Distributed Systems Architectures

Today

- Software architectures
- Systems architectures
- Architectures & middleware
- Self-* in distributed systems

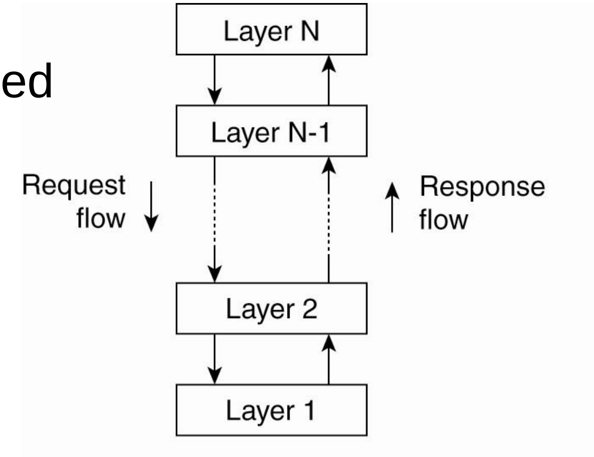
Software and system architectures

- Distributed systems are complex pieces of software – to master complexity: good organization
- Different ways to look at organization of distributed systems – two obvious ones
  - Software architecture – logical organization of software components – how the various software components are organized and how they should interact
  - System architecture – their physical realization – the instantiation of software components on real machines

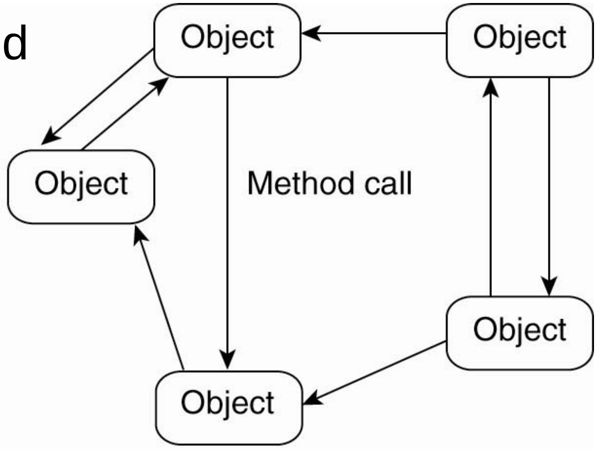
Architectural styles

- Organize into logically different components, and distribute those components over the various machines
  - Component: modular, replaceable unit with well defined I/F
  - Connector: a mechanism that mediates communication, coordination or cooperation among components
- Using components and connectors, different architectural styles

Layered



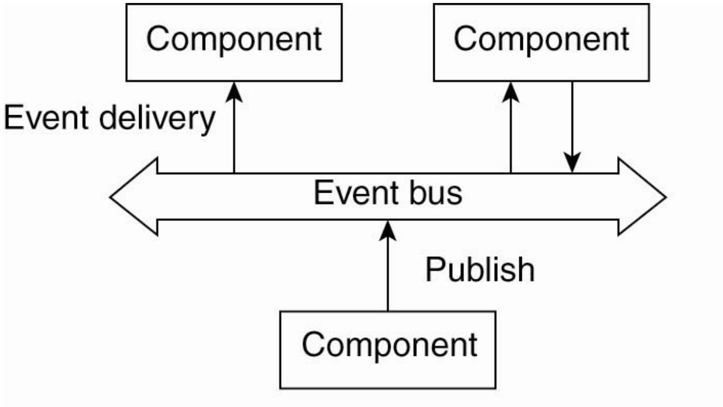
Object-based



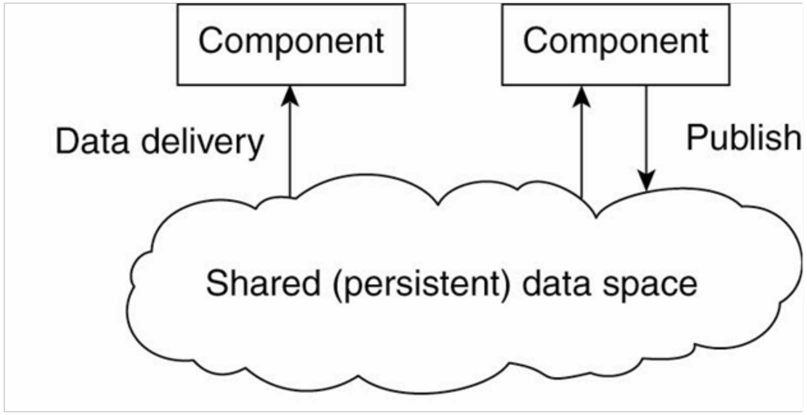
Architecture styles

- Decoupling processes in
  - Space (“anonymous” or referential decoupling) and
  - Time (“asynchronous” or temporal decoupling)
- Alternative styles

