

Distributed Systems

Principles and Paradigms

Maarten van Steen

VU Amsterdam, Dept. Computer Science
Room R4.20, steen@cs.vu.nl

Chapter 02: Architectures

Version: October 25, 2009



Contents

Chapter
01: Introduction
02: Architectures
03: Processes
04: Communication
05: Naming
06: Synchronization
07: Consistency & Replication
08: Fault Tolerance
09: Security
10: Distributed Object-Based Systems
11: Distributed File Systems
12: Distributed Web-Based Systems
13: Distributed Coordination-Based Systems

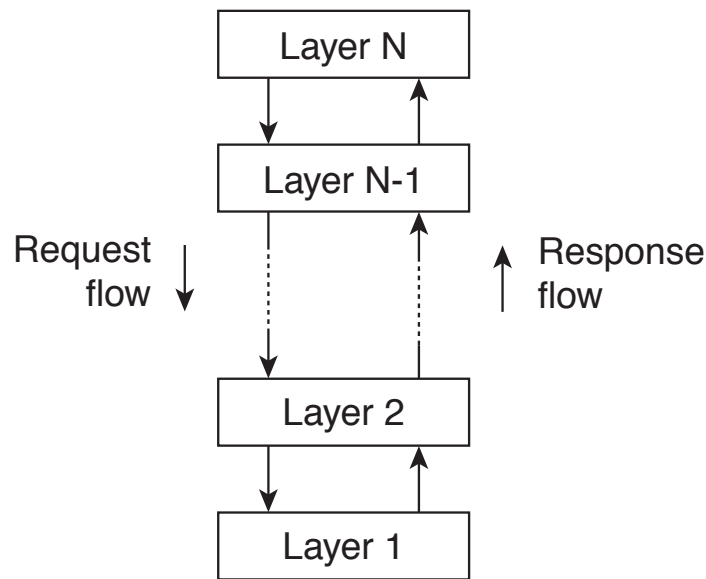
Architectures

- Architectural styles
- Software architectures
- Architectures versus middleware
- Self-management in distributed systems

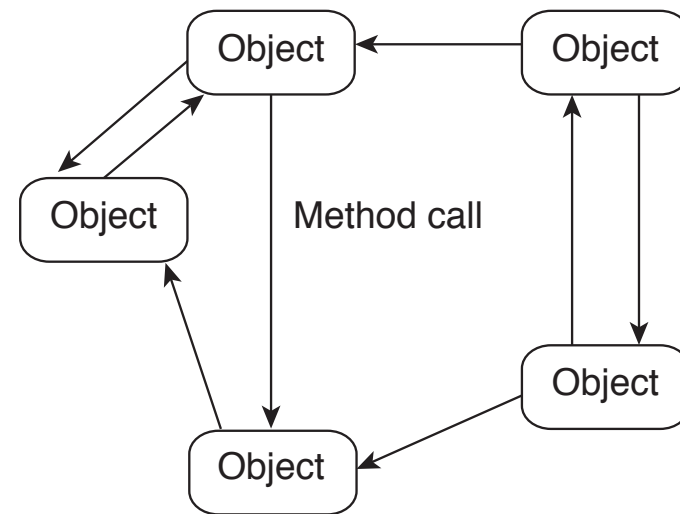
Architectural styles

Basic idea

Organize into **logically different** components, and distribute those components over the various machines.



(a)



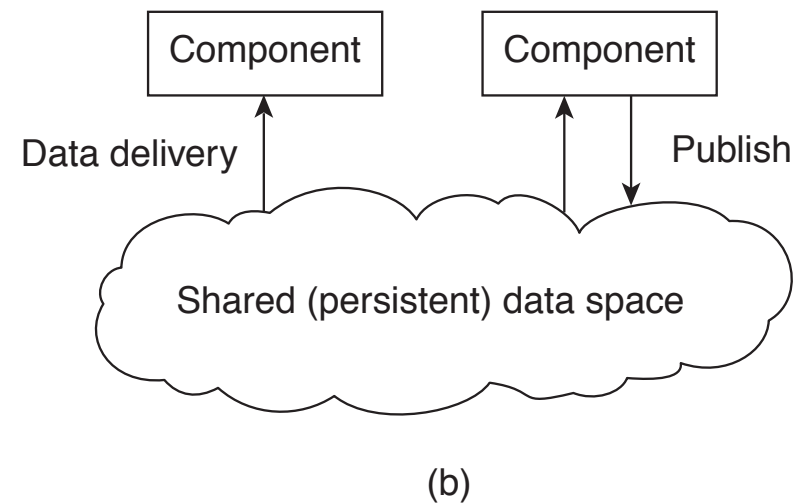
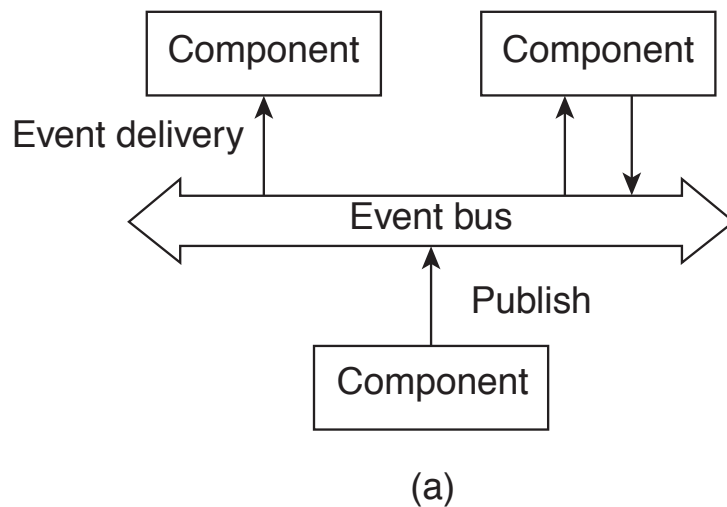
(b)

- (a) Layered style is used for client-server system
- (b) Object-based style for distributed object systems.

