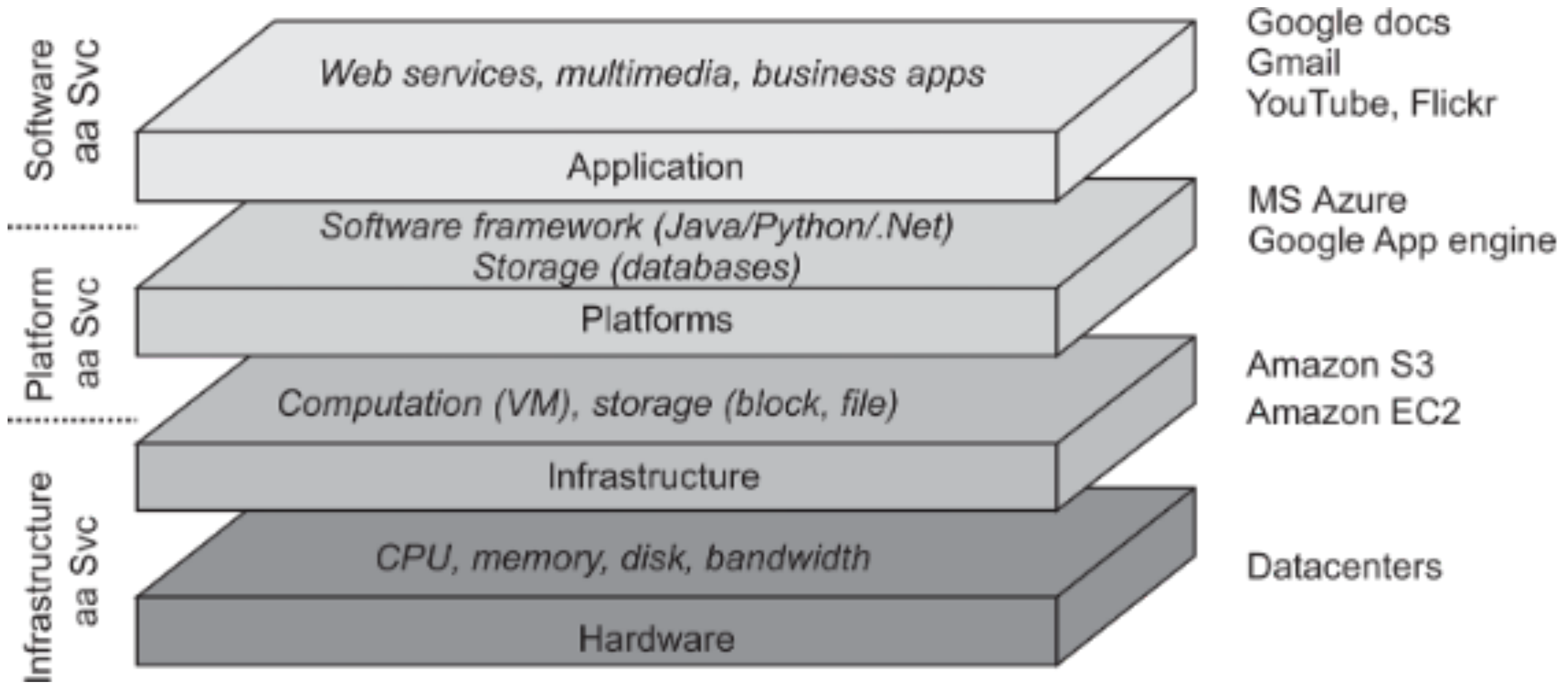
Hybrid System Solutions

- Real-world solutions often combine many architectures
- Some distributed systems cross organizational boundaries