Event-based



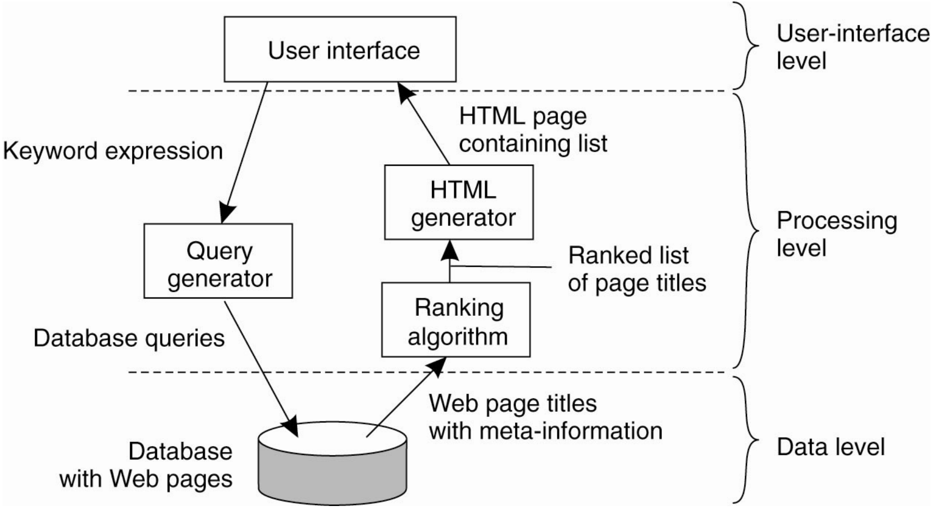
Shared data-space



System arch – vertical distribution

- Basic client/server model
  - Server processes offer services use by clients processes
  - Clients follow request/reply model in using services
  - Clients/servers can be distributed across different machines
- Traditional three-layered view
  - User-interface layer – an application's user interface
  - Processing layer – application, i.e. without specific data
  - Data layer – data to manipulate through the application

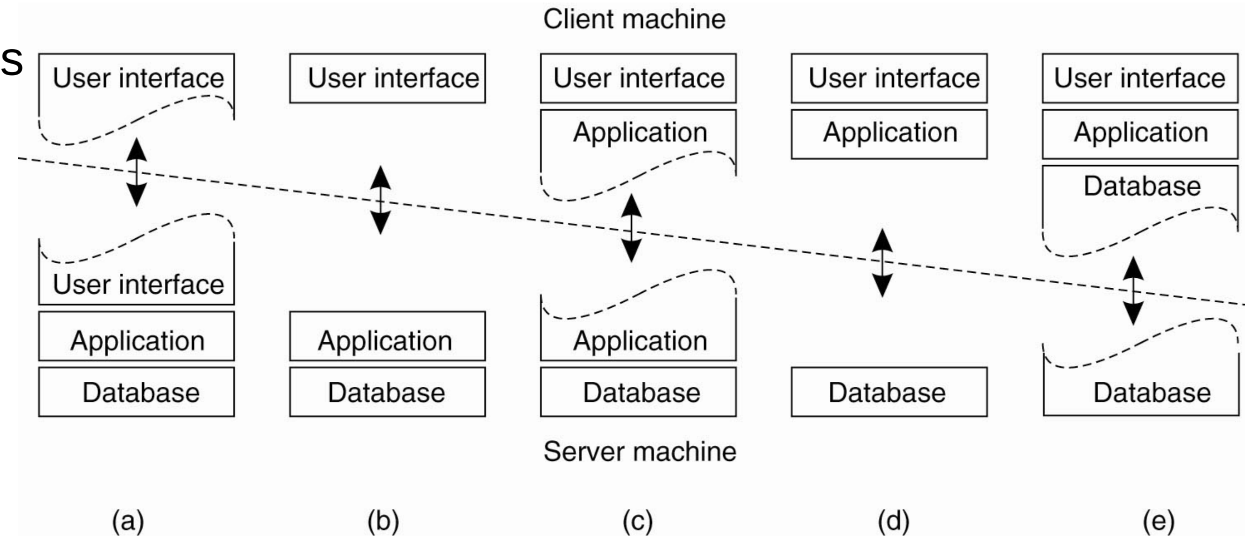
Internet search engine



System arch – vertical distribution

- Logically
  - Single-tiered: dumb terminal/mainframe configuration
  - Two-tiered: client/single server configuration
  - Three-tiered: each layer on separate machine
- Physically
  - Distributing components into client and server machines
  - With a two-tiered architecture

