Architectures
Lecture 2

Lecturer: Mike Mchunu

School of Computer Science
Wits University

August 8, 2013

Outline

Introduction

- Distributed systems are complex, with components dispersed across multiple machines.
- Can master complexity in such systems through proper organization.
- Different ways of how organization of such systems can be viewed.
  - Distinction between *logical* organization of software components and the actual *physical* realization of these systems.
- Distributed system organization is *mostly* about *software components* that constitute the system.
  - These *software architectures* tell us about the *organization* and *interaction* of the different software components.

- The actual realization of a distributed system is about placing software components on real (physical) machines.
- This can be done in several ways. An architecture may be realized in a
  - *centralized* manner, with most components being placed on a single machine, or it
  - *decentralized* manner, with most machines having more or less the same functionality, or it
  - *hybrid* manner, which combines the above two realizations.
- The final instantiation of a software architecture is referred to as a *system architecture*.

Architectural Styles

- Begin by considering logical organization of distributed systems into software components (aka *software architecture*).
- Software architecture research has matured.
  - Now commonly accepted that developing large systems requires *designing* or *adopting* an architecture.
- The concept of an *architectural style* is important. It is formulated in terms of components. That is,
  - the way they are connected to each other
  - the data exchanged between them, and
  - how they are jointly configured into a system.

- What is a *component*?
  - A replaceable modular unit with well-defined required and provided interfaces.
- What is a *connector*?
  - A mechanism that mediates communication, coordination, or cooperation among components.
- A combination of components and connectors produces various configurations.
  - These are classified into *architectural styles*.

- There are several architectural styles.
- For distributed systems, the most important styles are:
  1. Layered architectures
  2. Object-based architectures
  3. Data-centered architectures
  4. Event-based architectures

Layered Architectures

- The components organized in layered fashion.
- The component at layer L_i can only invoke those at layer L_{i-1}, but not the other way round.
- Model is highly adopted by the networking community.
- Control flows from layer to layer.
    - Requests go down the hierarchy and the results flow upwards.

- The following is a picture of a layered architecture:

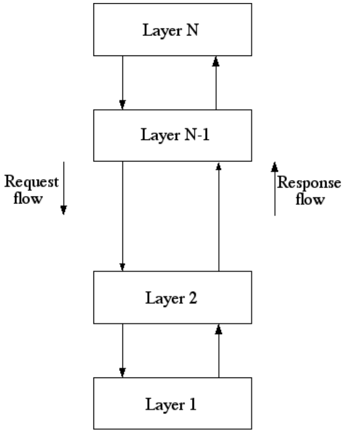

Figure: Layered Architectural Style

Object-based Architectures

- Each object corresponds to a component.
- Components connected through RPC mechanism.
- Along with layered architectures, they form the most important style for large software systems.
- Object-based architecture shown below:

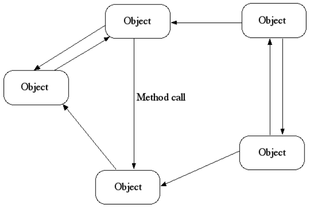

Figure: Object-based Architectural Style

Data-centered Architectures

- They are based on the idea that processes communicate through a common (passive or active) repository.
- A data-centered architecture is pictured as follows [1]:

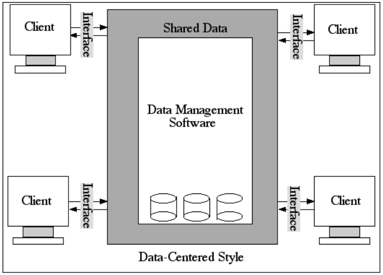

Figure: Data-Centered Architectural Style

- Their principal goal is the integrability of data [1, 2].
- These architectures are mainly characterized by *access to shared data* [2].
- The centralized data repository communicates with several clients
  - The clients are independent entities, which can be modified without affecting other clients [2].
  - It is also easy to add new clients.
- Three protocols are used: communication, data definition and data manipulation.

Event-based architectures

- In [3], an event-based architecture (EBA) is defined as follows:

  An EBA is an architecture based on parts that interact solely or predominantly using event notifications, instead of direct method calls.

  - *Event notification* refers to a signal containing information regarding an event detected by the sender.
- There are a number of communication styles: publish/subscribe, broadcast, point-to-point.
- Processes communicate by propagating events.

- In distributed systems, event propagation is associated with *publish/subscribe* systems.
    - Events are published.
    - The only processes that are notified of these events are the ones that have subscribed to receive them.
        - For example, a process subscribing to a particular service and in turn receiving email updates.
- Sender and receiver are decoupled.
    - They communicate asynchronously.