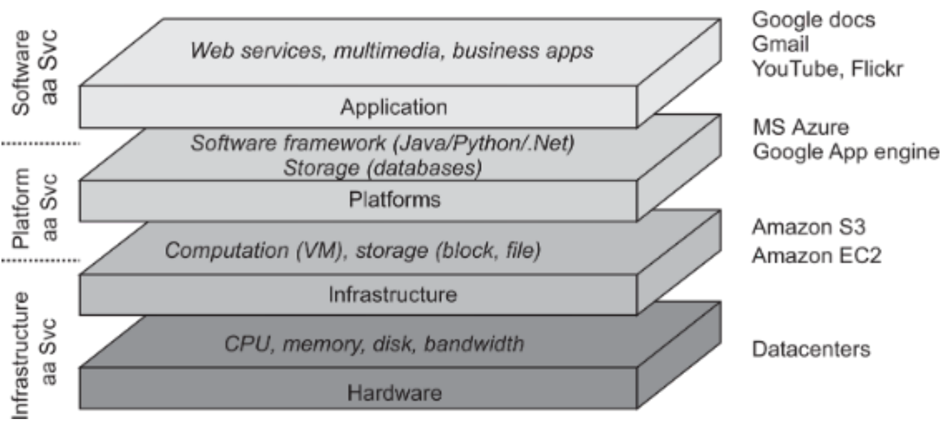
Cloud Computing

- Organizations with data centers have looked for ways to open up their facilities to paying clients
  - *Utility computing*: Clients upload tasks to a data center and are charged on a per-resource basis
  - Morphed into cloud computing:
    - Virtualization of resources
    - Service-Level Agreements (SLA)

Cloud Computing



Cloud Computing



- Make a distinction between four layers
  - Hardware: Processors, routers, power and cooling systems. Customers normally never get to see these.
  - Infrastructure: Deploys virtualization techniques. Evolves around allocating and managing virtual storage devices and virtual servers.
  - Platform: Provides higher-level abstractions for storage and such.
    - Example: Amazon S3 storage system offers an API for (locally created) files to be organized and stored in so-called buckets.
  - Application: Actual applications, such as office suites (text processors, spreadsheet applications, presentation applications). Comparable to the suite of apps shipped with OS.