Thin clients



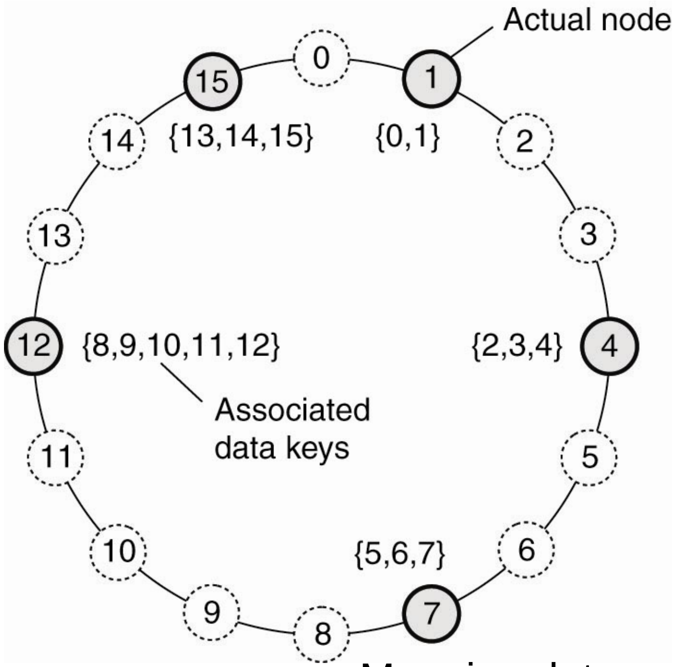
Fat clients

System arch – horizontal distribution

- In the last couple of years we have been seeing an impressive growth in P2P systems
  - Structured, DHT-based, 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
- In all cases, we are dealing with overlay networks: data is routed over connections setup between the nodes

Structured P2P systems

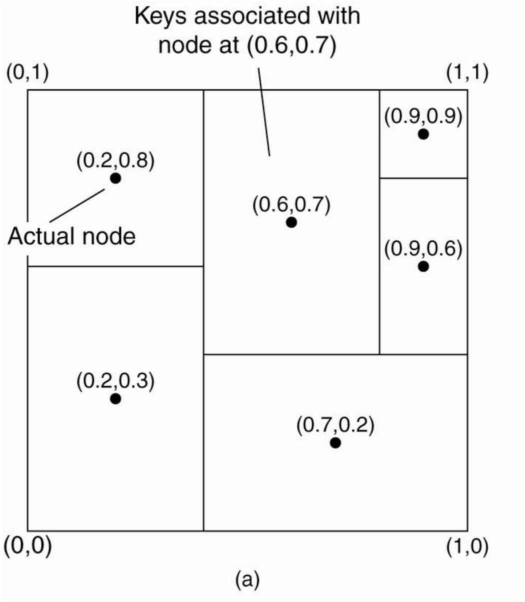
- Organize the nodes in a structured overlay network such as a logical ring, and make specific nodes responsible for services based only on their ID
- The system provides an operation LOOKUP(key) to route the lookup request to the associated node
- Node join is straightforward
  - Generate a random id
  - Do a lookup on id, getting the succ(id)
  - Contact succ(id), and its predecessor, to insert itself in the ring
  - Transfer data items from succ(id) to new node

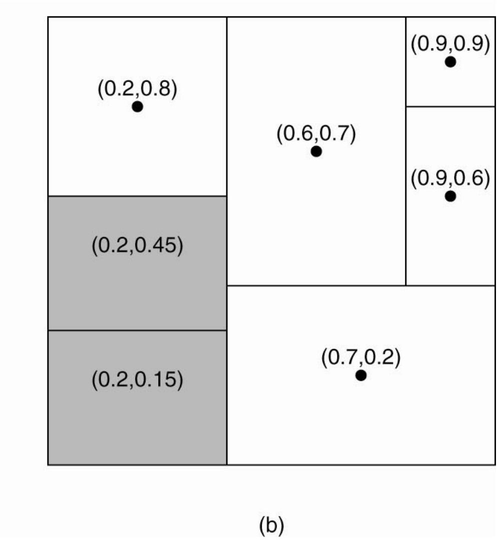


Mapping data items onto Chord nodes

Structured P2P systems

- CAN – Content Addressable Network
- Organize nodes in a *d-dimensional* space and let every node take the responsibility for data in a specific region
- When a node joins ⇒ split a region
- Leaving it's a bit more complicated