- An event-based architecture is shown below:

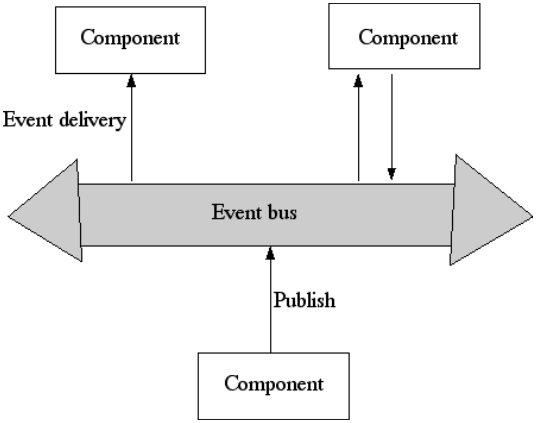

Figure: Event-Based Architectural Style

- Focusing on publish/subscribe systems, we discuss their advantages and disadvantages.
- Advantages
    - Loosely coupled [4, 5]
        - Publishers and subscribers loosely coupled, each being able to operate on its own. regardless of the other.
        - The term *referentially decoupled* is used to denote the fact that these loosely coupled processes do not have to explicitly refer to each other.
    - Scalable
        - Systems are scalable when their size is relatively small.
        - However, this benefit is lost when systems scale up, with many servers being supported.
- Disadvantages [5]
    - Decoupling publisher from subscriber may lead to incorrect or damaging messages.

Shared Data-space Architectural Style

- Combining event-based architectures with data-centered architectures produces a shared data-space architecture, shown below:

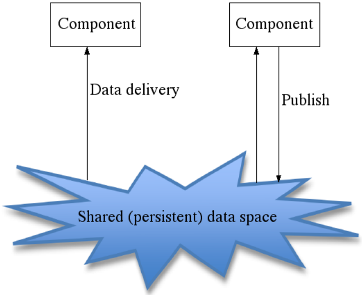

Figure: Shared Data-Space Architectural Style

- Such architecture may be used in several ways. For example
  - Shared distributed file systems.
  - Web-based distributed systems.
- Asynchronous communication between processes
- Advantage?
  - Process decoupled in time.
    - When communication occurs, not necessary for *both* processes to be active.
  - "Descriptive" reference
    - Instead of explicit reference, a descriptive SQL-like interface is used to access data in the repository.

Importance Of Software Architectures

- They all seek to achieve distribution transparency.
  - However, distribution transparency requires trade-offs between performance, fault tolerance, ease-of-programming, etc.
- Different distributed applications must be solved using different applications/architectures.
  - There is no way a single distributed system can cover 90% of all possible cases.

System Architectures

- What is a system architecture?
  - An instance of a distributed system that results after deciding on software components, their interaction, and their *placement* (on physical machines).
- Depending on placement, we get the following distributed system architectures:
  - Centralized architectures.
  - Decentralized architectures.
  - Hybrid architectures.

Centralized Architectures

- Based on *client-server* model.
- Processes divided into two groups.
  - A *server* process that implements a specific service.
  - A client process that requests a service from a server.
- Client-server interaction (*request-reply*) shown below:

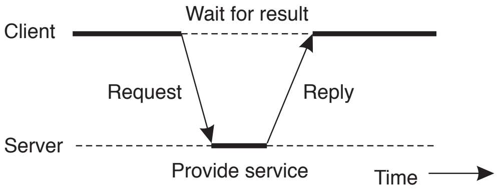

Figure: General interaction between a client and a server

- When network is fairly reliable, as is the case with many LANs, client and server can use a simple *connectionless* protocol to communicate.
- Examples of this protocol include the Internet Protocol (IP) and the User Datagram Protocol (UDP).
- A few issues to consider (**read page 37 textbook**):
  - How does the idea of a connectionless protocol "work"?
  - What is its advantage?
  - What is the disadvantage of using such a protocol?
  - What is an idempotent operation?

- Can also get client and server to communicate using reliable *connection-oriented* protocol.
    - Not wholly appropriate in LANs, due to low performance.
    - OK in WANs, where communication is inherently unreliable.
    - How does this protocol "work"? (Revision: read about it).

Application Layering (Centralized Architectures)

- Many client-server applications are geared to support user access to databases.
- So, many people advocate distinction between following three levels, as in layered architectural style:
  1. The user-interface level.
  2. The processing level.
  3. The data level.