Architectural Styles

Observation

Decoupling processes in **space** (“anonymous”) and also **time** (“asynchronous”) has led to alternative styles.



(a) Publish/subscribe [decoupled in **space**]

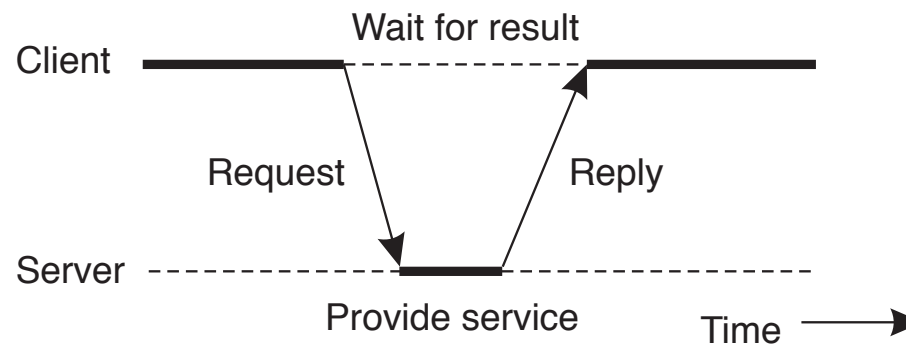
(b) Shared dataspace [decoupled in **space** and **time**]

Centralized 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 wrt to using services



Application Layering

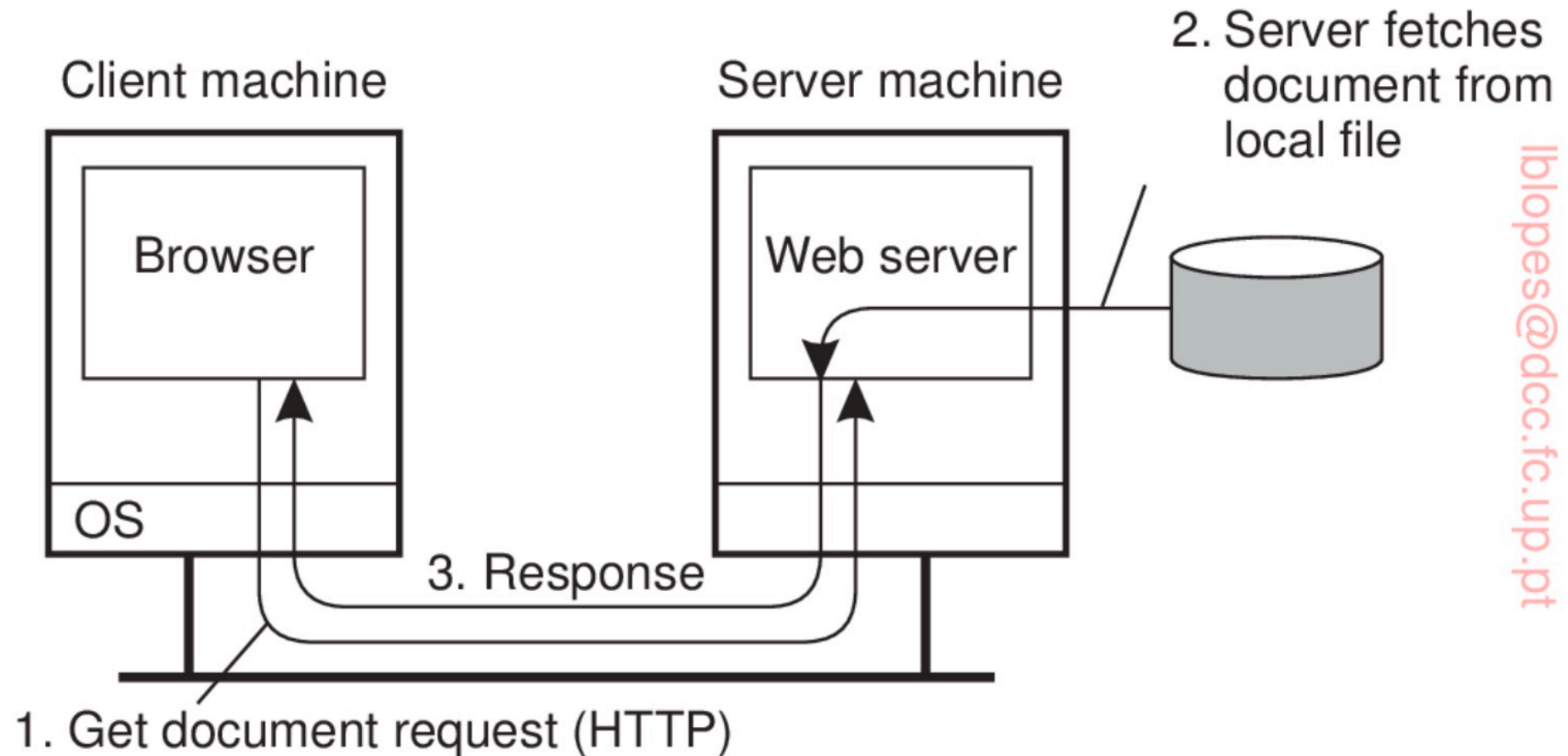
Traditional three-layered view

- User-interface layer contains units for an application's user interface
- 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

