DSC516/EPL602: Cloud Computing

Part I: Basic Concepts and Models

Module1: Distributed Computing Concepts and Models

University of Cyprus
Department of Computer Science

Learning Objectives

- Review and explain key concepts: Architecture, System Architecture, Resource, Middleware, Client, Server, COD, REV, Middleware, End-to-End Arguments in Systems Design.
- Review, explain and apply distributed computing models, client-server computing, etc.
- Understand and explain function decomposition concerns in the design of distributed systems.

University of Cyprus
Department of Computer Science

Lecture 2a

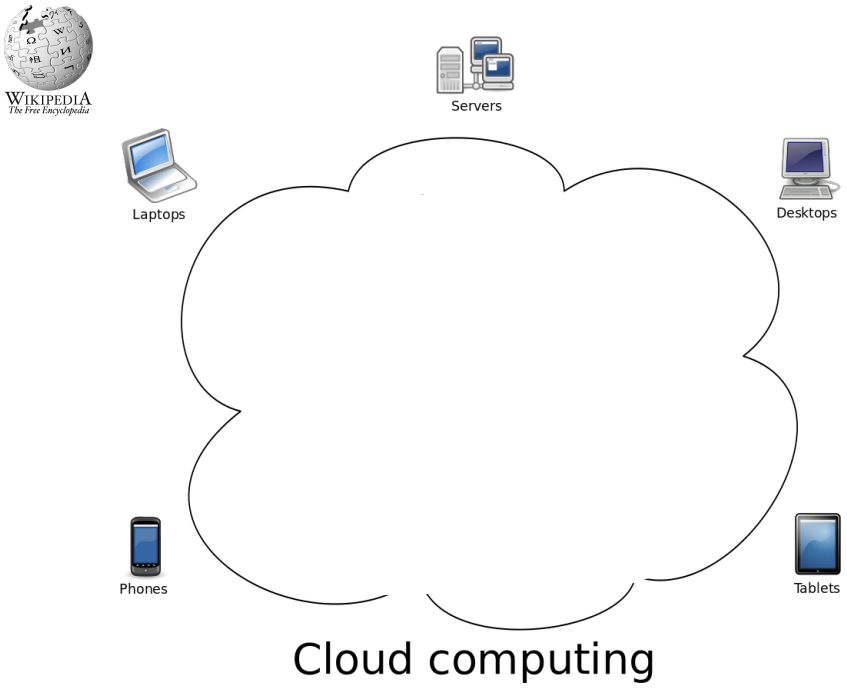
Distributed Computing: Concepts, Models, Middleware

University of Cyprus
Department of Computer Science

Readings

- Aiken, B., Strassner, J., Carpenter, B., Foster, I., Lynch, C., Mambretti, J., Moore, R., and Teitelbaum, B. **"Network Policy and Services: A Report of a Workshop on Middleware"**, RFC 2768.," 2000.
- Gazis, A., Katsiri, E., **Middleware 101,** Communications of the ACM, Volume 65, Issue 9, September 2022, pp. 38-42.
- J. H. Saltzer, D. P. Reed, and D. D. Clark, **"End-to-end arguments in system design"** ACM Trans. Comput. Syst., vol. 2, no. 4, pp. 277–288, Nov. 1984.
- A. Carzaniga, G. Pietro Picco and G. Vigna, **"Designing distributed applications with mobile code paradigms"** Proceeding ICSE '97 Proceedings of the 19th international conference on Software engineering, 1997.
- Client-Server Model (Wikipedia).

University of Cyprus
Department of Computer Science



Distributed Systems

- Hardware and software components located on networked computers that communicate on message passing
- Aspects:
  - Concurrent access
  - Coordination (no global clock)
  - Failures
- Examples
  - The 'Internet'
  - Intranets
  - Mobile and ad-hoc networks
  - P2P
  - Cloud

Distributed Systems Challenges

- Heterogeneity
  - See middleware
- Openness
  - See APIs
- Security
  - Encryption, types of attacks
- Scalability
  - In adding new nodes/users and service access
- Failure Handling
  - Deal with multiple types of failures
- Concurrency
  - Safe concurrent access
- Transparency

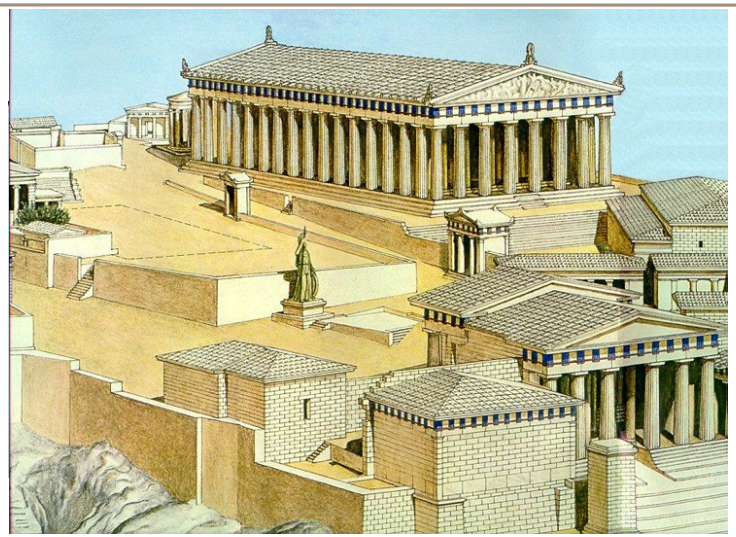
Distributed Computing: Concepts, Models, Middleware

Key concepts and Abstractions

Descriptive Models

- Models are intended to provide an abstract, simplified and consistent description of a relevant aspect of Distributed System design.
- Architectural models of D/S define the way in which components of systems:
  - Interact with one another
  - Are mapped onto an underlying network of computers
- Represent (abstract) key principles used to design and build systems.

