Understanding Networked Applications:
A First Course

Chapter 6

by
David G. Messerschmitt

Goal

• Appreciate the importance of complexity management in networked computing
• Understand better the role of architecture in complexity management
• Examine infrastructure layering in more depth

Complexity

• A system that cannot be understood in all its detail by a single person or small group of people is complex
• The intricacy of the logic embodied in software
  – suffers no physical limitations
  – complexity is a primary limitation
  – advances allow us to extend that complexity

Some sources of complexity

• Problem domain is complex
• Top-down design (as opposed to independent actors in the economy)
• Software is not adaptable like people
• Large team efforts required
• Integration of heterogeneous suppliers

Caution

• The applications considered in this course are relatively simple
• We have addressed
  – only the top of the hierarchy
  – ignored details
  – but this is the essence of hierarchical design: make that which is complex appear simple

Some solutions to complexity

• Modularity properties
  – separation of concerns
  – reuse
• Interoperability through interfaces
  – abstraction
  – encapsulation
Modularity

- A system is modular when it is divided into subsystems (called modules) with good properties
  - Modules have distinct functional groupings
  - Hierarchy supports views at different granularity and scale
  - Separation of concerns among modules
  - Reusability of some modules

Hierarchy in hardware architecture

- Computer subsystem
- Board subsystem
- Integrated circuit subsystem

Physical-world example

<table>
<thead>
<tr>
<th>Poor modularity</th>
<th>Better modularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer service</td>
<td>Customer service</td>
</tr>
<tr>
<td>Loan department</td>
<td>Credit checking</td>
</tr>
<tr>
<td>Janitorial</td>
<td>Floor polishing</td>
</tr>
<tr>
<td>Credit checking</td>
<td>Janitorial</td>
</tr>
<tr>
<td>Physical plant</td>
<td>Floor polishing</td>
</tr>
</tbody>
</table>

Infrastructure example

- Level of interaction high
- Network

<table>
<thead>
<tr>
<th>Poor modularity</th>
<th>Better modularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Application</td>
</tr>
<tr>
<td>End-to-end network</td>
<td>Switch-to-switch</td>
</tr>
<tr>
<td>Host</td>
<td>Host</td>
</tr>
<tr>
<td>Network</td>
<td>Network</td>
</tr>
</tbody>
</table>

Separation of concerns

- The assignment of functionality to different modules should allow them to be designed and implemented as independently as possible
- The level of interaction
  - may be internally high
  - should be externally low
- They can then be assigned to different groups or companies for design
  - minimum coordination costs
**Parts of a module**

- **Module**
- **Interface**
  - What other modules see
- **Implementation**
  - What only the implementer sees

**Interfaces**

- Focus of module interaction and interoperability
- Two purposes:
  - Informs other modules how to interact
  - Informs implementer about what has been promised to other modules

---

**Hardware interface**

- Physical connection
- Electrical properties
- Formats of data passing through the interface (structure and interpretation)

**Possible software interface**

- Menu of actions
  - action-1
  - action-2
  - action-3
  - ...

What are some other examples of types of interaction at interfaces?

---

**Module interaction through interfaces**

- Data customizing an action and disclosing its results
- Both subsystems are affected by the interaction

**Data types**

- Data passing an interface is often specified in terms of a limited number of standard data types
- Data type = range of values and allowable manipulation
- Data type does not presume a specific representation, to allow heterogeneous platforms
  - Representation must be known when data passes a specific module interface
Example data types

• Integer
  – “natural number between -32,767 and +32,768”
  – Could be represented (in many ways) by 16 bits
    • since $2^{16} = 65,536$

• Float
  – “number of the form $m \times 10^n / 32768$, where $m$ is in the
    range -32,767 to +32,768 and $n$ is in the range -255 to
    +256”
  – Could be represented by 16+8 = 24 bits

More data types

• Character
  – “values assuming a-z and A-Z plus space and
    punctuation marks”
    • could be represented by 7 or 8 bits

• Character string
  – “collection of $n$ characters, where $n$ is
    customizable”
    • could be represented by $7^n$ bits