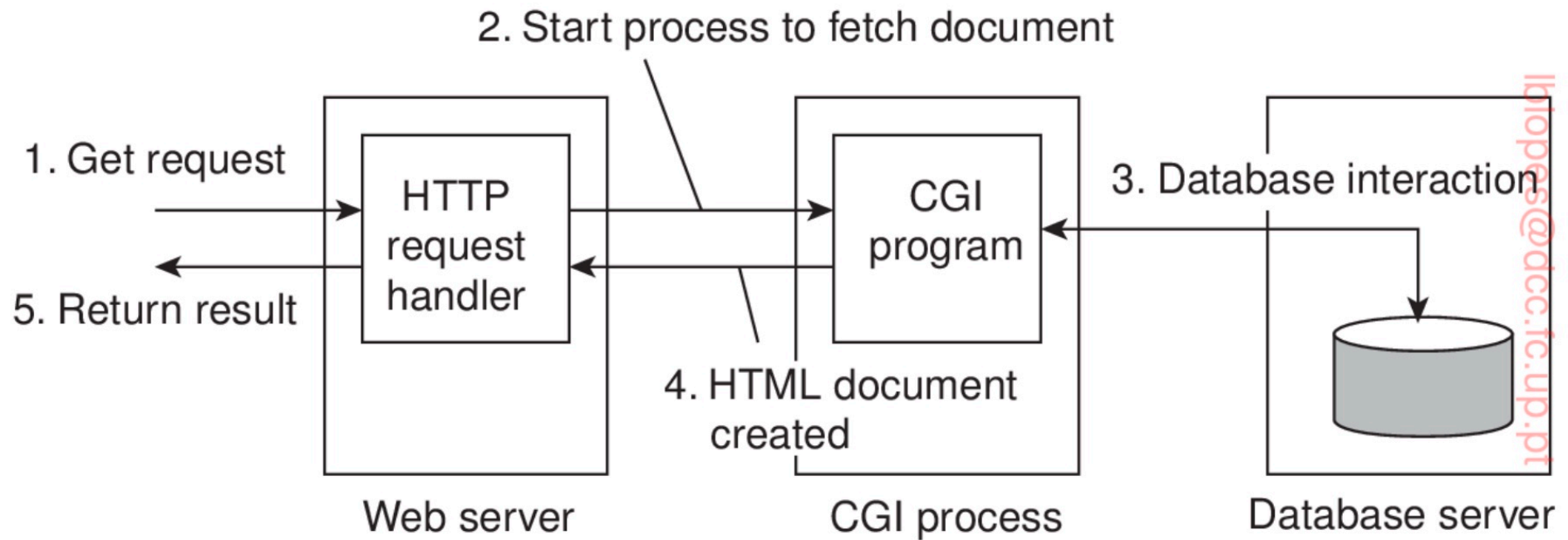
Observation

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

2-layered architecture



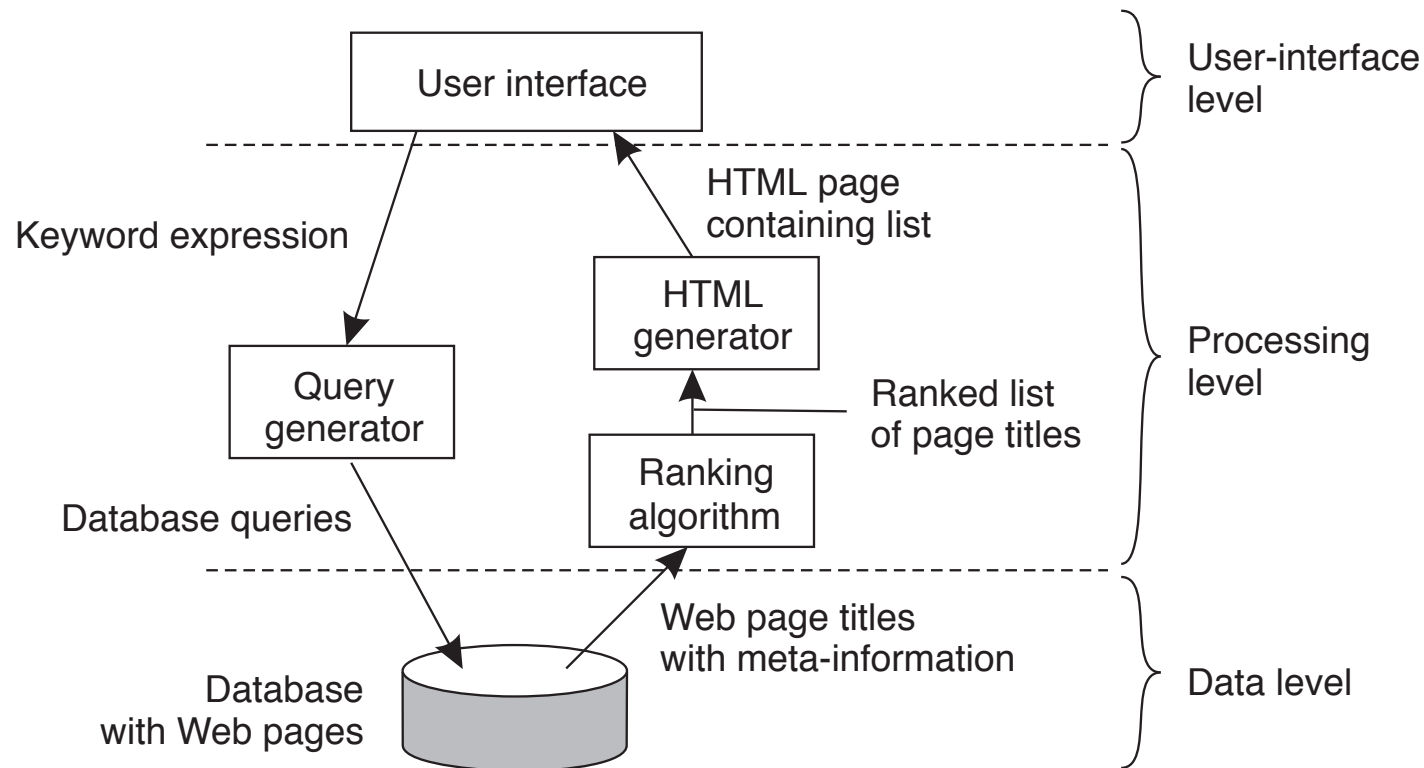
3-layered architecture



CGI = Common Gateway Interface

Application Layering

layers => conceptual organization
of software components



a typical 3-layered architecture: search engines

Multi-Tiered Architectures

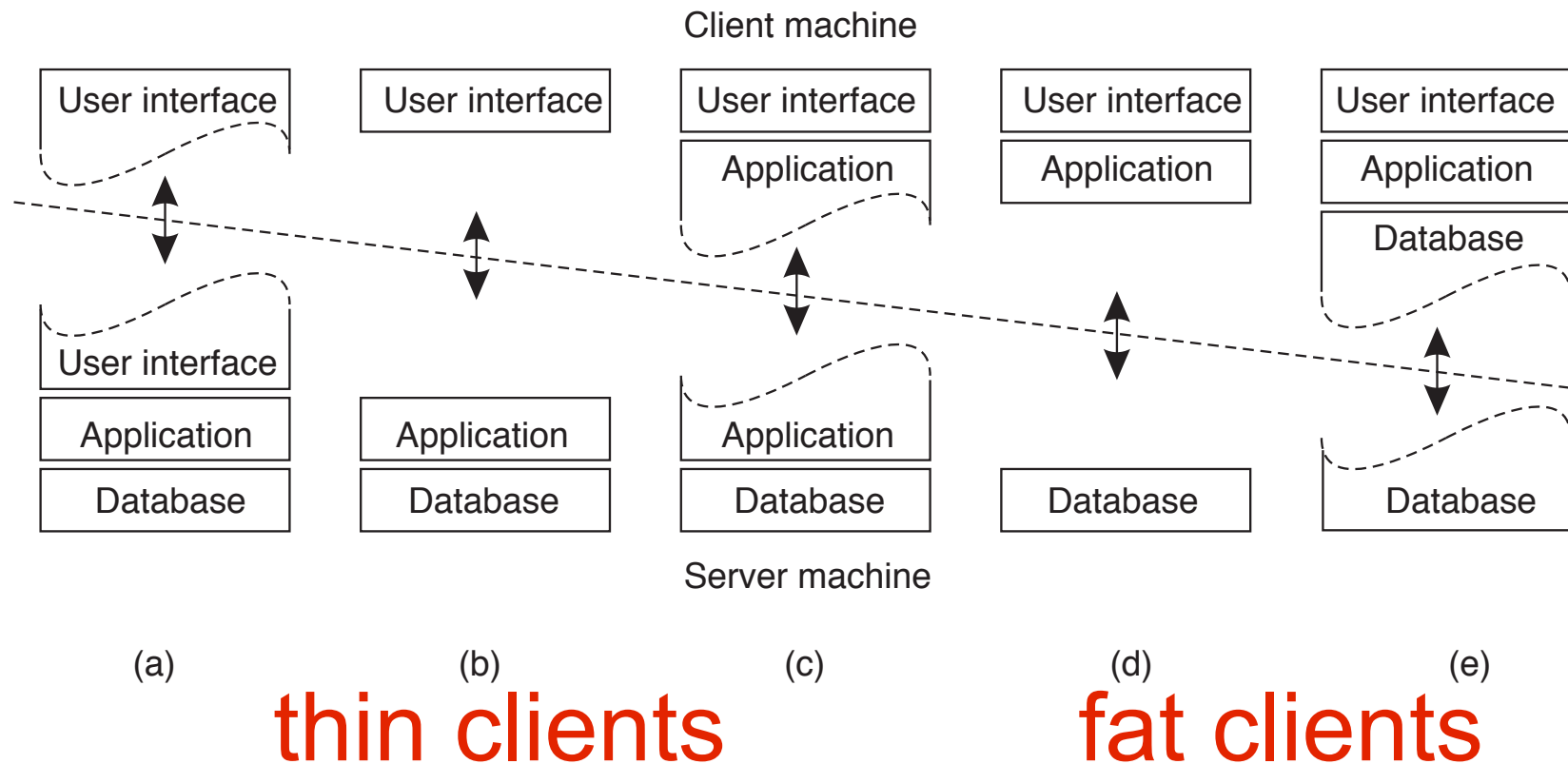
tiers => physical deployment of software components

Single-tiered: dumb terminal/mainframe configuration

Two-tiered: client/single server configuration

Three-tiered: each layer on separate machine

Traditional two-tiered configurations:



Decentralized Architectures

Observation this was back in 2009 ;-)

In the last couple of years we have been seeing a tremendous growth in peer-to-peer systems.