Architectural Models



Architecture of D/S

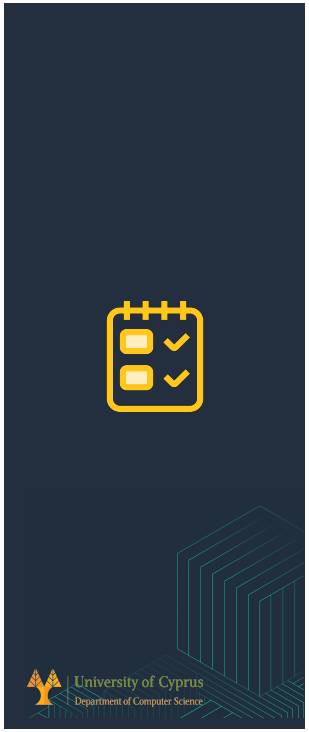
- Architecture of a system: its structure in terms of separately specified **components**.
- Goal: to ensure that the structure will meet present and likely future demands on it.
- Several Concerns:
  - Reliability
  - Manageability
  - Adaptability
  - Cost-effectiveness

Why Discuss Architecture?

- **Descriptive**
  - Provide a common vocabulary for use when describing systems
- **Guidance**
  - Identify key areas in which services are required
- **Prescriptive**
  - Define standard protocols and APIs to facilitate creation of interoperable systems and portable applications

Architectural Models

- Simplify and abstract the **functions** of the individual components of a D/S.
- Represent the **placement** of the components across a network of computers, seeking to define useful **patterns** for distribution of data and workload.
- Capture the **interrelationships** between the components – that is, their functional roles and the patterns of communication between them.



Can you identify and name components of a distributed system we presented in the previous lecture and identify their functions?

Resource components

- Components: composing elements of an architecture
- Resource components:
  - Embody architectural elements representing passive data or physical devices
  - Code components - contain the implementation of an algorithm necessary to execute a particular task
- Entities meant to be shared
  - E.g., computers, storage, data, software
  - Not necessarily physical
    - E.g., Task pool, distributed file system, …
  - Can be defined in terms of interfaces, not devices
    - E.g. scheduler such as LSF and PBS define a compute resource
    - Open/close/read/write define access to a distributed file system, e.g. NFS, AFS, DFS

Computational components

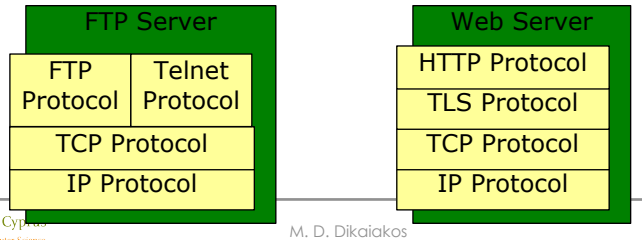
- Embody a **flow of control**
  - E.g. process, thread
- Characterized by **state**, which includes:
  - Private data
  - State of execution
  - Bindings to other components (code and resource)
- Examples:
  - Client processes (διεργασίες πελάτη)
  - Server processes (διεργασίες εξυπηρετητή) : a process accepting requests from other processes.
  - Peer processes (ομότιμες διεργασίες)

Interactions & sites

- Interactions: Events that involve 2 or more components
  - E.g. a message exchanged between 2 computational components
- Sites embody the intuitive notion of location
  - Execution environments: they host components and provide support for the execution of computational components.
  - **Local** vs. **remote** interactions.

Network Enabled Services

- Implementation of a protocol that defines a set of capabilities
  - Protocol defines interaction with service
  - All services require protocols
  - Not all protocols are used to provide services (e.g. IP)
- Examples: FTP and Web servers



Network Protocol

- A formal description of message formats and a set of rules for message exchange
  - Rules may define sequence of message exchanges
  - Protocol may define state-change in endpoint, e.g., file system state change
- Good protocols designed to do one thing
  - Protocols can be layered
- Examples of protocols
  - IP, TCP, TLS (was SSL), HTTP, Kerberos

Application Programming Interface (API)

- A specification for a set of routines to facilitate application development
  - Refers to definition, not implementation
  - E.g., there are many implementations of MPI
- Specification is often language-specific (or IDL)
  - Routine name; number, order and type of arguments; mapping to language constructs
  - Behavior or function of routine
- Examples
  - GSS API (security), MPI (message passing)

Software Development Kit

- A particular instantiation of an API
- SDK consists of libraries and tools
  - Provides implementation of API specification
- Can have multiple SDKs for an API
- Examples of SDKs
  - MPICH, Motif Widgets

Syntax

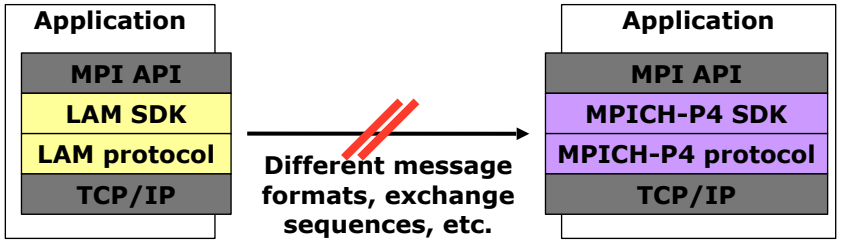
- Rules for encoding information, e.g.
  - XML, Condor ClassAds, Globus RSL, TOSCA
  - X.509 certificate format (RFC 2459)
  - Cryptographic Message Syntax (RFC 2630)
- Distinct from protocols
  - One syntax may be used by many protocols (e.g., XML, JSON); & useful for other purposes
- Syntaxes may be layered
  - E.g., Condor ClassAds -> XML -> ASCII
  - Important to understand layerings when comparing or evaluating syntaxes

A Protocol can have Multiple APIs

- TCP/IP APIs include BSD sockets, Winsock, System V streams, …
- The protocol provides interoperability: programs using different APIs can exchange information
- I don't need to know remote user's API

An API can have Multiple Protocols

- MPI provides portability: any correct program compiles & runs on a platform
- Does **not** provide interoperability: all processes must link against same SDK
  - E.g., MPICH and LAM versions of MPI