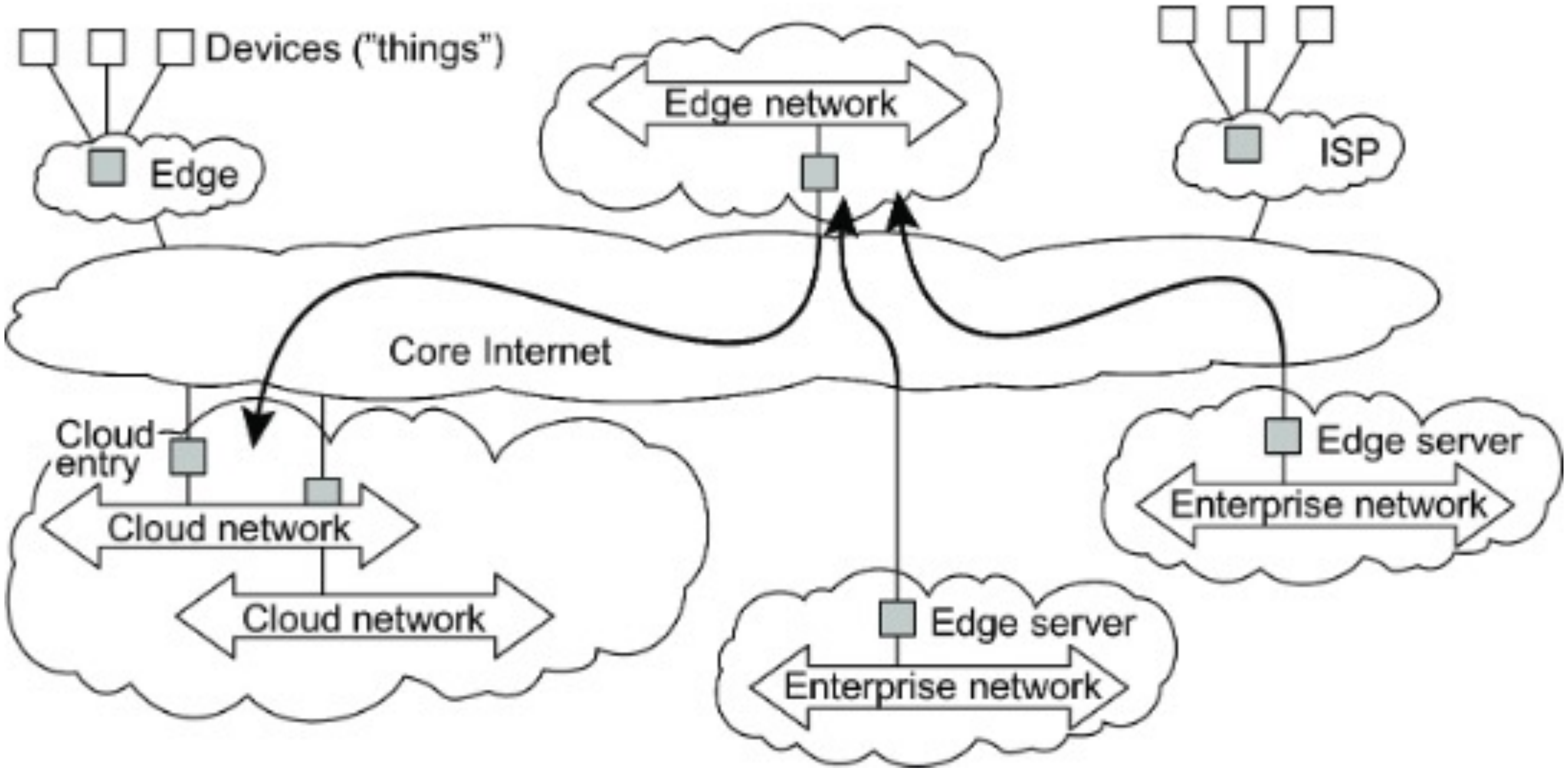
Cloud Computing

- Driven by:
  - Provisioning: Only pay for what you currently use
  - Outsourcing of IT & Security
  - Agility
  - Clustering
  - …

Edge Computing

- **I**nternet **o**f **T**hings
- Moving big data from IoT to cloud is not efficient:
  - bandwidth, privacy, low-latency requirements
- Edge computing
  - Move computation closer to IoT devices
- Fog-computing: Computing, storage, networking, data management at within close vicinity of IoT devices
- Mist computing: computation at networks, sensors and actuators that make up IoT components

Edge Computing



Edge Computing

- Commonly (and often misconceived) arguments
  - Latency and bandwidth: Especially important for certain real-time applications, such as augmented/virtual reality applications. Many people underestimate the latency and bandwidth to the cloud.
  - Reliability: The connection to the cloud is often assumed to be unreliable, which is often a false assumption. There may be critical situations in which extremely high connectivity guarantees are needed.
  - Security and privacy: The implicit assumption is often that when assets are nearby, they can be made better protected. Practice shows that this assumption is generally false. However, securely handling data operations in the cloud may be trickier than within your own organization

Edge Computing

- Managing resources at the edge may be trickier than in the cloud
  - Resource allocation: we need to guarantee the availability of the resources required to perform a service.
  - Service placement: we need to decide when and where to place a service. This is notably relevant for mobile applications.
  - Edge selection: we need to decide which edge infrastructure should be used when a service needs to be offered. The closest one may not be the best one.
- Observation
  - There is still a lot of buzz about edge infrastructures and computing, yet whether all that buzz makes any sense remains to be seen.

