Distributed Systems
Principles and Paradigms

Chapter 02
(version September 5, 2007)

Maarten van Steen

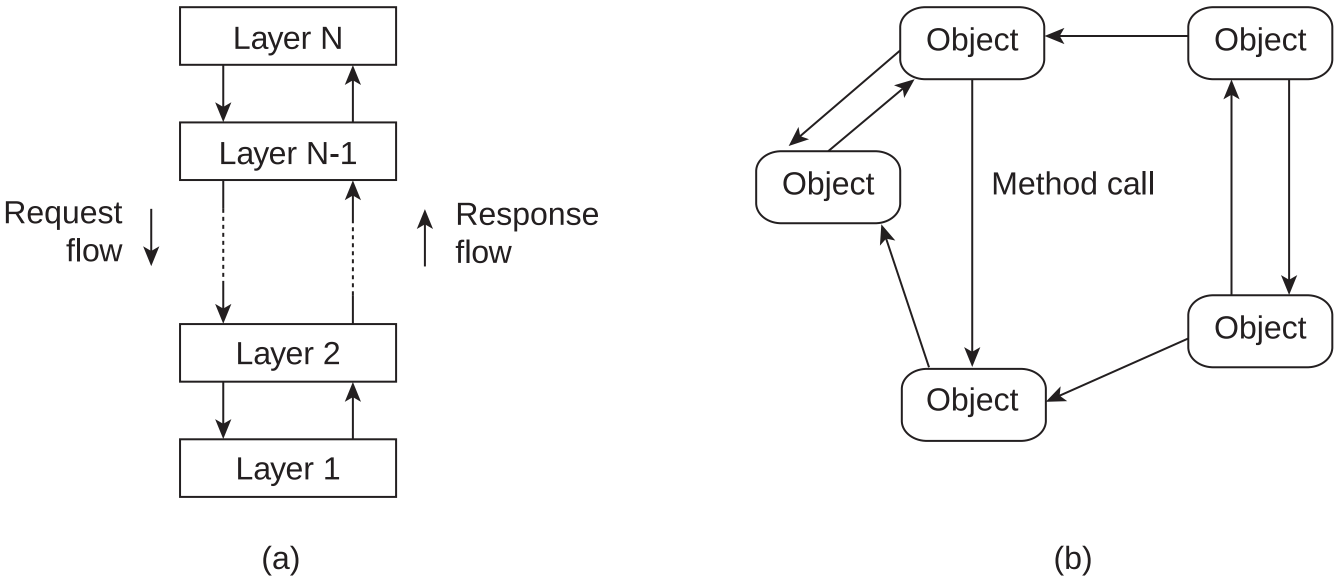
Vrije Universiteit Amsterdam, Faculty of Science
Dept. Mathematics and Computer Science
Room R4.20. Tel: (020) 598 7784
E-mail:steen@cs.vu.nl, URL: www.cs.vu.nl/∼steen/

Architectures

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

Architectural styles (1/2)

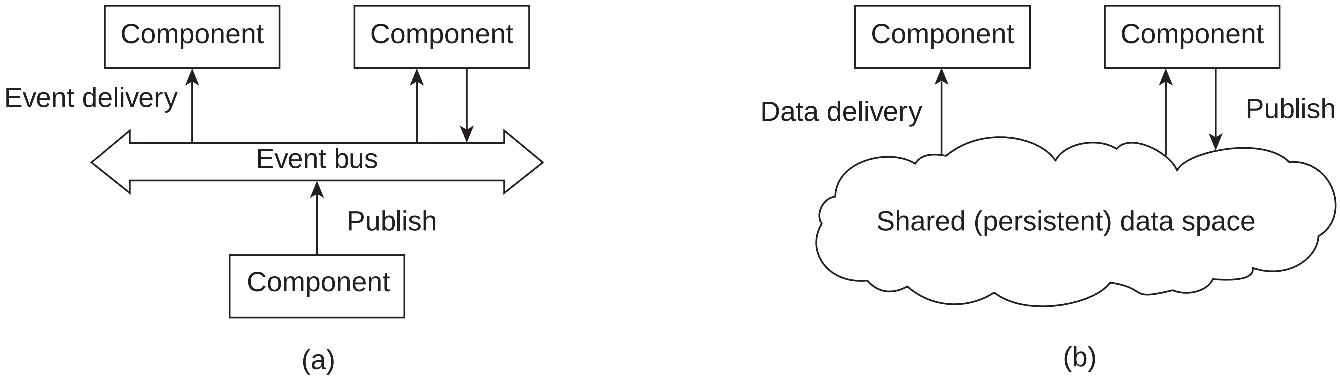
Basic idea: Organize into **logically different** components, and subsequently distribute those components over the various machines.