- Internet search engine's organization into three layers:

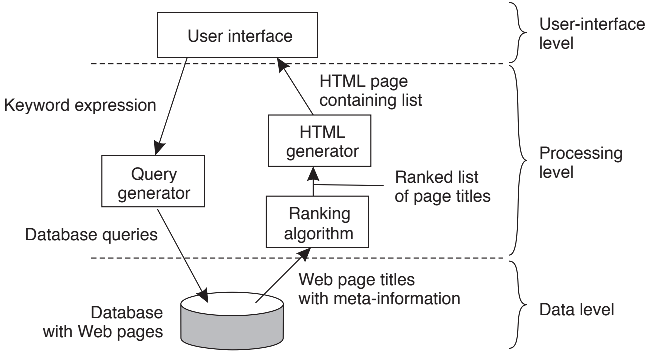

Figure: Simplified Internet search engine organization

- *User-interface level*
    - contains facilities for direct interface with users, such as GUIs.
- *Processing level*
    - Contains data processing applications - the core functionality.
- *Data level*
    - Manages data being acted on.
    - Interacts with database or file system.
    - Data usually *persistent*. Exists even when not being accessed by client.

- **Read** the following examples (**pages 39-40** of textbook):
  - **Example 1:** Internet search engine.
    - *Interface level*: Accepts a keyword string.
    - *Processing level*: Processes for generating database queries, rank replies and format response.
    - *Data level*: Database containing web pages.
  - **Example 2:** Decision support system.
    - *Interface level*: Accepts somewhat complex input than simple search.
    - *Processing level*: Methods and techniques from AI and statistics for financial data analysis.
    - *Data level*: Database containing financial information.
  - **Example 3:** Desktop package.
    - *Interface level*: Access to documents and data.
    - *Processing level*: Word processing, database queries, spreadsheets, etc.
    - *Data level*: file systems and/or databases.

Multitiered Architectures (Centralized Architectures)

- Two-tier and three-tier architectures are the most common.
- **Two-tier Architectures**
  - Distinction into three logical levels makes it possible to physically distribute a client-server application across several machines.
  - Simplest organization is to have only two types of machines:
    1. A *client* machine with programs that implement (part of) user-interface level.
    2. A *server* machine containing programs that implement processing and data levels.
  - In this organization, server handles everything and client is a dumb terminal, possibly with a pretty graphical interface.

- Clients and servers can be organized by distributing programs in application layer across different machines.

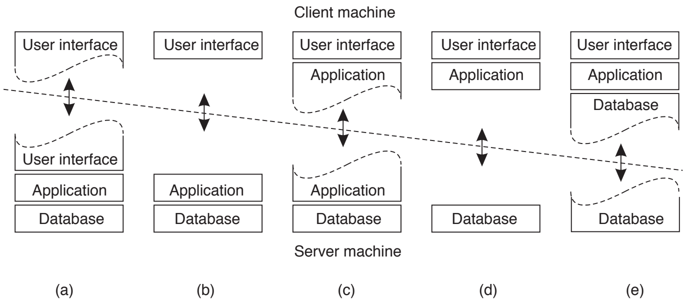

Figure: Alternative client-server organizations (a)-(e)

- There are two kinds of machine: client and server. This leads to a **(physically) two-tiered architecture**.

- A note about client-server organizations
  - In one extreme, server performs processing and manages data. The client only provides a simple graphical interface (*thin client*).
  - At the other extreme, client does all application processing, and also stores some data (*fat client*).
- **Study** and **understand** the descriptions on possible client-server organizations that can be realized (**pages 41-42** of textbook).

- **Three-tiered Architecture**
  - So far, distinction only made between client and server machines.
  - However, a server may need to act as a client, leading to a **(physically) three-tiered** architecture, shown below:

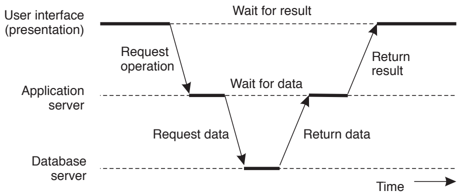

Figure: An example of a server acting as a client

  - Here, programs forming part of processing level reside on a separate server, but may additionally be partly distributed across client and server machines.

- A separate processor may be used for each application architecture layer.
- Advantages of the architecture [6]
  - Better performance compared to thin-client approach.
  - Simpler to manage, compared to fat-client approach.
  - More scalable architecture. With increasing demand, possible to add more servers.