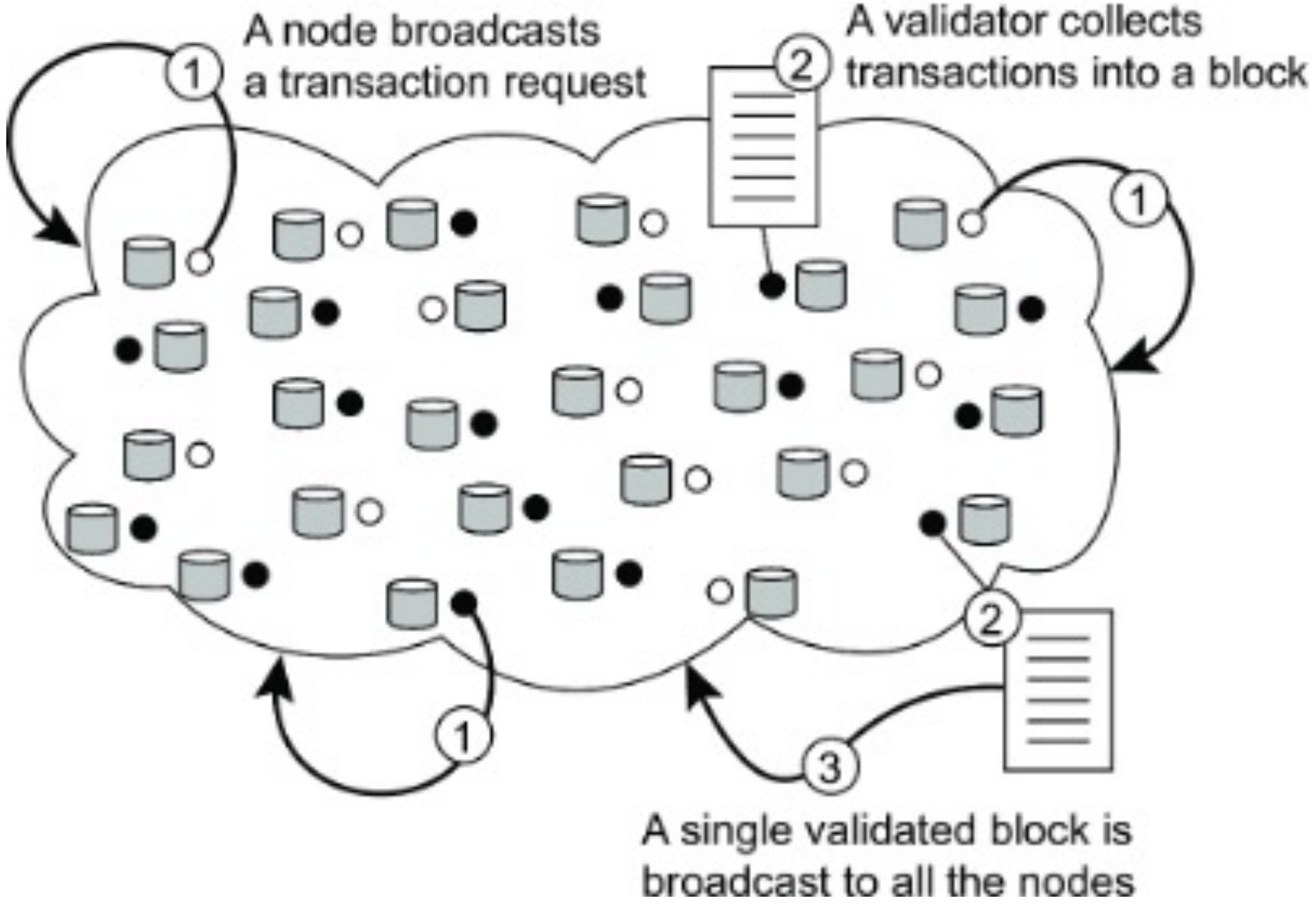
Block Chain Architecture

- **Distributed ledgers:** Enable registration of transactions
- Transaction system:
  - A transaction is
    - validated
    - effectuated
    - stored for auditing purposes