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

Architectural Styles (2/2)

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

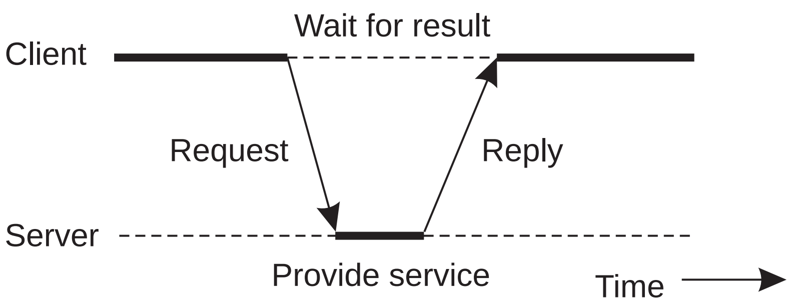


(a) Publish/subscribe and (b) Shared dataspace

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 distributed across different machines
- Clients follow request/reply model with respect to using services



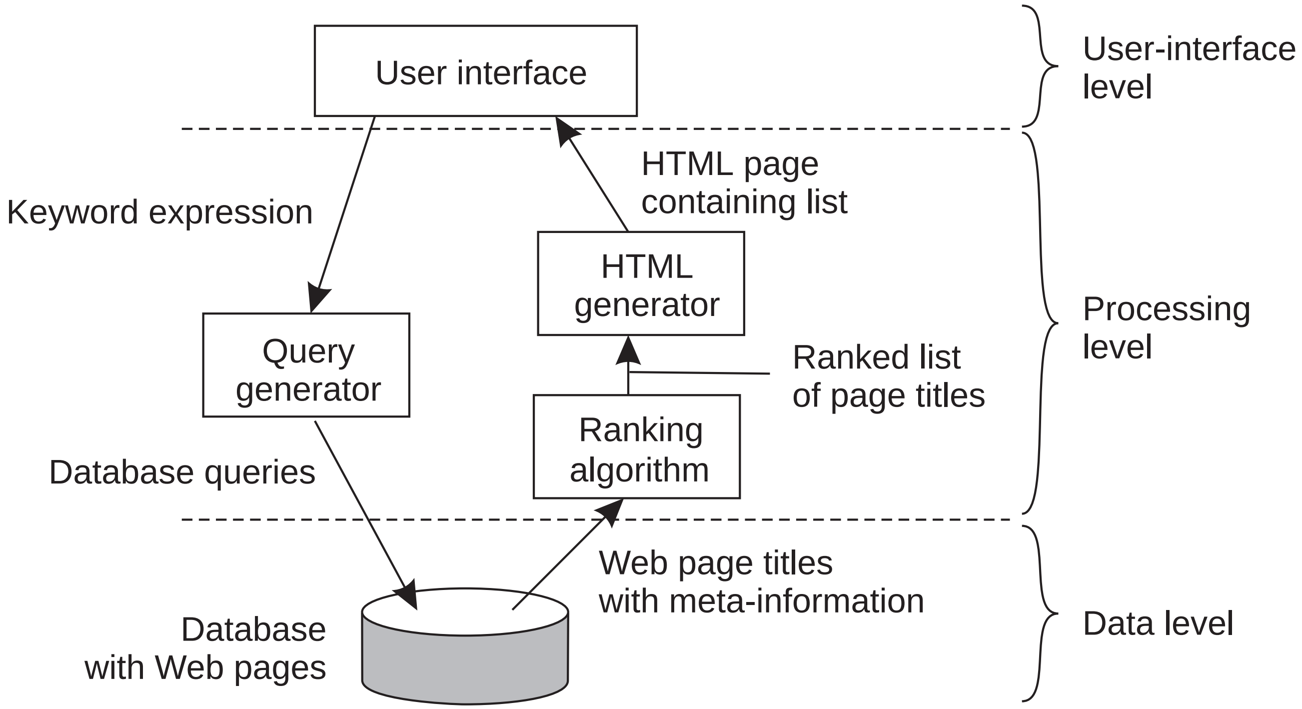
Application Layering (1/2)

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.