- Useful picture (3-tier Internet banking system [6])

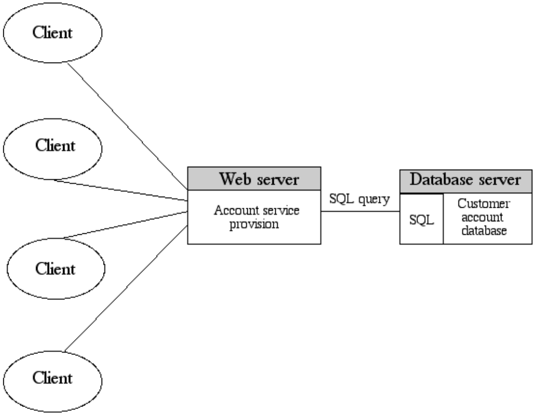

Figure: An Internet banking system

Decentralized Architectures

- We focus on two types of distribution: vertical and horizontal distribution.
- **Vertical Distribution**
    - Achieved by placing logically different components on different machines.
    - Exhibited by traditional client-server architectures.
        - Where each level serves a different purpose in the system.
    - It *distributes the traditional server functionality over multiple servers* [7].
    - Advantage of having a vertical distribution?
        - Functions are logically and physically split across multiple machines.

- Useful picture - vertical distribution (adapted from [7])

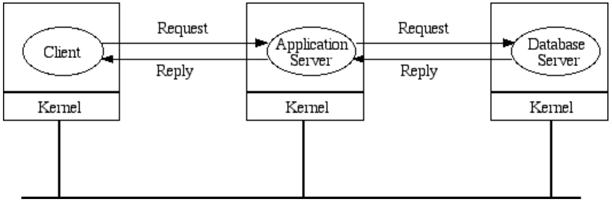

Figure: The vertical distribution communication architecture

- The first server receives the client's request.
- The request is passed on to the next server, and so on, until the last server is reached.
- To fulfill the client's request, each server does its bit in its own tier (step).

- Benefits? Scalability and flexibility [7]
  - Scalability improved since, with each server having less processing load, the system is able to take up more users.
  - The internal functionality of each server can be modified, which makes the architecture flexible.

- **Horizontal Distribution**
  - More common in modern architectures.
  - Involves distribution of clients and servers.
  - A client or server is physically split up into logically equivalent parts.
    - Each part operates on its own share of complete data set, thus balancing the load.

- Useful picture [7]

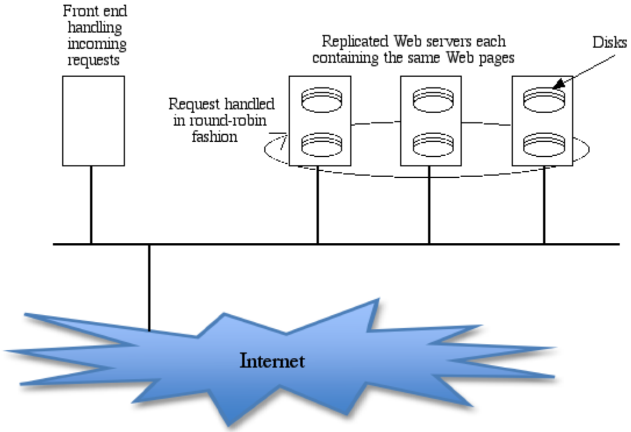
Front end handling incoming requests
Replicated Web servers each containing the same Web pages
Disks
Request handled in round-robin fashion
Internet

Figure: Horizontal distribution - Web services example

- Each server computer has its own copy of all hosted Web pages.
- The servers handle client requests in *round robin* fashion.

- Benefits? Scalability and reliability.
  - Scalability is achieved by using a number of servers, thereby reducing the load on each server.
  - Since each server contains the same web pages, such redundancy provides reliability.

Peer-to-peer (P2P) Systems (Decentralized Architectures)

- A class of modern system architectures that support horizontal distribution.
- Resources are distributed and shared among *peers* (user processes).
- Tasks that must be performed are performed by every process that forms part of the distributed system.
- Characteristics of P2P systems.
  - All processes in a P2P system are equal.
  - The processes interact *symmetrically*.
    - Each process acts as client and server at the same time (i.e., acting as a *servent*).