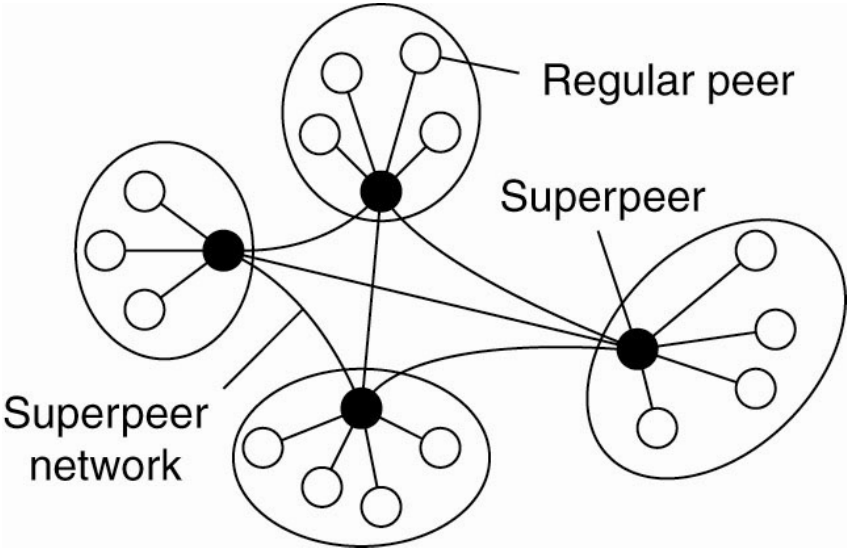


Unstructured P2P systems

- Many unstructured P2P systems attempt to maintain a random graph:
- Basic idea – each node contacts 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
- An exclusive pull/push model can easily conduct to disconnected overlays
- In general, much easier to leave/join the network

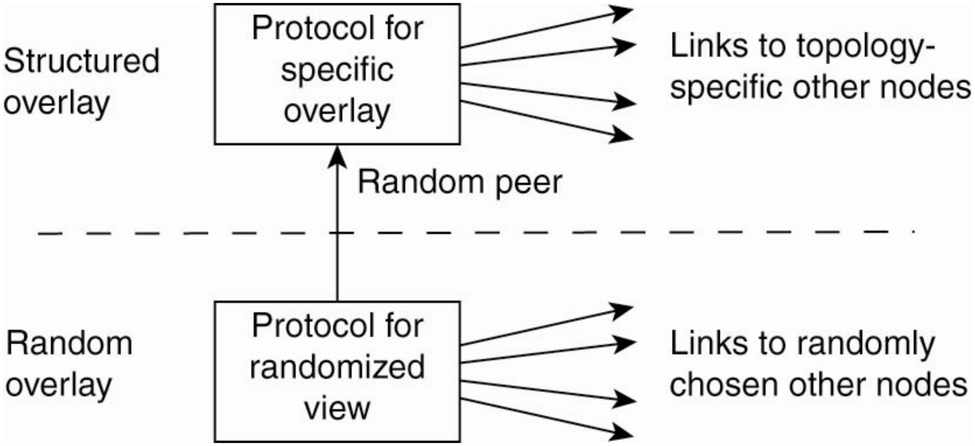
Super-peers in unstructured P2P systems

- Sometimes it may help break with the symmetric nature of P2P – super/ultra-peers
- Some obvious examples
  - Transiency – pick the most stable ones
  - Search – have them keep the indexes for scalable searches
  - Organization – have them monitor the state of the network