Block Chain Architecture

- Can have a single (or a few) **trusted third party**
  - that acts as a notary
- Lack of trust:
  - Large set of participants who jointly register transactions among them in a public ledger

Block Chain Architecture


1
A node broadcasts a transaction request
2
A validator collects transactions into a block
1
1
2
3
A single validated block is broadcast to all the nodes

Block Chain Architecture

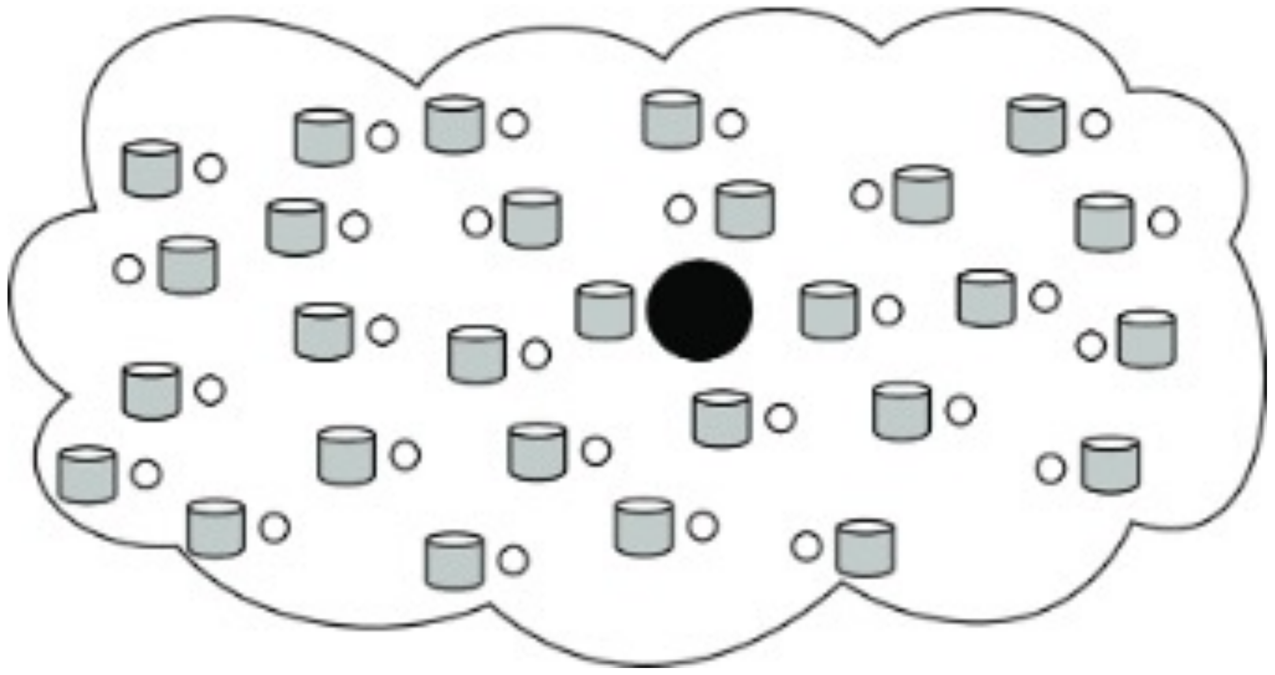
- There is logically only a single chain of blocks with validated transactions
- Each block is immutable
- Different block chain systems use different modes of
    - Who carries out validation