- Useful picture (P2P network - adapted from [7])

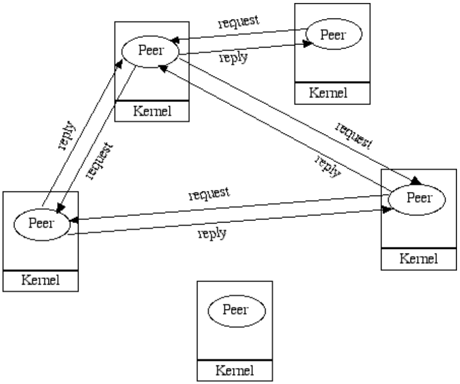

Figure: Peer-to-peer Communication Architecture

- In this figure, each process acts as client and server

- A large number of nodes may participate in a P2P network.
    - This makes it difficult for nodes to keep track of each other, and the information each one has.
- To minimize this problem, the nodes are arranged in an *overlay network*.
    - This is a virtual network the nodes form among themselves.
    - The network is built on top of a physical network.
    - A connection between two overlay nodes may consist of several physical connections
    - A node wanting to send a message to another node locates the node by sending a request along the overlay network links.
        - Once the target node has been located, the two nodes are then able to communicate

- Useful picture: Overlay network [7]

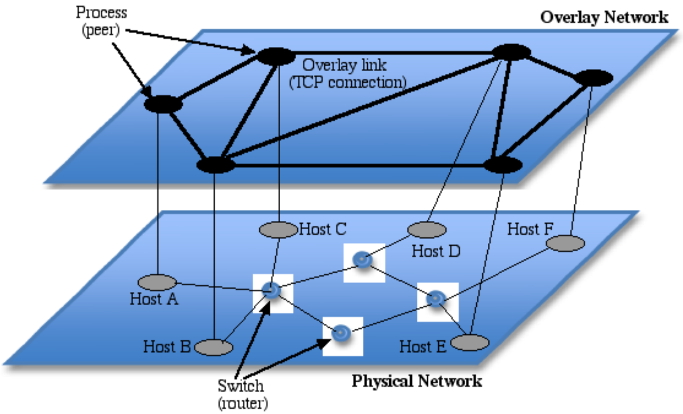

Figure: Overlay Network Example

- There are two types of overlay networks: *structured* and *unstructured*.
- In both, a node in the network maintains a list of its neighbours (referred to as its *partial view*)
- **Structured Peer-to-Peer Architectures**
  - The nodes are organized using a *distributed hash table* (DHT).
  - Generally, a hash function converts a key to a hash value that is used to index a hash table.
  - In a DHT-based system, data items are assigned a random key from a large identifier space, such as a 128-bit or 160-bit identifier.
  - Nodes also assigned random numbers from the same identifier space.
  - The key is to implement an efficient scheme that uses some distance metric to uniquely map a data item's key to a node's identifier.

- Looking up a data item returns the network address of the node that stores that data item.
- DHT characteristics
  - *Scalable*: To thousands and millions of network nodes. Slow increase of search time with size (i.e., `O(log(N))`).
  - *Fault tolerant*: In the event of node failure, system is able to re-organize itself.
  - *Decentralized*: No central coordinator.
- **Example:** Chord - A Structured P2P System
  - Each data item gets assigned a random key from a large identifier space.
  - Each node gets assigned a random identifier from the same identifier space.
  - Nodes logically organized in a ring such that data item with key `k` is mapped to a node with smallest identifier `id`$\geq$`k`.

- This node is referred to as the *successor* of key `k` and denoted as `succ(k)`, as shown below.

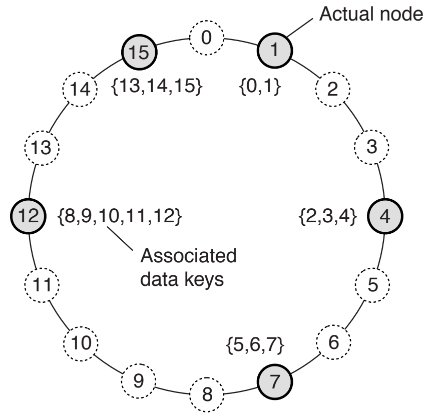

Figure: The mapping of data items onto nodes in Chord

- To look up an item, an application running on arbitrary node calls function `LOOKUP(k)`, which returns the network address of `succ(k)`.
- The application contacts the node (`succ(k)`), from which it obtains a copy of the required data item.

- **Membership management** in Chord
  - A node joining the system generates a random identifier `id`.
  - It then performs a lookup on `id` (`LOOKUP(id)`), which returns the network address of `succ(id)`.
  - At that point, the joining node can contact `succ(id)` and its predecessor and insert itself in the ring.
  - A node id leaves ring by informing its successor and predecessor of its departure and transferring its data items to `succ(id)`.
  - Similar approaches used in other DHT-based systems (**Read** about CAN networks **pp. 45-46** of textbook).