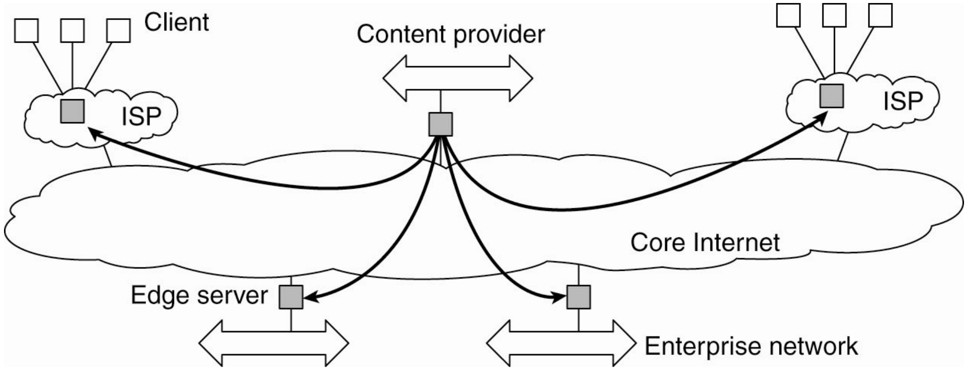
Combining structured and unstructured

- Distinguish two layers: (1) maintain random partial views in lowest layer; (2) be selective on who you keep in higher-layer partial view
- Lower layer feeds upper layer with random nodes; upper layer is selective when it comes to keeping references
  - Instead of simple random, ranking peers based on some simple function (latency, semantic) may help


Structured overlay
Protocol for specific overlay
Links to topology-specific other nodes
Random peer
Random overlay
Protocol for randomized view
Links to randomly chosen other nodes

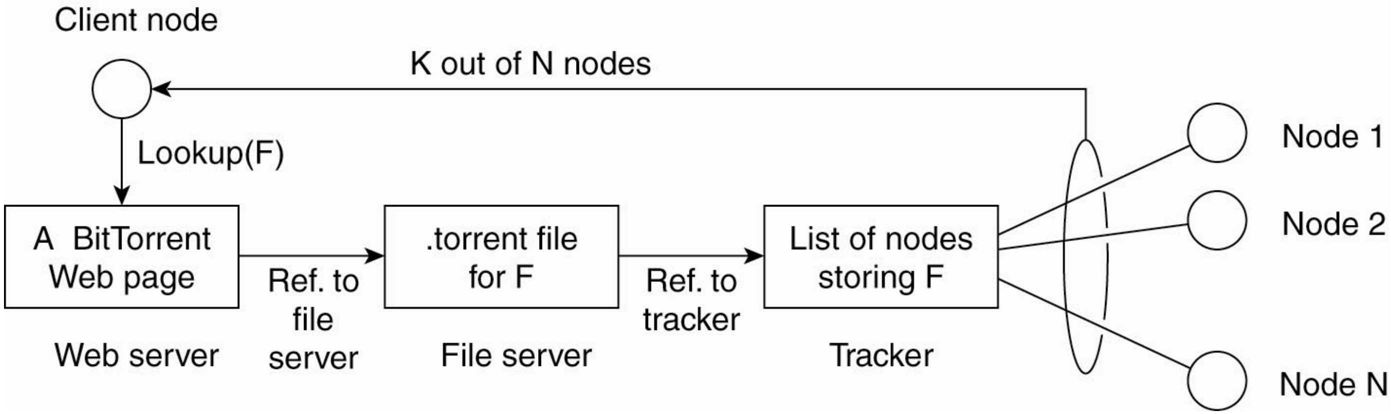
Hybrid architectures

- Client-server architectures and P2P solutions
- E.g. Edge-server architectures often used for Content Delivery Networks
- Edge-servers are placed at the edge of the network
- Responsible for caching, filtering, transcoding …
- Clients connect through the edge-server



Hybrid architectures

- E.g. BitTorrent – client-server to connect to the swarm and P2P from then on
- Files are splits into chunks, peers swap chunks within a swarm
- Get a torrent from a web site
- Contact the tracker listed in the torrent
- Get a set of peers from the tracker and connect to the swarm