- **Structured P2P**: nodes are organized following a specific distributed data structure
- **Unstructured P2P**: nodes have randomly selected neighbors
- **Hybrid P2P**: some nodes are appointed special functions in a well-organized fashion

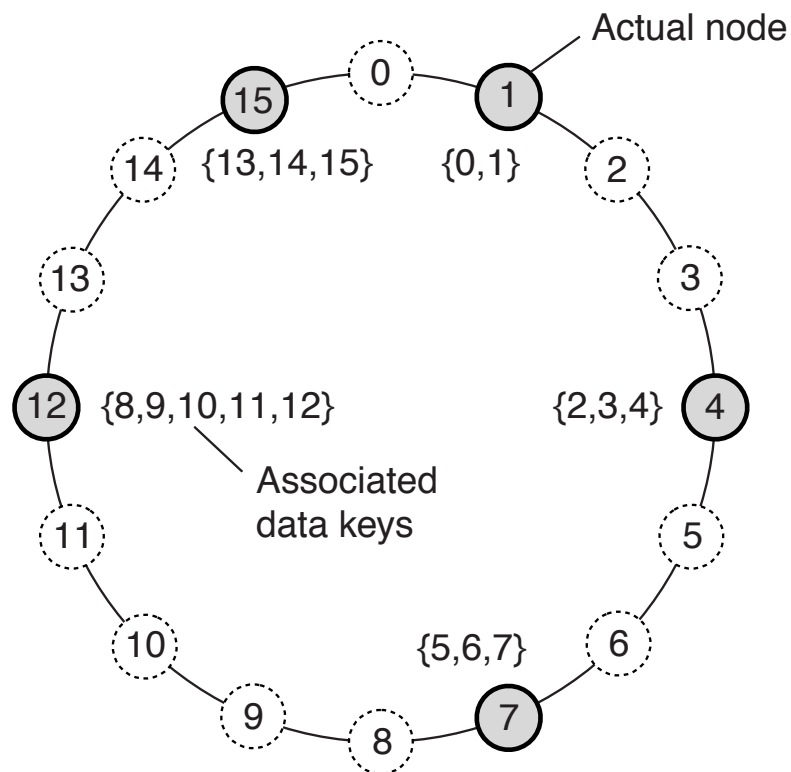
Note

In virtually all cases, we are dealing with **overlay networks**: data is routed over connections setup between the nodes (cf. application-level multicasting)

Structured P2P Systems

Basic idea

Organize the nodes in a structured **overlay network** such as a logical ring, and make specific nodes responsible for services based only on their ID.



Note

The system provides an operation **LOOKUP(key)** that will efficiently **route** the lookup request to the associated node.

Unstructured P2P Systems

Observation

Many unstructured P2P systems attempt to maintain a **random graph**.

Basic principle

Each node is required to contact a randomly selected other node:

- Let each peer maintain a **partial view** of the network, consisting of c other nodes
- Each node P periodically selects a node Q from its partial view
- P and Q exchange information **and** exchange members from their respective partial views

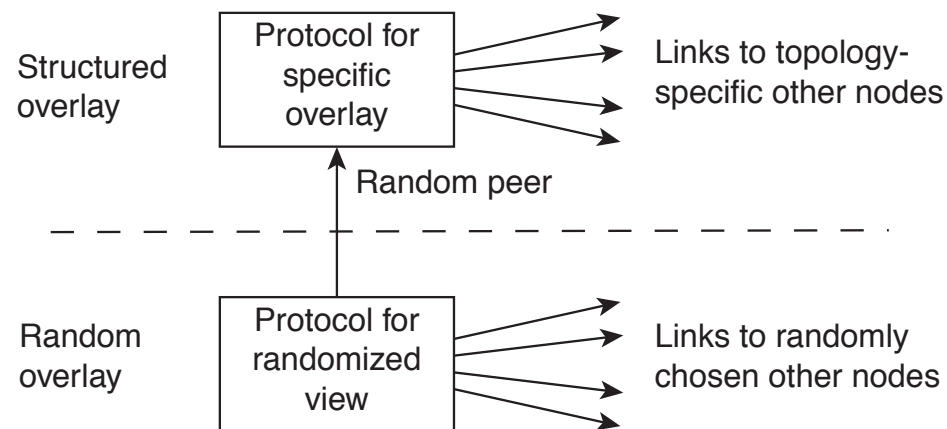
Note

It turns out that, depending on the exchange, randomness, but also **robustness** of the network can be maintained.

Topology Management of Overlay Networks

Basic idea

Distinguish two layers: (1) maintain random partial views in lowest layer; (2) be selective on who you keep in higher-layer partial view.



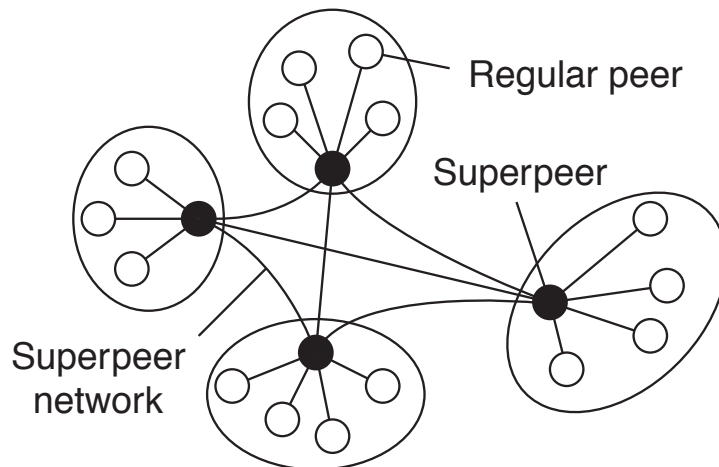
Note

Lower layer **feeds** upper layer with random nodes; upper layer is **selective** when it comes to keeping references.

Superpeers

Observation

Sometimes it helps to select a few nodes to do specific work:
superpeer.