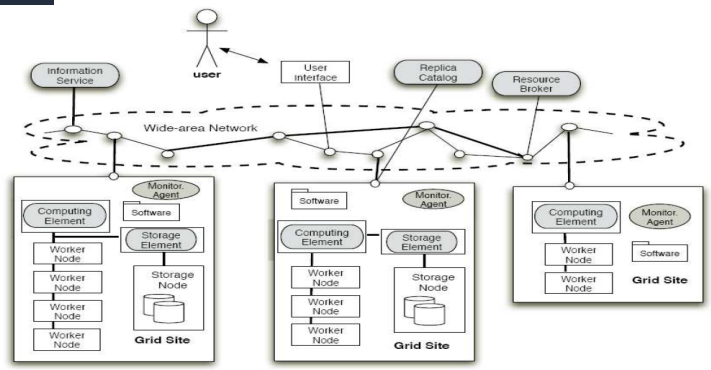
APIs & Protocols: both Important

- Standard APIs/SDKs are important
  - They enable application portability
  - But w/o standard protocols, interoperability is hard (every SDK speaks every protocol?)
- Standard protocols are important
  - Enable cross-site interoperability
  - Enable shared infrastructure
  - But w/o standard APIs/SDKs, application portability is hard (different platforms access protocols in different ways)

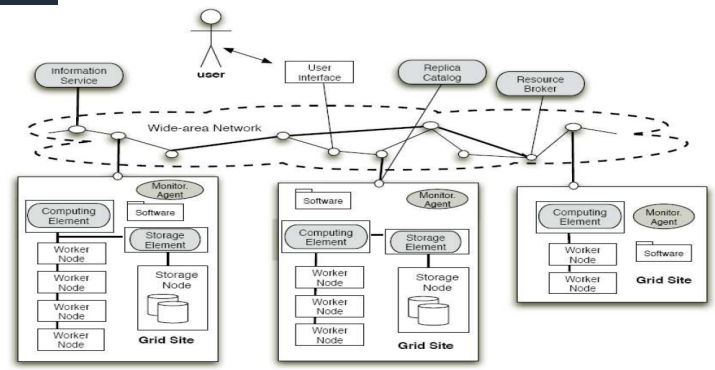
Programming & Systems Problems

- The programming problem
  - Facilitate development of sophisticated apps
  - Facilitate code sharing
  - Requires programming environments: APIs, SDKs, tools
- The systems problem
  - Facilitate coordinated use of diverse resources
  - Facilitate infrastructure sharing
    - e.g., certificate authorities, information services
  - Requires systems: protocols, services

In the Grid architecture below, can you identify resource components, computational components, sites, services, interrelationships?



In the Grid architecture below, where would you place a scheduler?



Distributed Computing: Concepts, Models, Middleware

Architecture Models

System Architecture

- The structuring of software as layers or modules in a single site or in terms of services offered and requested between processes located in the same or different sites.
  - Client-server
  - Alternatives:
    - Client-proxy-server
    - Push model
    - Remote evaluation
    - Code on demand
    - Mobile agent: running program+data migrating
    - Peer to Peer

System Architecture Elements

- Client processes (διεργασίες πελάτη)
- Server processes (διεργασίες εξυπηρετητή) : a process accepting requests from other processes.
- Peer processes (ομότιμες διεργασίες)
- Software architecture: the structuring of software as layers or modules in a single computer or in terms of services offered and requested between processes located in the same or different computers.

System Architecture: Client/Server

- Client-server model: a distributed application structure that partitions tasks or workloads between the providers of a resource or service (**servers**), and service requesters (**clients**).
- Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system.
- A **server host** runs one or more server programs, which share their resources with clients.
- **Services**: can be provided by multiple servers in separate hosts. Examples:
  - Partitioned data: Web server example.
  - Replicated data: Network Information Service (NIS) example.
- Replication is used to increase performance and availability and to improve fault tolerance – provides multiple consistent copies of data in processes running in different computers.
  - At what cost?

Variations of Client-Server

- Mobile code, e.g., applets, Javascript (Code-on-demand).
- Push model: the server instead of the client initiates interactions.
- Mobile agents: running program+data migrating.
- Network computers or thin clients

A. Carzaniga, G. Pietro Picco and G. Vigna, **"Designing distributed applications with mobile code paradigms"** Proceeding ICSE '97 Proceedings of the 19th international conference on Software engineering, 1997.

Implementing a service

- A computational component A located at a site S_A needs the result of the computation of a service.
- Assume the existence of another site S_B, which will be involved in the delivery of the service.
- To obtain service results, A starts an interaction pattern that leads to service delivery.
- Service execution involves:
  - A set of **resources**
  - **Know-how** about the service (its code)
  - A **computational component** responsible for the execution of the code

These need to be located at one site at the same site.

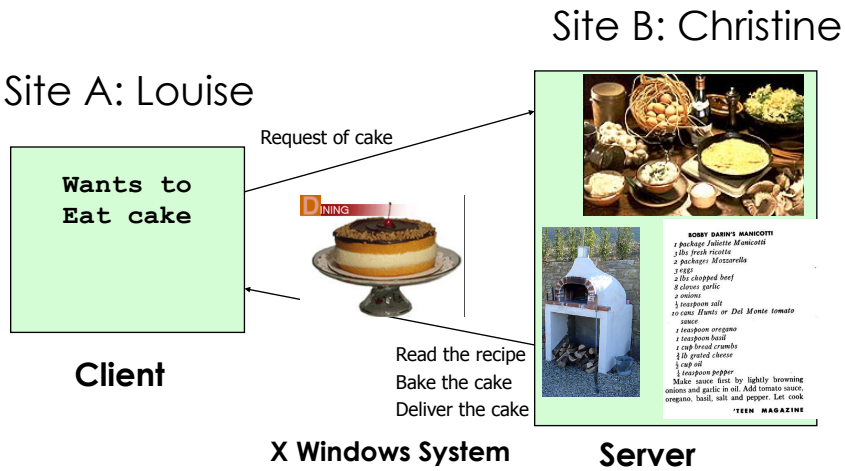
A. Carzaniga, G. Pietro Picco and G. Vigna, **"Designing distributed applications with mobile code paradigms"** Proceeding ICSE '97 Proceedings of the 19th international conference on Software engineering, 1997.