Application Layering (2/2)



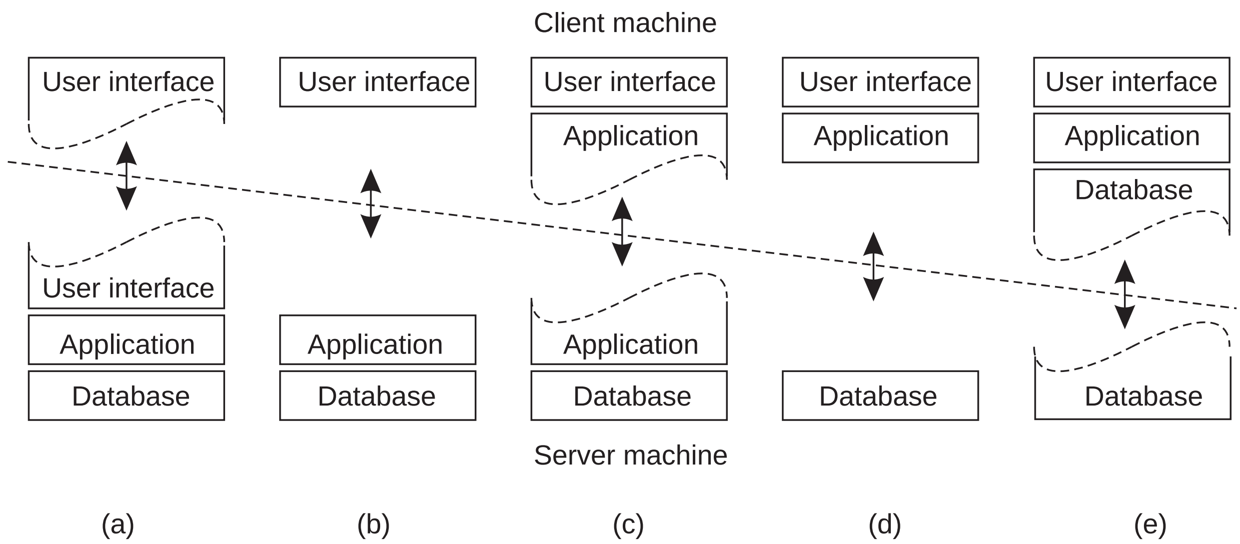
Multi-Tiered Architectures

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

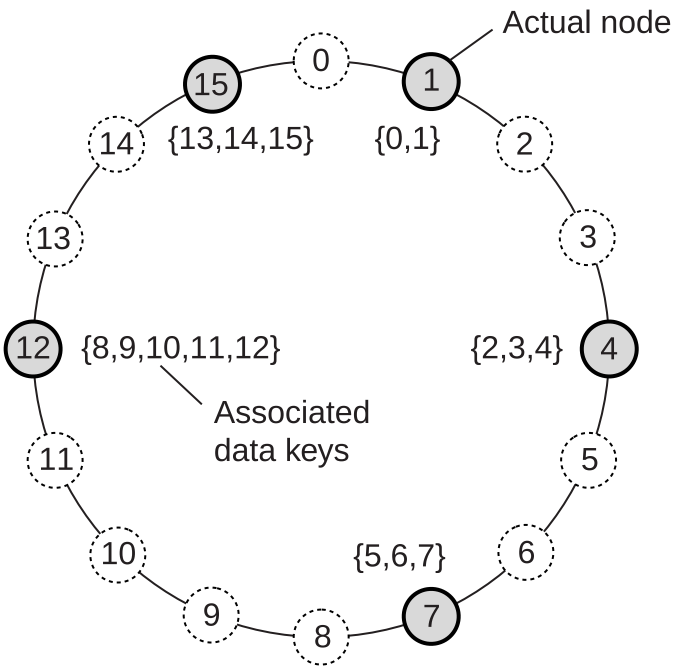
Observation: In the last couple of years we have been seeing a tremendous growth in **peer-to-peer systems**:

- **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 (1/2)

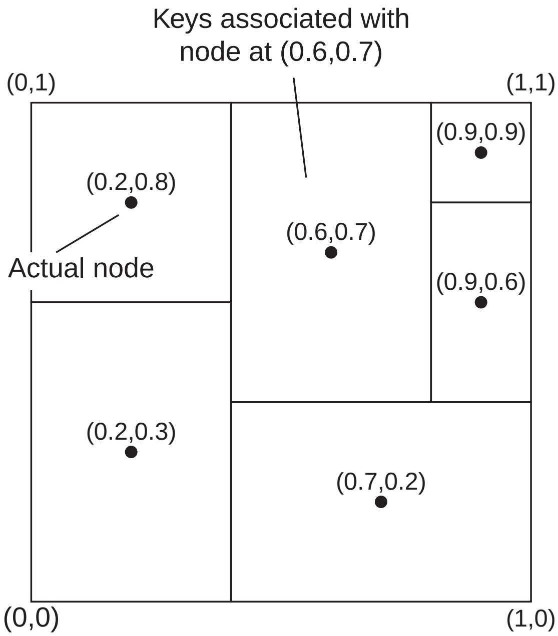
Basic idea: Organize the nodes in a structured **overlay network** such as a logical ring, and make specific nodes responsible for services based only on their ID:



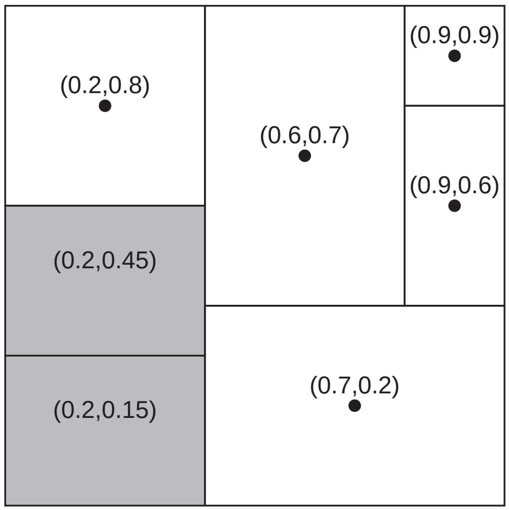
Note: The system provides an operation `LOOKUP(key)` that will efficiently **route** the lookup request to the associated node.

Structured P2P Systems (2/2)

Other example: Organize nodes in a d-dimensional space and let every node take the responsibility for data in a specific region. When a node joins $\Rightarrow$ split a region.



(a)



(b)

Unstructured P2P Systems

Observation: Many unstructured P2P systems attempt to maintain a **random graph**:

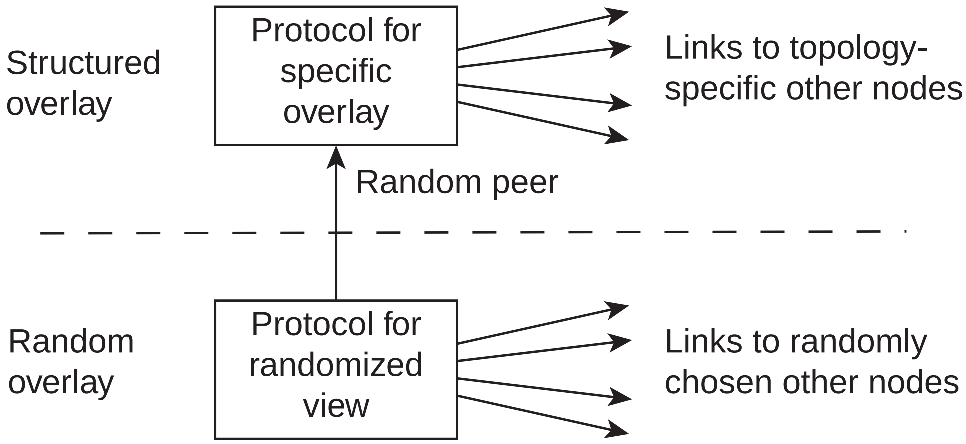
Basic principle: Each node is required to be able to contact a randomly selected other node:

- 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

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

Topology Management of Overlay Networks (1/2)

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.