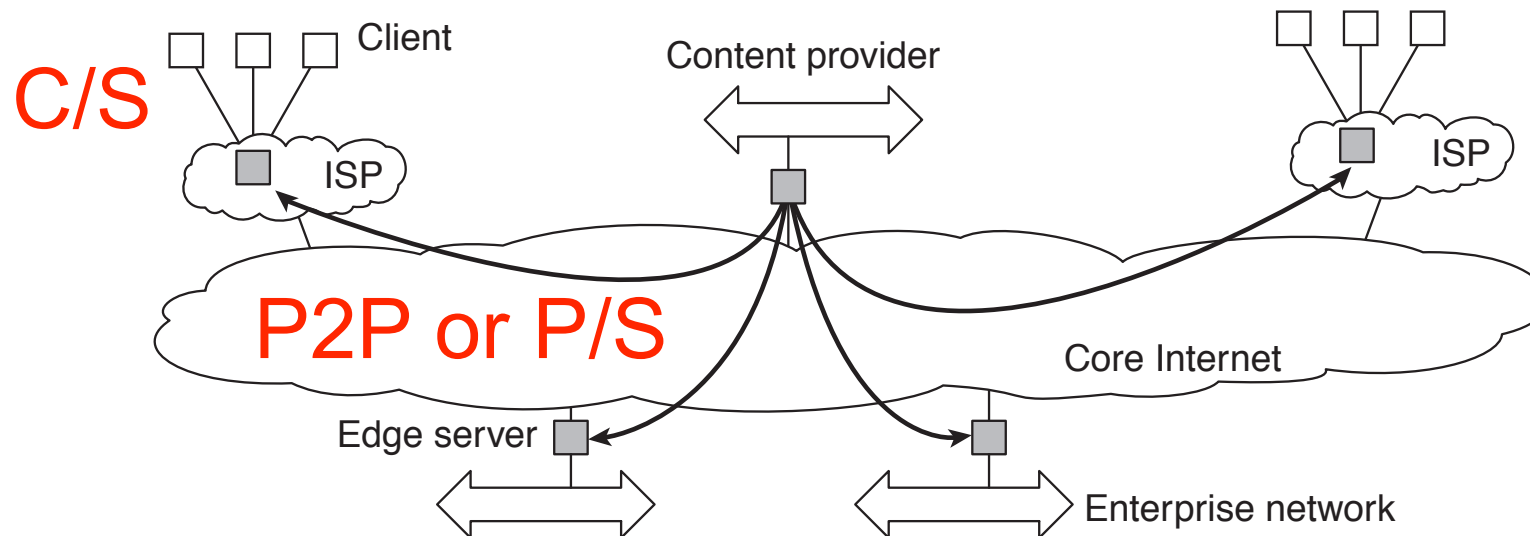
Examples

- Peers maintaining an index (for search)
- Peers monitoring the state of the network
- Peers being able to setup connections

Hybrid Architectures: Client-server combined with P2P

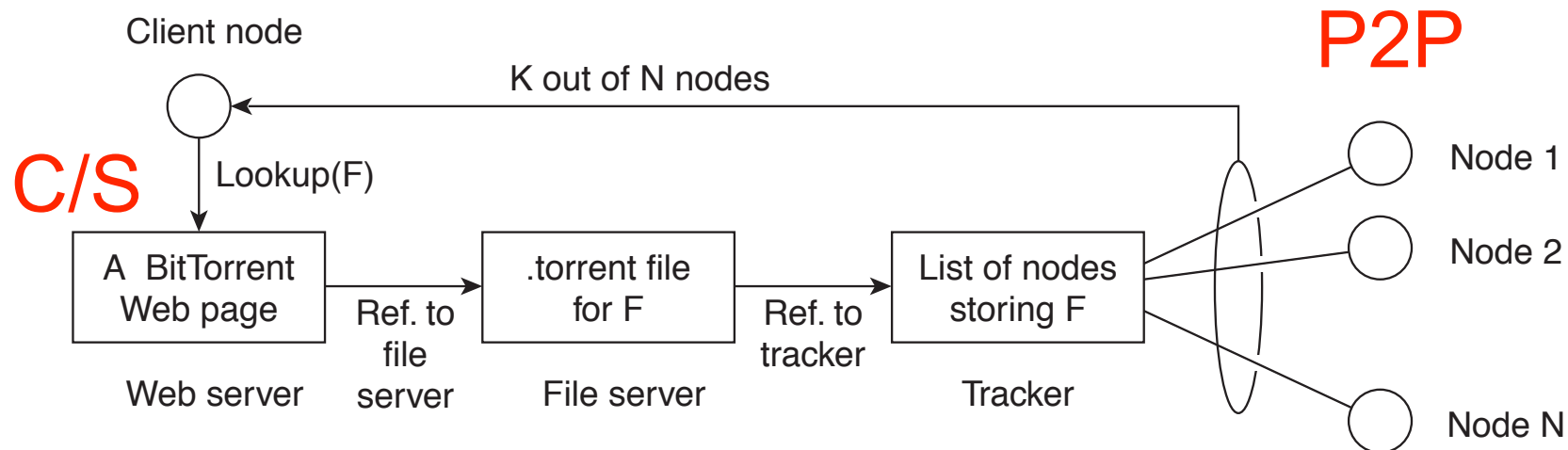
Example

Edge-server architectures, which are often used for **Content Delivery Networks**



C/S + P2P or P/S

Hybrid Architectures: C/S with P2P – BitTorrent



Basic idea

Once a node has identified where to download a file from, it joins a **swarm** of downloaders who **in parallel** get file chunks from the source, but also distribute these chunks amongst each other.