Louise and Christine make a cake

- Cake -- result of the service
- Recipe -- know-how / code
- Ingredients -- resource components / data
- Louise -- computational component A (process)
- Christine -- computational component B (process)
- Louise's home -- Site A
- Christine's home -- Site B

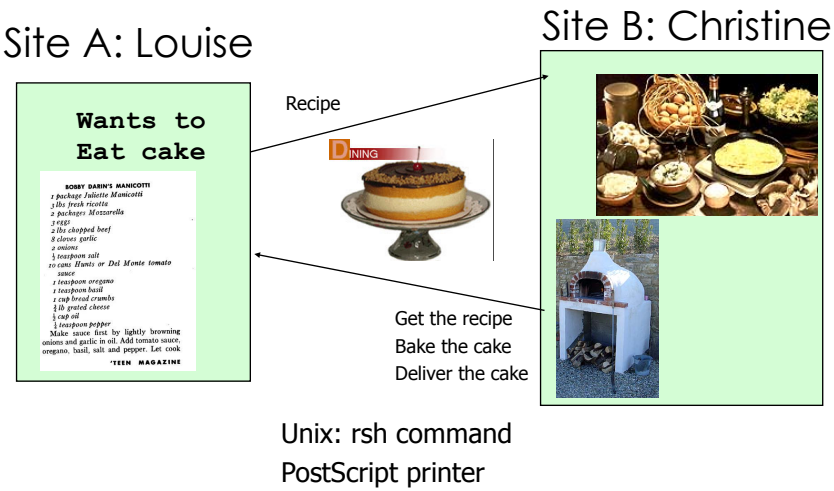
A. Carzaniga, G. Pietro Picco and G. Vigna, **"Designing distributed applications with mobile code paradigms"** Proceeding ICSE '97 Proceedings of the 19th international conference on Software engineering, 1997.

Traditional Client and Server Model



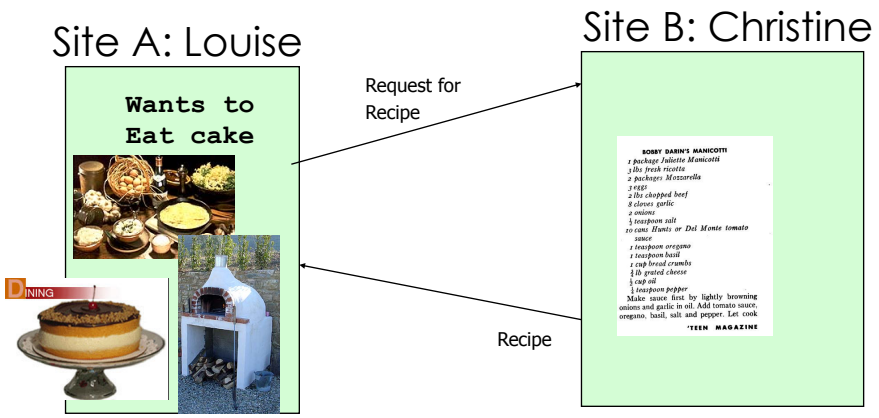
A. Carzaniga, G. Pietro Picco and G. Vigna, **"Designing distributed applications with mobile code paradigms"** Proceeding ICSE '97 Proceedings of the 19th international conference on Software engineering, 1997.

Remote Evaluation Model: (REV)



A. Carzaniga, G. Pietro Picco and G. Vigna, **"Designing distributed applications with mobile code paradigms"** Proceeding ICSE '97 Proceedings of the 19th international conference on Software engineering, 1997.

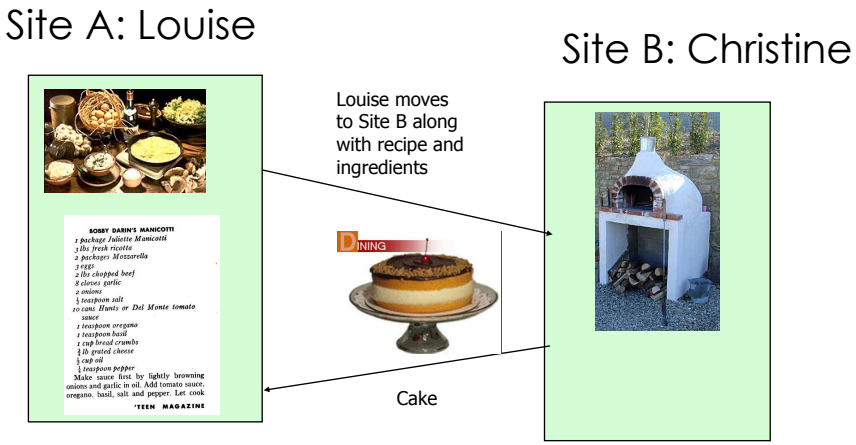
Code on Demand (COD)



Terminal gets a new type of document. Document header may contain a reference (URL address) to the code that is needed to interpret the document. Then the principle will go to the reference and download the necessary code and execute it afterwards.

A. Carzaniga, G. Pietro Picco and G. Vigna, **"Designing distributed applications with mobile code paradigms"** Proceeding ICSE '97 Proceedings of the 19th international conference on Software engineering, 1997.

Mobile Agent Model: (MA)



A. Carzaniga, G. Pietro Picco and G. Vigna, **"Designing distributed applications with mobile code paradigms"** Proceeding ICSE '97 Proceedings of the 19th international conference on Software engineering, 1997.

Paradigm Recap

	Before		After	
Paradigm	Site A	Site B	Site A	Site B
Client – Server	A	know-how resources B	A	know-how resources B
Remote Evaluation	know-how A	resources B	A	know-how resources B
Code on Demand	resources A	know-how B	resources know-how A	B
Mobile Agent	know-how A	resources	---	know-how resources A

A and B are already in execution

A. Carzaniga, G. Pietro Picco and G. Vigna, **"Designing distributed applications with mobile code paradigms"** Proceeding ICSE '97 Proceedings of the 19th international conference on Software engineering, 1997.

Choosing the Right Paradigm

- No paradigm is absolutely better than others.
- The choice of paradigm must be performed on a case-by-case basis, taking into account issues such as the cost of network communication, availability and performance of resources, etc.

Distributed Computing: Concepts, Models, Middleware

Middleware

Platforms and Middleware

- Platforms for D/S and applications: the lowest-level hardware and software layers. Provide services to layers above them.
- Middleware (μεσολογισμικό;) a layer of software whose purpose is to mask heterogeneity and provide a convenient programming model to application programmers.
- Middleware is represented by processes or objects in a set of computers that interact with each other to achieve communication and resource *sharing*.