Architecture and middleware

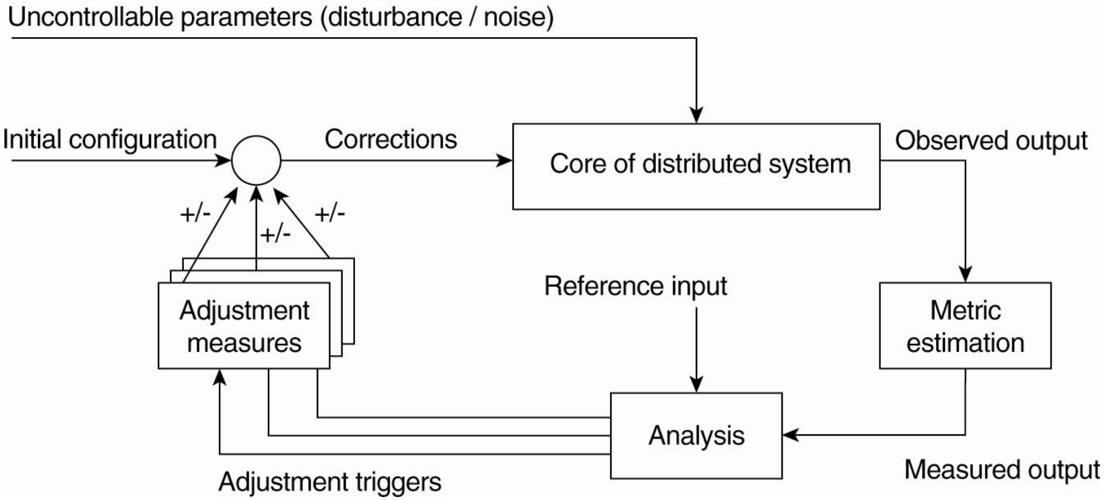
- A key goal for middleware is to provide distribution transparency
- Typically, however, middleware adopts particular architecture styles
  - Makes it simpler to develop applications for that style
  - *Makes it hard/inefficient to do it with any other!*
- To alternatives – build different versions or make them easy to adapt dynamically
- Interceptors: Intercept the usual flow of control when invoking a remote object
  - Make replication transparent
  - Make handling MTU transparent
  - …

Adaptive middleware

- To deal with changing environments/demands – adaptive middleware
- To facilitate software adaptation
  - Separation of concerns: Separate general functionalities and later weave them together into an implementation
  - Computational reflection: Let program inspect itself at runtime and adapt/change its settings dynamically if necessary
  - Component-based design: Organize a distributed application through components that can be dynamically replaced when needed
- Nothing that simple – component interdependencies?
- We do need adaptive systems, but is this a software or a system issue? i.e. adaptive software or adaptive systems?

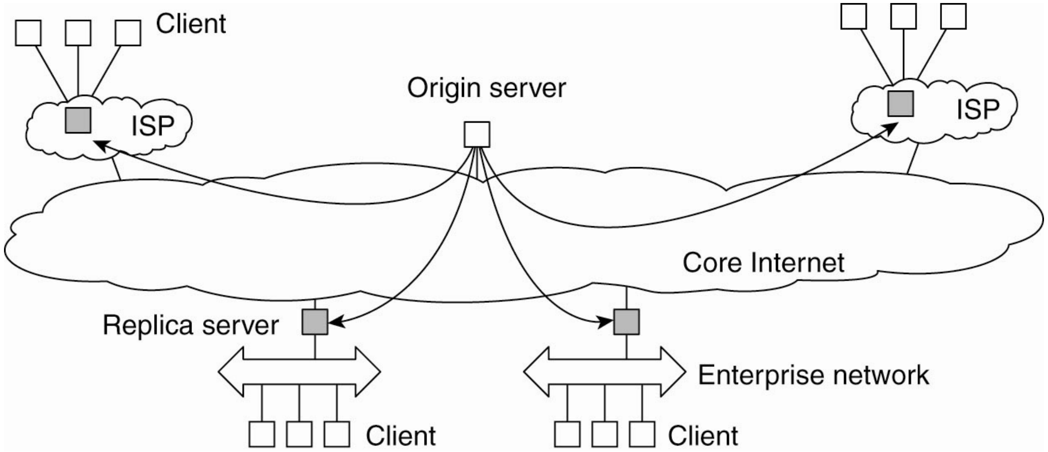
Self-management in distributed systems

- Systems should be adaptable not in terms of their software components, but rather execution behavior
- Self-*/Autonomics systems – self-configurable, Self-manageable, Self-healing, Self-optimizing
  - Commonly, organized as a feedback control system
    - System needs to be monitored
    - Collected measurements must be analyzed to decide on adaptation
    - Different mechanisms must be used to enact changes
    - (Not unlike manual management)



Self-management in Globus

- Collaborative CDN – it analyzes traces to decide where replicas of Web content should be placed. Decisions are driven by a general cost model
- Globule origin server
  - Collects traces
  - Does *whatif* 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.



Summary

- Organization to master complexity, both on how the components are interconnected and instantiated
- There’s a strong connection between software/system architectures and (self-) adaptation
- Should adaptation to environmental changes be seen as a software or a system issue?