Architectures versus Middleware

Problem

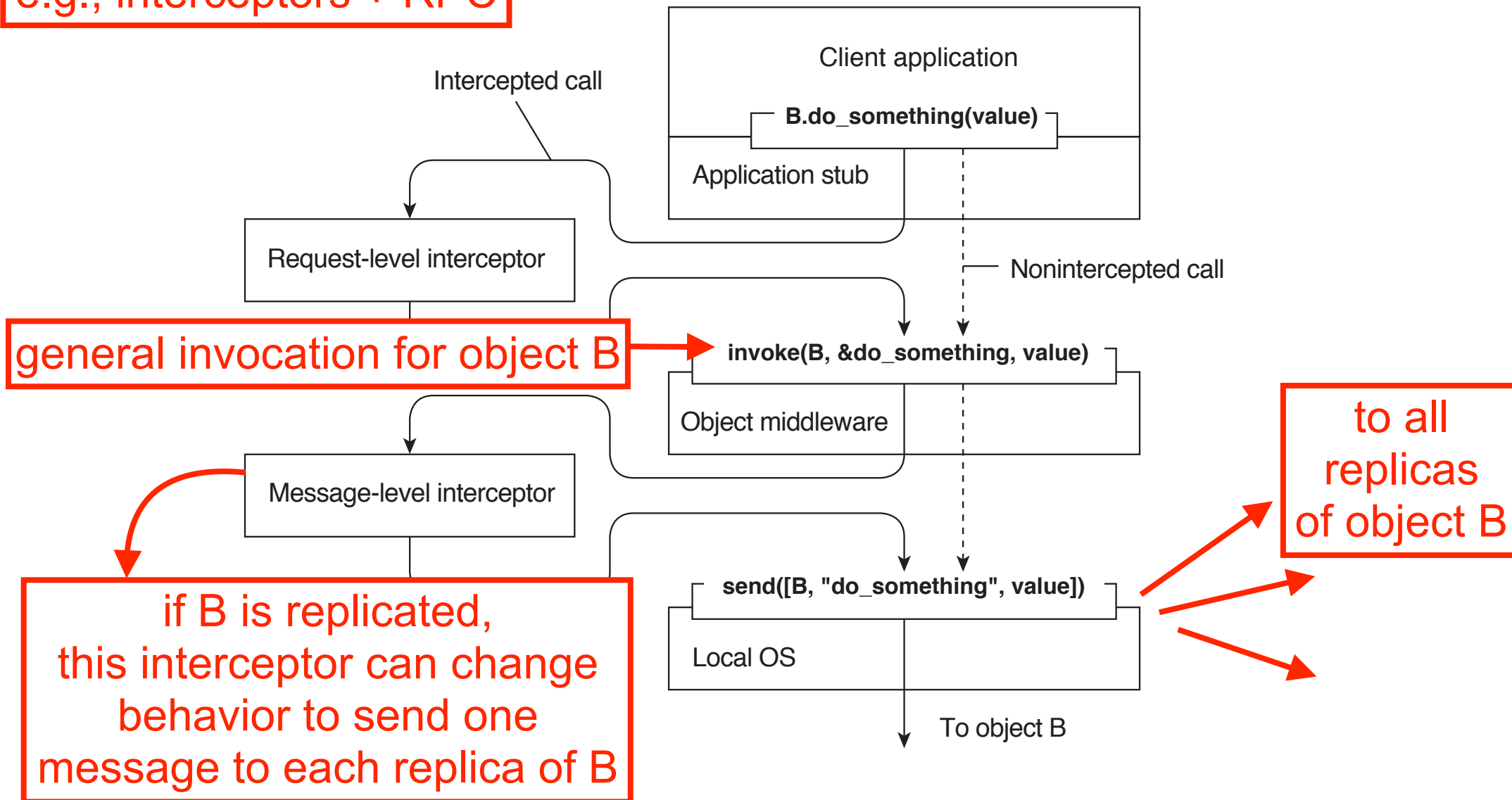
In many cases, distributed systems/applications are developed according to a specific architectural style. The chosen style may not be optimal in all cases \Rightarrow need to (dynamically) **adapt the behavior of the middleware**.

Interceptors

Intercept the usual flow of control when invoking a **remote object**.

Interceptors

e.g., interceptors + RPC



Adaptive Middleware

e.g., aspect-oriented programming



Separation of concerns: Try to separate **extra functionalities** and later **weave** them together into a single implementation \Rightarrow only toy examples so far.

Computational reflection: Let a program inspect itself at runtime and adapt/change its settings dynamically if necessary \Rightarrow mostly at language level and applicability unclear.

Component-based design: Organize a distributed application through components that can be dynamically replaced when needed \Rightarrow highly complex, also many intercomponent dependencies.

e.g., runtime programming



Fundamental question

Do we need adaptive **software** at all, or is the issue adaptive **systems**?

Self-managing Distributed Systems

Observation

Distinction between system and software architectures blurs when **automatic adaptivity** needs to be taken into account:

- Self-configuration
- Self-managing
- Self-healing
- Self-optimizing
- Self-*



software components must
be aware of changing
(e.g., hardware)
environment

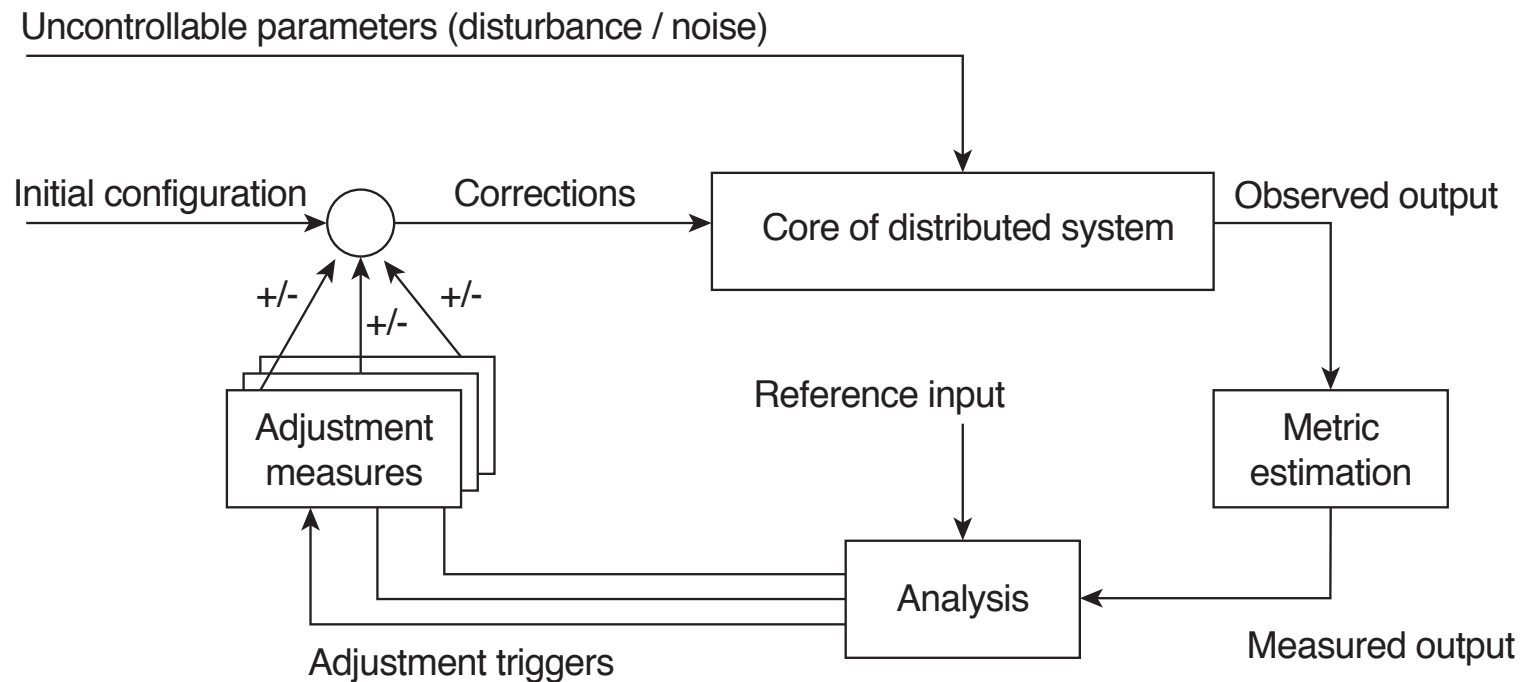
Warning

There is a lot of hype going on in this field of **autonomic computing**.

Feedback Control Model

Observation

In many cases, self-* systems are organized as a **feedback control system**.



Example: Globule

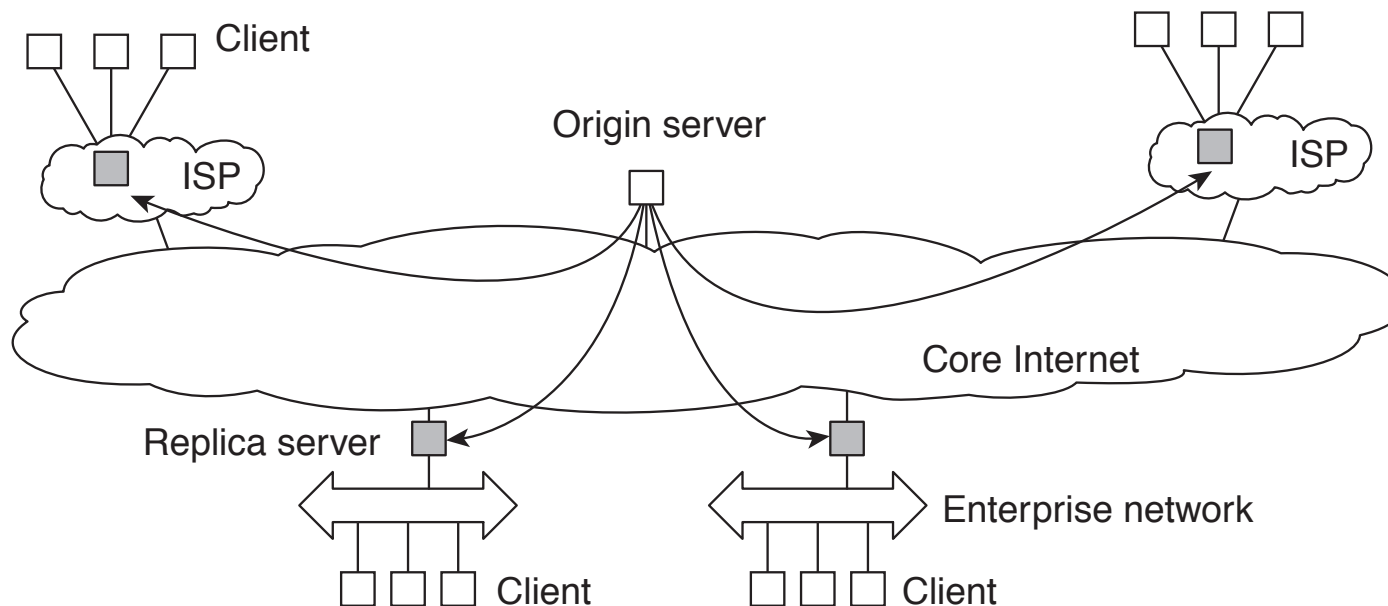
Globule

Collaborative CDN that analyzes traces to decide where replicas of Web content should be placed. Decisions are driven by a general **cost model**:

$$cost = (w_1 \times m_1) + (w_2 \times m_2) + \dots + (w_n \times m_n)$$

goal: minimize cost function by placing dynamically
adjusting the placement of resources

Example: Globule



- Globule origin server collects traces and does **what-if analysis** by checking what would have happened if page P would have been placed at edge server S .
- Many strategies are evaluated, and the best one is chosen.