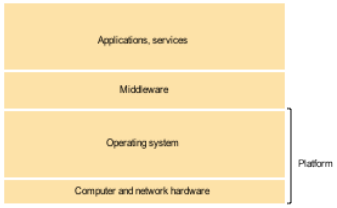
Definitions of Middleware

- *Software & DevOps engineer perspective*: the layer that "glues" together software by different system components
- *Network engineer*: the fault-tolerant and error-checking integration of network connections - communication management software.
- *Data engineer*: the technology responsible for coordinating, triggering, and orchestrating actions to process and publish data from various sources, harnessing big data and the IoT.

Gazis, A., Katsiri, E., **Middleware 101**, Communications of the ACM, Volume 65, Issue 9, September 2022, pp. 38-42.

Middleware (ctd)

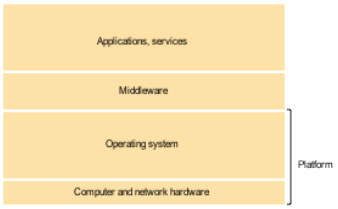
- Middleware is concerned with providing useful building blocks for the construction of software components that can work with one another in a D/S.
- It enables communication at higher levels of abstraction by providing things like:
  - remote method invocation
  - group communication
  - event notification
  - data replication
  - real-time transmission of data.



Aiken, B., Strassner, J., Carpenter, B., Foster, I., Lynch, C., Mambretti, J., Moore, R., and Teitelbaum, B. **"Network Policy and Services: A Report of a Workshop on Middleware"**, RFC 2768.," 2000.

Middleware (ctd)

- Middleware can also provide infrastructural services for use by application programs:
  - Naming
  - Security
  - Transactions
  - Persistent storage
  - Event notification



Categories of Middleware

Based on architecture operating principles:

- Transactional. Processing of **multiple synchronous/asynchronous transactions**
  - Serving as a cluster of associated requests from distributed systems such as bank transactions or credit card payments.
- Message-oriented. Message queue and message passing architectures, which support **synchronous/asynchronous communication**.
  - The first operates based on the principle that a queue is used to process information, whereas the second typically operates on a publish/subscribe pattern where an intermediate broker facilitates the communication.
- Procedural. **Remote** and **local architectures** to connect, pass, and retrieve software responses of asynchronous communications (e.g. a call operation).
  - Remote architecture calls a predetermined service of another computer in a network; local architecture interacts solely with a local software component.
- Object-oriented. Similar to procedural, however, this type incorporates OO programming design principles: object references, exceptions, and inheritance of properties via distributed object requests. Typically used synchronously, but can support asynchronous communication via the use of threads and concurrent programming.

Gazis, A., Katsiri, E., **Middleware 101,** Communications of the ACM, Volume 65, Issue 9, September 2022, pp. 38-42.

Categories of Middleware

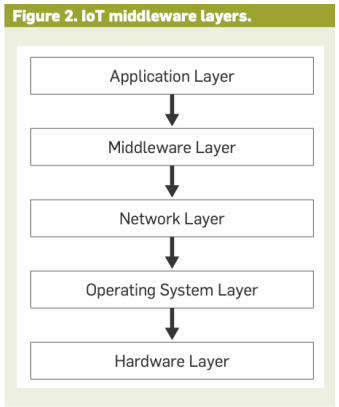
Application-driven:

- Reflective middleware: designed to "easily operate with other components and applications."
- Agent middleware: has multiple components that operate on complex domain-specific languages and laws.
- Database middleware: focuses on DB-to-DB or DB-to-apps communication—either natively or via call-level interfaces (CLIs).
- Embedded middleware: acts as the intermediary for embedded integration apps and operating-system communication.
- Portal middleware creates a context-management tool in a composite, single-screen application.
- Device (or robotics) middleware simplifies the integration of specific device operating systems or robotic hardware and firmware.

Gazis, A., Katsiri, E., **Middleware 101,** Communications of the ACM, Volume 65, Issue 9, September 2022, pp. 38-42.

Modern Middleware Capabilities

- Supports an approach for developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.
- Closely connected to APIs (application protocol interfaces), serving as the tier or a software bundle for different APIs used by a programmer.
- Simplifies sophisticated applications so that the developer focuses on not only the communication of components but also the business logic and the systems' interaction.



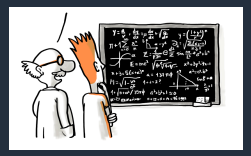
Gazis, A., Katsiri, E., **Middleware 101**, Communications of the ACM, Volume 65, Issue 9, September 2022, pp. 38-42.

Lecture 2b

End-to-end Arguments in System Design

University of Cyprus
Department of Computer Science

In previous lecture



Discussed and explained some basic concepts:

- Abstraction
- Architecture
- System Architecture
- Resource, Physical and Logical
- Process
- Distributed Computing Models: Client-Server, REV, COD
- Middleware
- Middleware services and categories

University of Cyprus
Department of Computer Science

Knowledge Check

- What information constitutes the state of a resource component?
- What is an execution environment? Give an example.
- What is a socket?
- Explain the difference between Remote Evaluation and Client Server models. Explain one problem that REV has for the security of systems.
- Describe and explain a scenario where a Mobile Agent model may be preferable over Client Server.
- Explain the differences between synchronous and asynchronous communication.
- Give a definition of middleware and 3 examples of middleware services.

University of Cyprus
Department of Computer Science

Distributed Computing: Concepts, Models, Middleware

End-to-end Arguments in System Design

University of Cyprus
Department of Computer Science

Readings

J. H. Saltzer, D. P. Reed, and D. D. Clark, **"End-to-end arguments in system design"** ACM Trans. Comput. Syst., vol. 2, no. 4, pp. 277–288, Nov. 1984.

University of Cyprus
Department of Computer Science

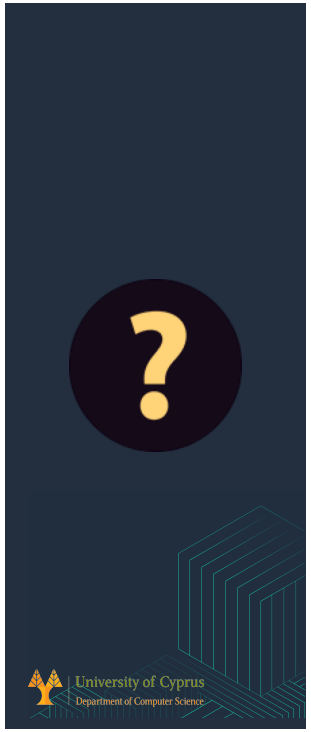
Managing Complexity

- Despite their incredible complexity, computer systems exist and continue to evolve because they are designed as hierarchies with well-defined interfaces that separate levels of abstraction.
- Using well-defined interfaces facilitates independent subsystem development by both hardware and software design teams.
- The simplifying abstractions hide lower level implementation details, thereby reducing the complexity of the design process.