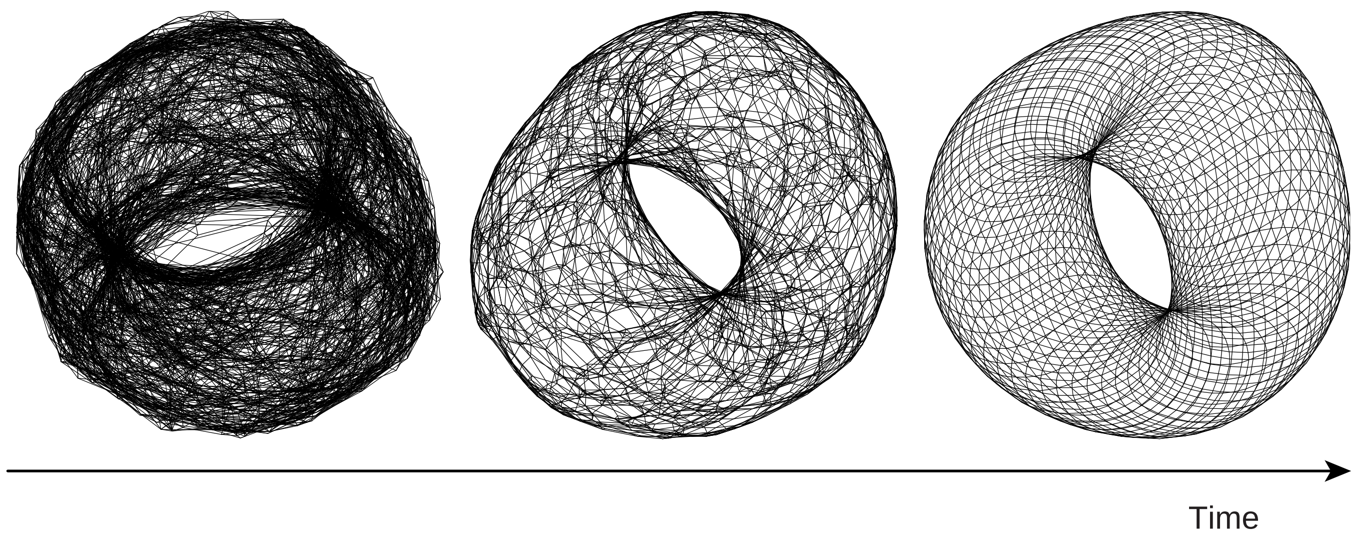
Topology Management of Overlay Networks (2/2)

Example: Consider a $N \times N$ grid. Keep only references to nearest neighbors:

$$\| (a_1, a_2) - (b_1, b_2) \| = d_1 + d_2$$

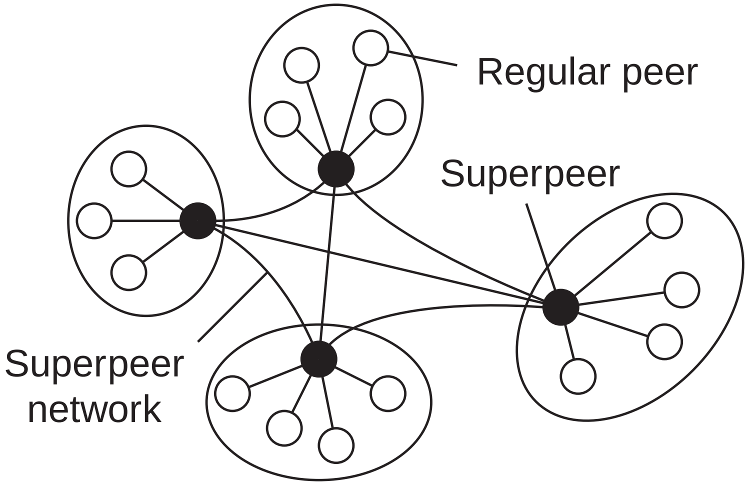
$$d_i = \min\{N - |a_i - b_i|, |a_i - b_i|\}$$