- **Unstructured Peer-to-Peer Architectures**
  - The overlay network is constructed as a random graph, using randomized algorithms.
  - Main idea
    - Each node keeps a randomly constructed list of neighbours.
    - Nodes also contain randomly placed data items
    - A node looking for a specific data item floods network with a search query.
  - Basic model
    - Each node has a list of c neighbours, each representing a randomly chosen live node from current set of nodes.
    - The list of neighbours constitute a *partial view*.

- **Membership management**
  - A node joins the network by contacting any node from a list of well-known access points.
  - A node simply leaves the network, without notifying any other node.
- This architecture is OK for small-to-medium sized networks. However, it does not scale well.
- It is used in systems such as Gnutella and Freenet.
- **Note:** Do **NOT** read the part on **Topology Management of Overlay Networks** (pages **49-50** of textbook).
- **Note:** You **MUST** read the part on **Superpeers** (pages **50-52** of textbook)

- **Superpeers**
  - The growth of the network in unstructured peer-to-peer systems makes it difficult to locate data.
  - It is difficult to find a way of routing a lookup request to a specific data item. So, one option a node has is to flood the system with its request.
  - The alternative is to use special nodes, known as *superpeers*, that maintain an index of data items.

- Superpeers often organized in peer-to-peer network, leading to a hierarchical organization:

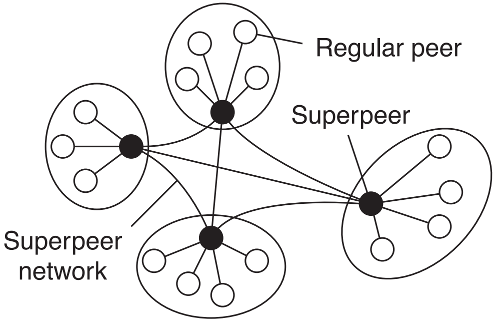

Figure: A hierarchical organization of nodes into a superpeer network

- Every regular peer is a client of a superpeer, which also controls all communication to and from the regular peer.

- The client-superpeer relation is mostly fixed. A regular peer joining the network attaches itself to one of the superpeers. It remains there until it leaves the network.
- Superpeer networks introduce a new problem. That is, how do we select the nodes that are destined to be superpeers?
- This problem closely related to leader-election problem, which deals with electing superpeers in a peer-to-peer network.

Hybrid Architectures

- Many distributed systems combine architectural features.
- We look at distributed systems that combine client-server solutions with decentralized architectures.
- **Edge-Server Systems**
  - Their organization is based on a hybrid architecture.
  - They are deployed on the Internet, with their servers being situated "at the edge" of the network.
    - The edge is formed by the boundary between enterprise networks and the actual Internet.

- Clients connect to the Internet via the edge server.
- We can consider an ISP as an edge server, where users connect to the Internet via their ISPs.

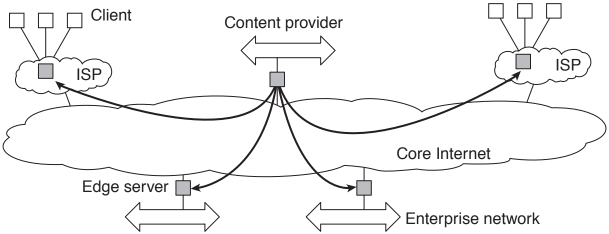

Figure: The Internet as consisting of a collection of edge servers

- Edge server serves content, after applying filtering and transcoding functions.
- In an organization, a single edge server can be used as an *origin server* that generates all content.

- **Collaborative Distributed Systems**
    - These systems also contain hybrid structures.
    - Their main objective is to support collaboration between communicating entities.
    - These systems “start off” by using a traditional client-server scheme.
        - Then, after joining system, a node uses a fully decentralized scheme for collaboration.