Summary

- Distributed systems can be organized in many ways

Block Chain Architecture

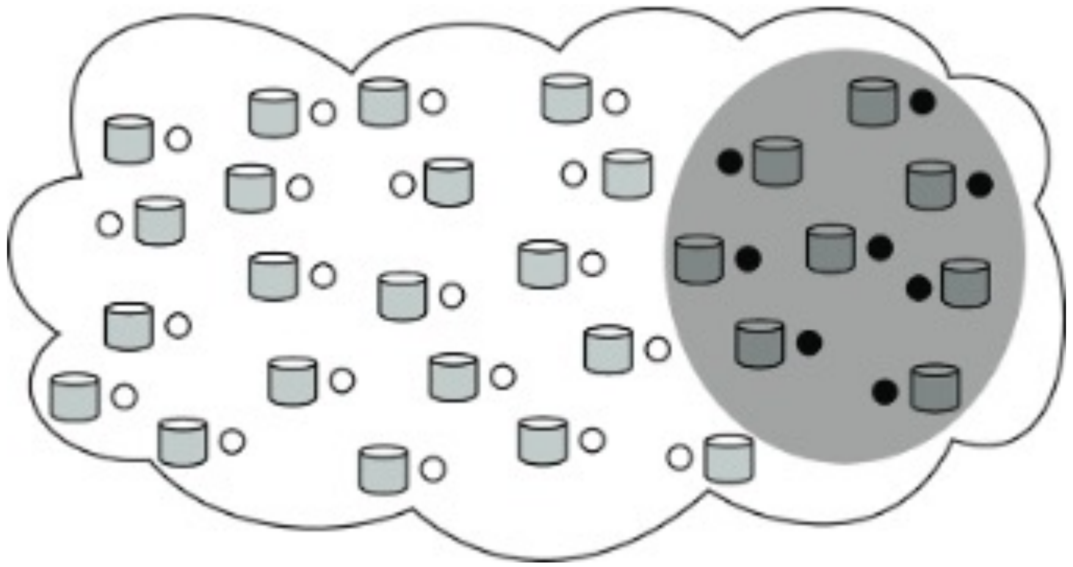
- Centralized selection of validator
  - A single entity decides who can validate



- against block chain principles

Block Chain Architecture

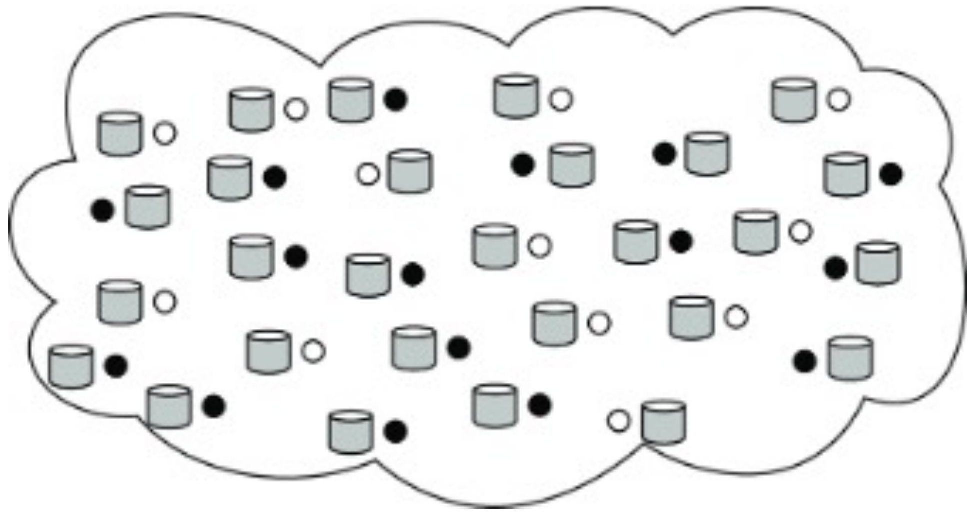
- Distributed solution with permissions



- A selected, relatively small group of servers jointly reach consensus on which validator can go ahead.
- None of these servers needs to be trusted, as long as roughly two-thirds behave according to their specifications.
- In practice, only a few tens of servers can be accommodated.

Block Chain Architecture

- Decentralized selection of validator



- Participants select a leader
- Only the elected leader is allowed to append a block of validated transactions.
- Large-scale, decentralized leader election that is fair, robust, secure, and so on, is far from trivial.

Block Chain Architecture

Block Chain Architecture