Compound data types

• Programmer-defined composition of basic
data types

• Example:
  Employee {
    String name;
    String address;
    Integer year_of_birth;
    etc.
  }

Protocols

• A defined sequence of actions
  between/among two or more subsystems
  required to achieve some higher-level
  functionality

• Interface specification focuses on actions
  (including formats of parameters and
  returns) and protocols

Example protocol: deposit

Client module

- ① Decides it needs
   to invoke an action
   of a server module
- ② Invokes the
   action by name
- ③ Passes parameter
data to the server
- ④ Processes
   parameters in
   accordance with the
   specified action;
   generates return values
- ⑤ Passes the
   return values back
to the client
- ⑥ Process the
   return values to complete
   the interaction

Server module

Anatomy of an action invocation
More on layering

by

David G. Messerschmitt

Goals

• Understand better
  – how layering is used in the infrastructure
  – how it contains complexity
  – how it coordinates suppliers
  – how it allows new capabilities to be added incrementally

Interaction of layers

Layer above is a client of the layer below

Each layer provides services to the layer above...

...by utilizing the services of the layer below and adding capability

Layer below as a server to the layer above

Layering

Elaboration or specialization

Existing layers

Layering builds capability incrementally by adding to what exists

Three types of software

Application

• Components and frameworks:
  What is in common among applications

• Infrastructure:
  Basic services (communication, storage, concurrency, presentation, etc.)
Part of Microsoft vs. DOJ dispute

Open layer interfaces

Data and information in layers

• The infrastructure should deal with data, or at most minimal structure and interpretation of data suitable for a wide range of applications
• The application adds additional structure and interpretation
• This yields a separation of concerns

Package = file, message

• In the simplest case, the infrastructure deals with a package of data (non-standard terminology)
  – collection of bits
  – specified number and ordering
• The objective of the infrastructure is to store and communicate packages while maintaining data integrity
• File for storage, message for communication
Data integrity

• Retain the
  – values
  – order
  – number
  of bits in a package

Information in the infrastructure

• Sometimes it is appropriate for the infrastructure to assume structure and interpretation for data
  – to add capabilities widely useful to applications
  – to help applications deal with heterogeneous platforms, where representations differ
• At most, data types

Data and information

Application
Deals with information

Assumes structure and interpretation

Assumes standard data types

Infrastructure
Deals with data types

Storage

Application
Deals with information

Assumes standard data types
and SQL = structured query language

Database management system (DBMS)
File system

The infrastructure can provide data management functions

Communication

Application
Deals with information

Assumes standard data types
and performs conversions

Distributed object management
Network

The infrastructure can transparently convert representations across platforms
Idea behind remote action invocation

Using a common intermediate form

Information appliances

Question

• What advantages and disadvantages do you see for the information appliance?
**Horizontal structure in layers**

<table>
<thead>
<tr>
<th>Application</th>
<th>Internet protocol</th>
<th>Network 1</th>
<th>Network 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows NT</td>
<td>TCP/UDP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mac OS</td>
<td>TCP/UDP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNIX</td>
<td>TCP/UDP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Spanning layer**

<table>
<thead>
<tr>
<th>Application</th>
<th>Distributed object management</th>
<th>Internet protocol</th>
<th>Network 1</th>
<th>Network 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows NT</td>
<td>Mac OS</td>
<td>TCP/UDP</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNIX</td>
<td>TCP/UDP</td>
<td>TCP/UDP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A spanning layer is ubiquitous and hides the layers below.

---

**Abstraction**

- A property of well-designed interfaces to modules
- Hide detail, displaying only what is necessary
- Simplify, displaying only what is meaningful to the outside
- Important for complexity management

---

**Encapsulation**

- Module implementation details (anything not explicit at interface) should be inaccessible from the outside
  - So other modules cannot become inadvertently dependent on implementation
  - In the case of components, for proprietary or security reasons

---

**Summary of modularity**

- Divide and conquer: decomposition of the system into modules with well-defined functional groupings
- Separation of concerns: great dependency internally, little dependency externally
- Abstraction: hide detail and simplify
- Encapsulation: make internal implementation inaccessible
- Reusability: meet generalized needs, configurable