- **Example 1: BitTorrent**
  - A file-sharing system that uses the P2P scheme for downloading files.
  - Useful picture

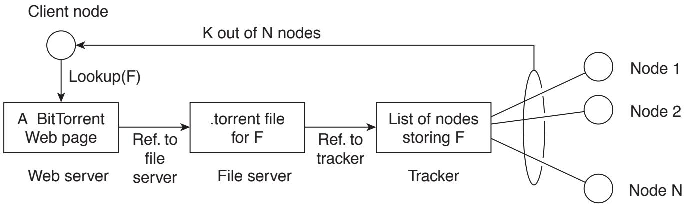

Figure: The principal working of BitTorrent

- An end user looking for a file downloads chunks of it from other users. These chunks are then assembled to produce the complete file
- An important design goal behind BitTorrent was to ensure collaboration.
- A file can only be downloaded on condition the downloading client is providing content to someone else (so-called "tit-for-tat" approach).
- **Read** the details on how BitTorrent files are downloaded (last two paragraphs, **pages 53-54** of textbook).

- **Example 2:** Read about the *Globule* system, on **page 54** of textbook.

Architectures versus Middleware

- Middleware forms layer between applications and distributed platforms:

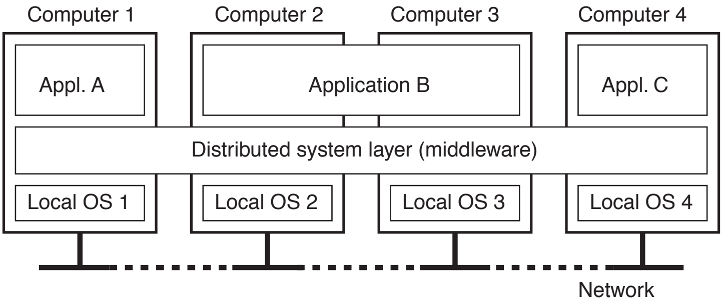

Figure: A distributed system organized as middleware

- Important purpose is to provide degree of distribution transparency.
  - We want to hide from applications aspects such as data distribution, processing and control.
- Middleware systems follow a specific architectural style.
  - Many middleware solutions, such as CORBA, have adopted an *object-based* architectural style.
  - Others, such as TIB/Rendezvous, follow event-based architectural style.
- Using middleware of a particular architectural style makes it simpler to design applications.
  - But, the middleware may no longer be suitable for the application developer's purpose.
  - For example, adding new features to the middleware may yield bloated middleware solutions.

- Middleware is meant to provide distribution transparency.
- However, specific middleware solutions should be modified to suit application requirements.
- One solution is to come up with several versions of a middleware system.
    - Each version is tailored to a specific class of applications.
- A better approach is to come up with middleware systems that are easy to configure, adapt and customize, as required by an application.
- Using several mechanisms, the behaviour of middleware can be modified.

Interceptors

- This is a software construct that breaks the usual flow of control and allows other code to be executed.
- Making interceptors generic requires substantial implementation effort.
- On other hand, limited interception improves management of the software and the distributed system as a whole.
- **Example**: Interceptors in object-based distributed systems.
  - Consider an object A calling a method belonging to another object B, located on a different machine.

- This remote-object invocation follows a three-step approach:
  1. A's local interface is the same as B's. A calls a method available in that interface.
  2. A's call is changed into a generic object invocation.
  3. The generic object invocation is transformed into a message that is sent through the transport-level network interface offered by A's local OS.

- The scheme “works” as follows:

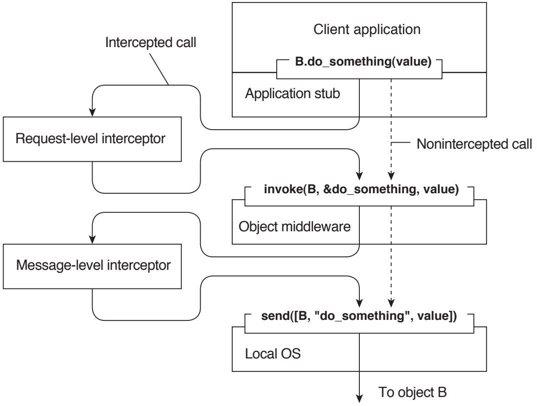

Figure: Using interceptors to handle remote-object invocation

- After the first step, the call

  ```
  B.do_something(value)
  ```

  is transformed by the interceptor into a general call such as

  ```
  invoke(B,&do_something,value),
  ```

  with a reference to B's method and the parameters that go along with the call.
- Suppose object B is replicated.
  - In such a situation, each replica should be invoked.
- This is a case where interception is able to help.
  - The *request-level interceptor* calls

    ```
    invoke(B,&do_something,value),
    ```

    for each of the replicas.

  - The good thing about this is that object A need not be aware of the replication of B. The middleware also does not need any special components to handle this replicated call.
  - Only the request-level interceptor, which may be *added* to the middleware, needs to know about B's replication.