University of Cyprus
Department of Computer Science

How do we decide where we place the functionalities required for a system/ application we wish to develop?

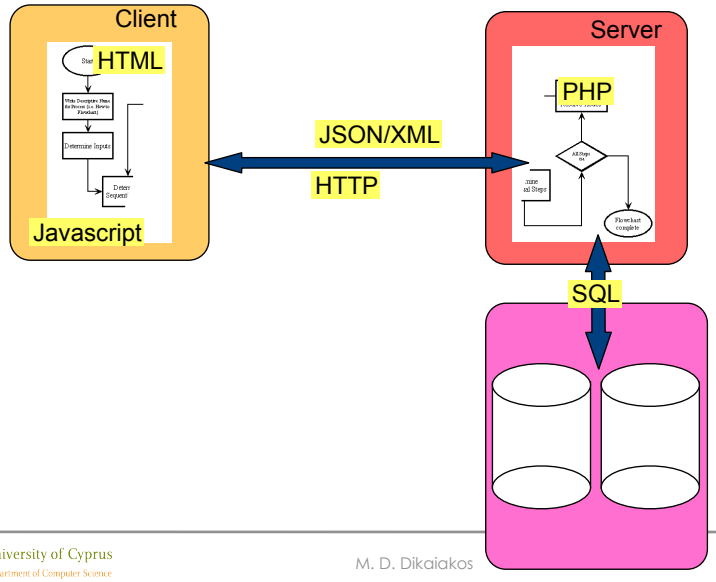
University of Cyprus
Department of Computer Science



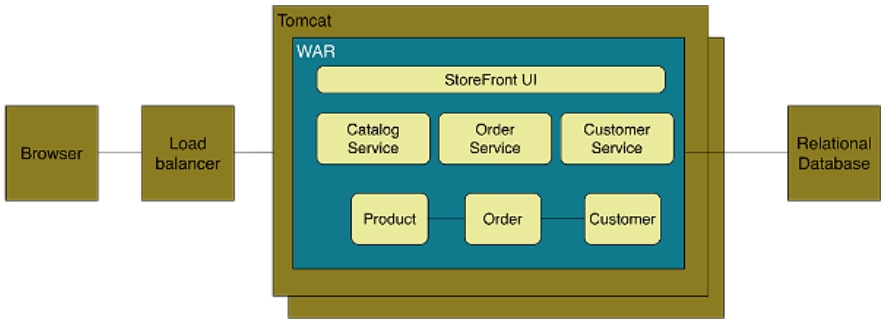
How do you call this problem?

Functional Decomposition

Example 1: "Three-tier"

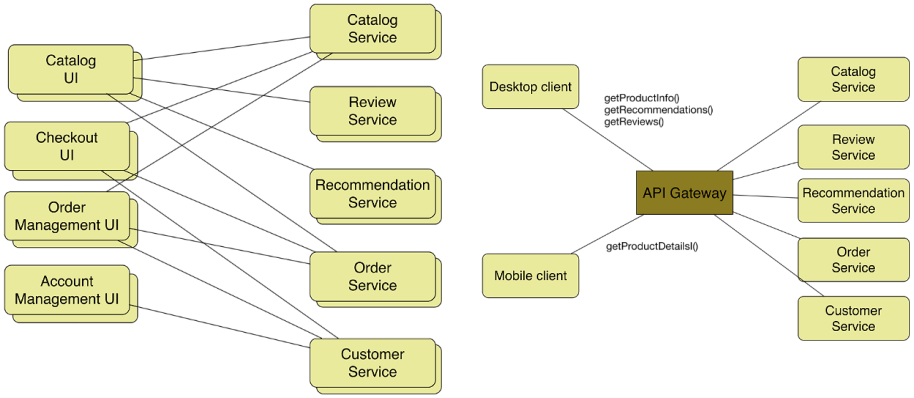


Example 2: "Monolithic" Architecture

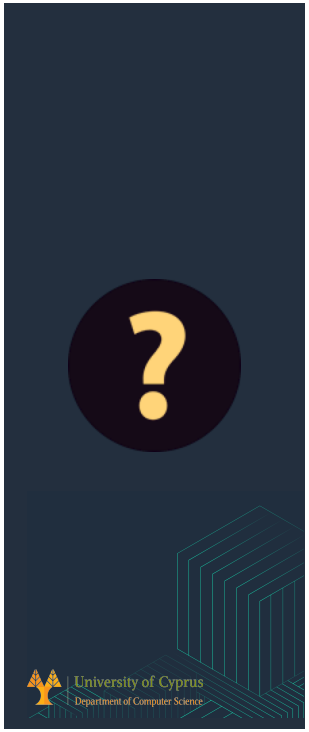


https://www.infoq.com/articles/microservices-intro/

Example 3: Microservices



More on this later

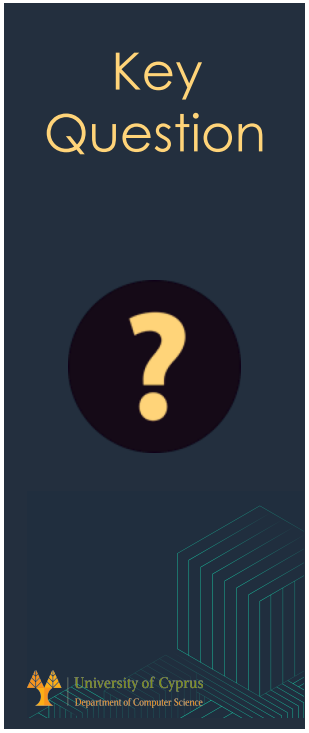


Where are you going to look for solutions to this problem?