Result: a nice **torus** will appear after a while:



Superpeers

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



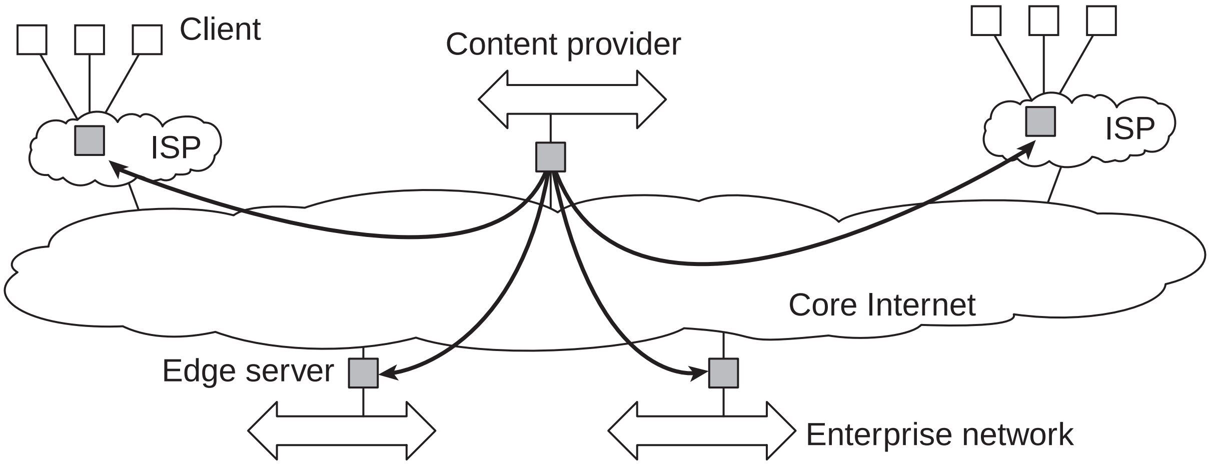
Examples:

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

Hybrid Architectures (1/2)

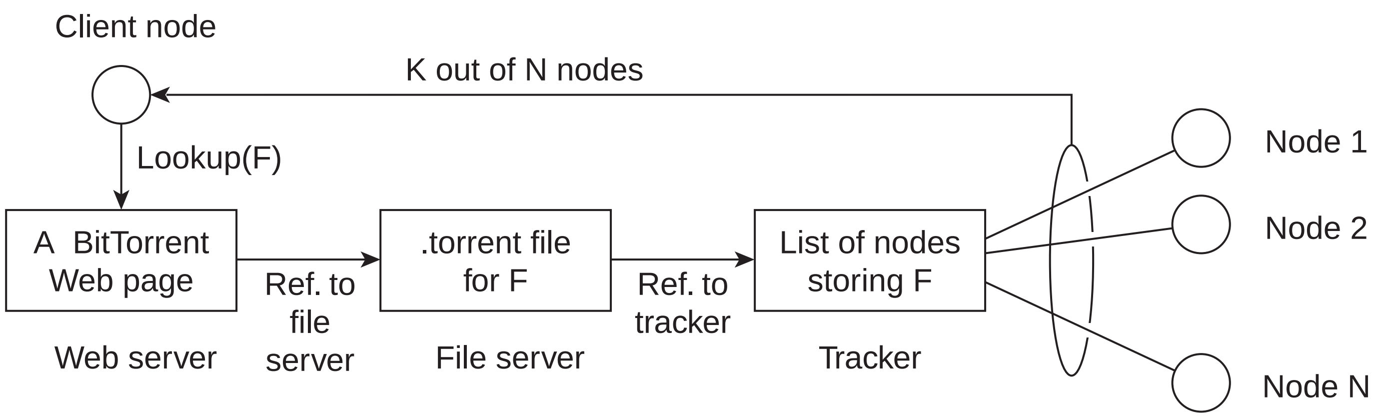
Observation: In many cases, client-server architectures are combined with peer-to-peer solutions