- The call to the remote object B must be conveyed over the network, which requires that the messaging interface offered by the local OS be invoked.
  - A *message-level interceptor* will be used to transfer the invocation to the target object.

General Approaches to Adaptive Software

- Interceptors offer the means to adapt the middleware.
- Adaptation is necessitated by continuous changes in the distributed application environment.
    - These changes are due to several factors: mobility, strong variance in the quality-of-service of networks, failing hardware and battery damage.
- The middleware, and not the applications, must be able to react to these changes.
- These strong environmental influences have prompted middleware designers to consider constructing *adaptive software*.

- Adaptive software has not lived up to its promise.
- However, it is considered by many researchers and developers to be an important aspect of distributed systems.
- The following techniques "characterize" software adaptation:
  1. Separation of concerns
     - Separate the parts that implement functionality from those that take care of other things (i.e. *extra functionalities*).
  2. Computational reflection
     - The ability of a program to inspect itself and, if necessary, adapt its behaviour.
  3. Component-based design
     - It supports adaptation through composition.

The Feedback Control Model

- Focus on organization of distributed systems as high-level feedback-control systems catering for automatic adaptations to changes.
  - Phenomenon is also known as *autonomic computing* or *self-star* systems.
- There are different views regarding self-managing systems.
- However, the common assumption is that adaptations occur through one or more *feedback control loops*.
  - Systems organized on this basis are known as *feedback control systems*.

- The following figure illustrates the basic idea of a feedback control system.

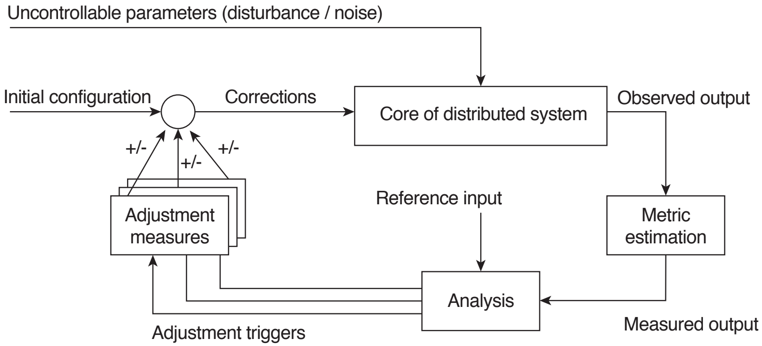

Figure: The logical organization of a feedback control system

- The components being managed form the core of feedback control system.
    - Assumption: Components are driven through controllable input parameters.
        - However, their behaviour may be influenced by all kinds of uncontrollable input, also known as disturbance or noise input.
- Three elements constitute a feedback control loop.
    - First, the system must be monitored by measuring its various aspects.
        - Measuring such behaviour is not easy.
        - Another problem is when a node A must estimate the latency between two other completely different nodes B and C, without being able to intrude on either two nodes.
        - For these reasons, a feedback control loop contains *metric estimation component*.

- Another part of a feedback control loop analyzes measurements and compares them to reference values.
    - This *feedback analysis component* forms the heart of the control loop.
    - It contains algorithms to be used when adaptations need to be made.
- The last group of components contain mechanisms that influence system behaviour.
    - The analysis component will trigger one or several of these mechanisms, and later observe the effect.

- **Reading Task:** Study `Section 2.4.3` **Differentiating Replication Strategies in Globule: pp. 63-65** of textbook). Include Section 2.4.3.
    - Do **NOT** read `Section2.4.4`.

References I

- TU Graz, 2008. Architectural Styles, http://coronet.iicm.tugraz.at/sa/s5/sa_styles.html
- Pree W., 2005. Architectural styles (according to CMU's SEI), http://www.softwareresearch.net/fileadmin/src/.../02_ArchStyles
- Dr Dobb's, 2008. Event-Based Architectures, http://drdobbs.com/architecture-and-design/208801141
- Tanenbaum A. S., Van Steen M., 2007. Distributed Systems - Principles and Paradigms, Second Edition, Prentice Hall.
- Wikepedia, 2011. Publish/subscribe, http://en.wikipedia.org/w/index.php,last modified on 22 September 2011 at 09:56.

References II

- Sommerville I., 2004. Software Engineering, 7th Edition, Chapter 12.
- Kuz I., Rauch F., Chakravarty M. M. T., Heiser G. COMP9243 - Week 2 (10s1), The University *of* New South Wales, School *of* Computer Science & Engineering