System Design

- Choosing the proper boundaries between functions is a primary activity of the **computer system designer**.
- **Design principles** that provide guidance in this choice of function placement are **among the most important tools of a system designer**.
- In systems involving communication a designer usually:
  - Draws a modular boundary around comm. subsystem
  - Firm interface between comm. subsystem and the rest of the system.

Key Question



Where do we implement a system's functionality?

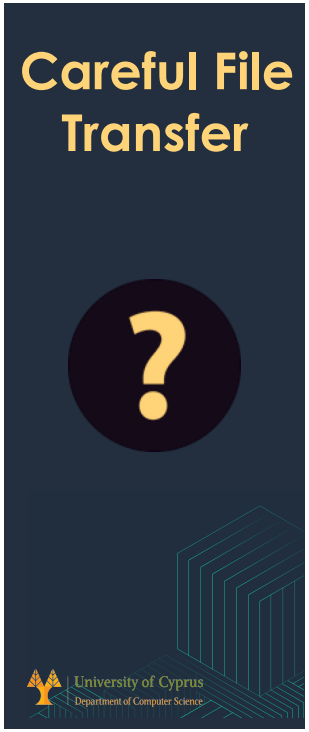
- In the communication subsystem?
- In the clients of the comm. subsystem?
- As a joint venture
- Redundantly?

End-to-end argument

*"The function in question can be **completely** and **correctly** be implemented **only** with the **knowledge** and **help** of the **application** standing at the endpoints of the communication system"*

- Therefore, providing that questioned function as a feature of the communication subsystem itself is not possible.
  - Sometimes, an incomplete version of the function be provided by the comm. subsystem may be useful as a performance enhancement.

Careful File Transfer

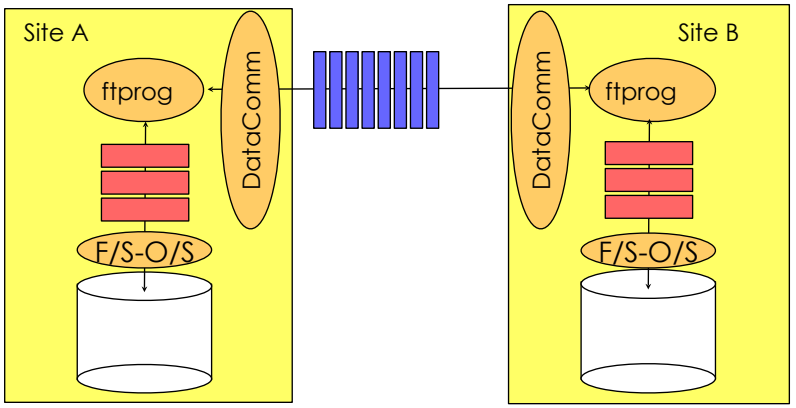


- Suppose you are asked to write a program called **ftprog** that can move a file from one computer to another, over the network.
- What do you do?

Careful File Transfer

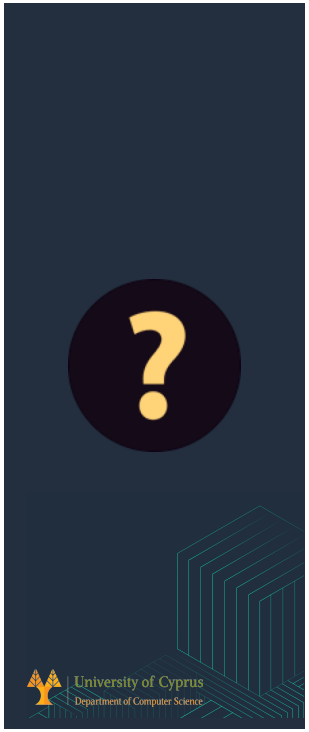
- Move file from computer A to computer B without damage.
- Steps taken:
  1. At host A, the file transfer program calls the file system to read the file from disk. The f/s passes the file to the file transfer program in fixed-sized blocks chosen to be disk format independent.
  2. At host A, the ftprog asks the data communication system to transmit the file using some communication protocol that involves splitting the data into packets. The packet size is typically different from the file block size and the disk track size.
  3. The data communication network moves packets from A to B.
  4. At host B, the data communication program removes packets from the protocol and hands the contained data to a second part of the data transfer application operating on B.
  5. At host B the file transfer program asks the file system to write the received data on the disk of host B.

Careful File Transfer



Threats to transaction

1. Hardware faults in the disk storage result to reading incorrect data.
2. File system software or file transfer program or data communication system make a mistake in buffering and copying the data of the file either at A or B.
3. Hardware processor or local memory have transient error while doing buffering and copying at A or B.
4. Communication system drops or changes bits in a packet or deliver a packet more than once.
5. Either of the hosts may crash part way through the transaction after performing an unknown amount of the transaction.



How can we deal with errors?

Errors:

1. Hardware faults in the disk storage result to **reading incorrect data**.
2. File system software or file transfer program or data communication system make a mistake in **buffering and copying** the data of the file either at A or B.
3. Hardware processor or local memory have **transient error while doing buffering and copying** at A or B.
4. Communication system **drops or changes bits** in a packet or deliver a packet more than once.
5. Either of the hosts may **crash part way** through the transaction after performing an unknown amount of the transaction.

Dealing with threats

- Reinforce each of the steps along the way using **duplicate** copies, **time-out** and **retry**, carefully located redundancy for **error detection**, crash **recovery**, etc.
- Systematic countering of threat (2) requires writing correct programs…
  - Not all programs are written by the file-transfer programmer.
- Using tri-modular redundancy for the whole process…

End-to-end Check and Retry

- Suppose that a checksum is **stored with each file**, reducing the possibility of error to an acceptably negligible value.
- Then, transfer file from A to B.
- FT Application at B reads the file back to its memory, computes the checksum and sends the value back to A for comparison.
- If comparison fails, retry…

What if?

- The communication subsystem provides internally a guarantee for reliable data transmission, through:
  - Selective redundancy in the form of packet checksums
  - Sequence number checking
  - Internal retry mechanisms
- We can lower the probability of dropped bits to a very small number, and eliminate threat (4).
- Henceforth, we achieve a reduction of the frequency of retries by the file transfer application.
- What is the effect on the correctness of the file-transfer outcome?
  - Still in need to counter the other threats at the end-level

Effect

- Some extra cost (where?)
- If failures are **fairly rare**, this technique will normally work on the first try;
  - occasionally a second or even third try might be required;
- One would probably consider two or more failures on the same file transfer attempt as indicating that some part of the system is in need of repair.