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



Hybrid Architectures (2/2)

Example: Combining a P2P download protocol with a client-server architecture for controlling the downloads: **Bittorrent**

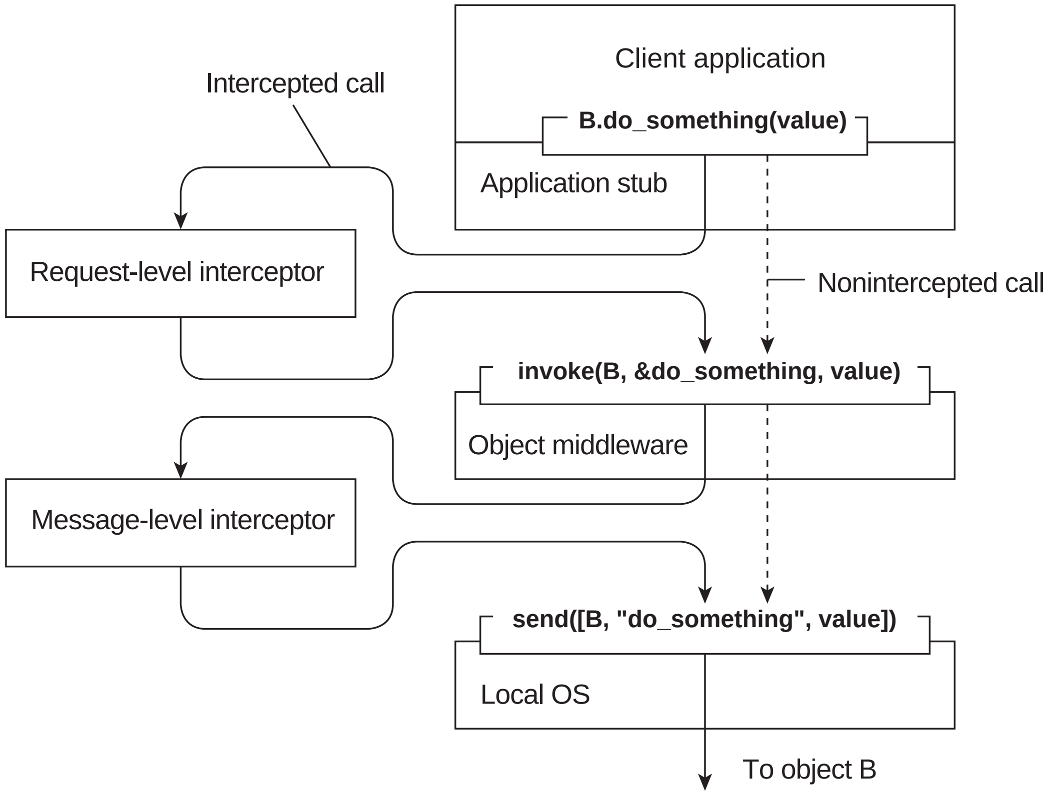


Basic idea: Once a node has identified where to download a file from, it joins a **swarm** of downloaders who in parallel get file chunks from the source, but also distribute these chunks amongst each other.

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 ⇒ there is a need to (dynamically) adapt the behavior of the middleware when needed.

Interceptors: Intercept the usual flow of control when invoking a remote object:



Adaptive Middleware

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

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

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

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

Self-managing Distributed Systems

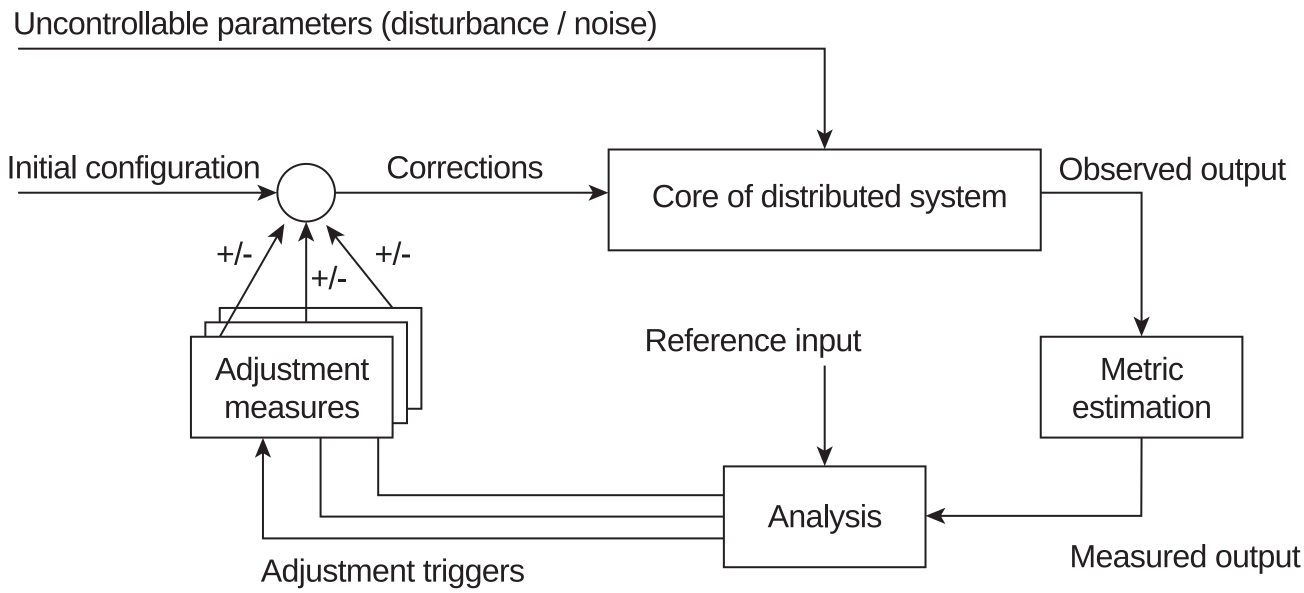
Observation: Distinction between system and software architectures blurs when **automatic adaptivity** needs to be taken into account:

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

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

Feedback Control Model

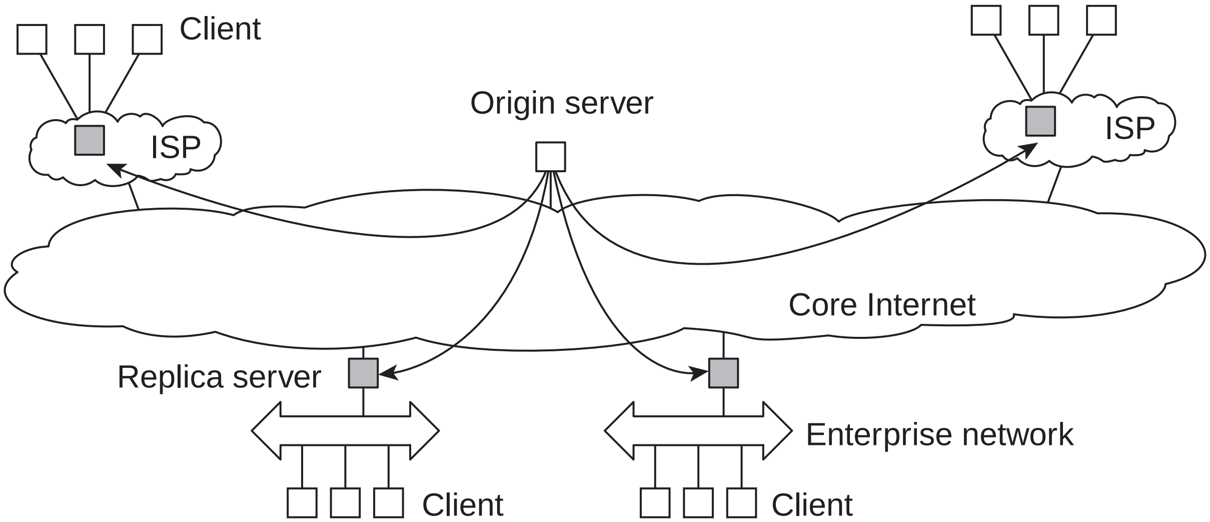
Observation: In many cases, self-* systems are organized as a **feedback control system**:



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**:

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



- 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.