Achievement

- Extra effort expended in the communication system to provide a guarantee of reliable data transmission is **only reducing the frequency of retries** by the file transfer application;
- No effect on inevitability or correctness of the outcome:
  - Correct file transmission is **assured** by *the end-to-end checksum and retry* whether or not the data transmission system is especially reliable.

Conclusion

- To achieve careful file transfer, the application program that performs the transfer must supply a file-transfer-specific, **end-to-end reliability guarantee**.
- For the communication subsystem to go out of its way to be extraordinarily reliable does not reduce the burden on the application program to ensure reliability.

Performance Aspects

- So, should lower levels play no part in obtaining reliability?
- Consider a somewhat unreliable network, dropping a message in each hundred messages sent.
- What is the effect of this, as we transmit files of increasing size?
  - The probability that all packets of a file arrive correctly decreases exponentially with the file length (prove this).
  - So, the expected time to transmit the file grows exponentially with the file length.
- Performance of the file transfer application hurts!

Performance Trade offs

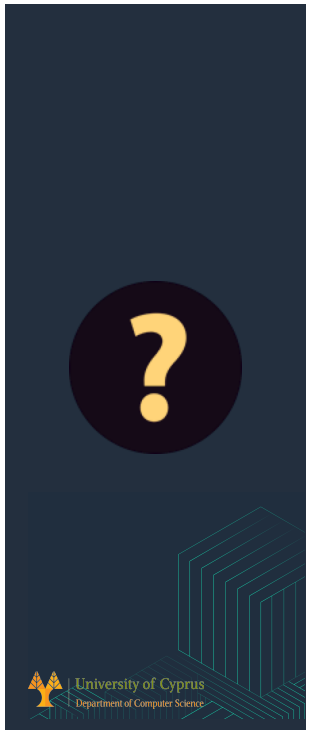
- The amount of effort to be put into reliability measures within the data communication system is an **engineering trade-off** based on **performance**, rather than a requirement for correctness.
- If comm subsystem is **too unreliable**, the file transfer application performance suffers.
- If comm subsystem is **beefed up** with internal reliability measures, those measures have a performance cost:
  - Lost bandwidth to redundant data
  - Added delay from waiting for internal consistency checks to complete
  - And, after all, the end-to-end consistency check is still required, no matter how reliable the communication system becomes.

Performance Trade offs

- Using performance to justify placing functions in a low-level subsystem must be done carefully.
- Sometimes, the same or better performance can be achieved at the high level.
- Performing the function at the low level may:
  - Be more efficient if the function can be performed with minimum perturbation of the machinery already included in the low-level.
  - Cost more because:
    - Most applications using the low-level subsystem do not need the function.
    - The low-level subsystem does not have adequate information, like the higher levels, to do the job efficiently.

Other Examples of the e2e Argument

- Acknowledgment of message delivery:
  - The comm network can easily return an **ack** to the sender, whenever a message is delivered to a recipient.
  - This is **not very helpful** for applications, since:
    - The application wants to know if the target host acted on the message (receipt does not directly translate to action).
    - So, the appl. needs **end-to-end ack**.



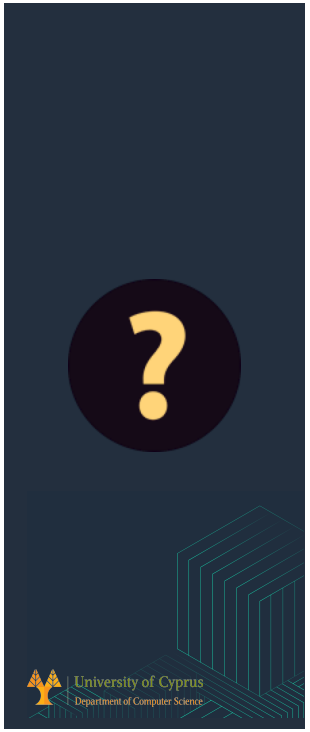
Is the encryption of data by the communication subsystem necessary?

Other Examples of the e2e Argument

- **Secure transmission of data**: if the data communication system performs encryption-decryption:
  - It must be trusted to **manage securely the encryption keys**.
  - Data will be **in the clear** and **vulnerable** while passing from the **communication system** to the **application.**
  - Authenticity of the message must still be checked by the application: if the application performs the encryption, it can also do the authentication checks and keeps its keys.

Secure Data Transmission

- If the **application** performs **end-to-end encryption**:
  - It obtains its required authentication check,
  - Can handle key management to its satisfaction
  - The data is never exposed outside the application
- Thus, to satisfy the requirements of the application, there is no need for the communication subsystem to provide for automatic encryption of all traffic.



Yet, *some network encryption* may be useful.

Why?

Secure Data Transmission

- Automatic encryption of all traffic by the communication subsystem can ensure that a **misbehaving user** or **application** program does not **deliberately** transmit information that should not be exposed.
- The automatic encryption of all data as it is put into the network is one more firewall the system designer can use to ensure that information does not escape outside the system.
- Note that:
  - This is a different requirement from authenticating access rights of a system user to specific parts of the data.
  - Network-level encryption can be **quite unsophisticated** – the same key can be used by all hosts, with frequent changes of the key.

Self-study

Apply the end-to-end arguments to think how to manage the suppression of duplicate messages that are erroneously sent by an application

Identifying the ends

- The end-to-end argument is not an absolute rule but rather a **guideline**:
- Suppose we use a communication subsystem to send voice packets:
  - To support two people carrying a real-time conversation.
  - To transport a speech message (to be listened to later).
- What are the choices we have?

End-to-end arguments now

- The end-to-end argument provided:
  - Innovation
  - reliability
  - In the end, we have a "transparent" network
- This is threatened nowadays by:
  - Loss of trust (e.g., firewalls)
  - ISP control desires
  - 3rd parties wish to observe data flow
  - Caching, mirroring
  - Regulation
- Improve performance of today's apps in favor of new ones?

Future of end-to-end argument (2)

- The end-to-end argument is still valid:
  - But needs redefinition in today's world…
- Evolution of existing apps is inevitable
- Keep the net open and transparent for new apps
- Cope with the loss of transparency