建筑风格

鄂尔多斯市
北京交通大学

内容

- 建筑风格的定义
- 建筑风格

建筑风格的定义

建筑风格的定义

- 架构风格定义了一套规则，描述了其组件的属性和约束，以及组件之间的互动方式。
  - 建筑风格适用于高层设计；设计模式适用于详细设计
  - 风格是开放式的，所以会出现新的风格
  - 一个建筑通常只使用一种风格

建筑风格的好处

- 核心是可重复使用
  - 设计再利用
    - 同一领域的系统往往具有反映领域概念的类似架构
    - 应用产品系列是围绕着一个核心架构建立的，有满足客户特殊要求的变体。
  - 代码重复使用
  - 文件再利用

建筑风格的好处

- 提高开发效率和生产力
- 为额外的和新的设计理念提供了一个起点
- 有助于对设计进行权衡和预评估
- 减少了设计的风险
- 促进设计师之间的交流
  - 诸如 "客户-服务器 "这样的短语传达了很多信息

建筑风格

建筑风格

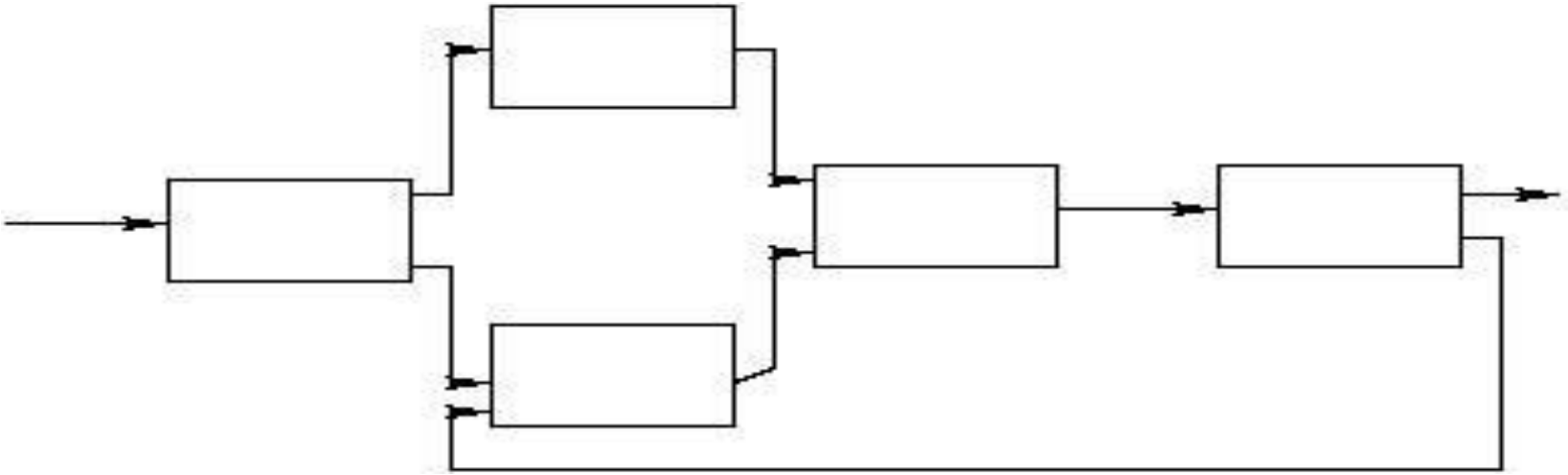
- 管道过滤器
- 存储库模型（共享数据存储）。
- 客户端-服务器
- 模型-视图-控制器(MVC)
- 分层系统
- 点对点

- 事件驱动
- 面向服务的架构（SOA）

管道过滤式

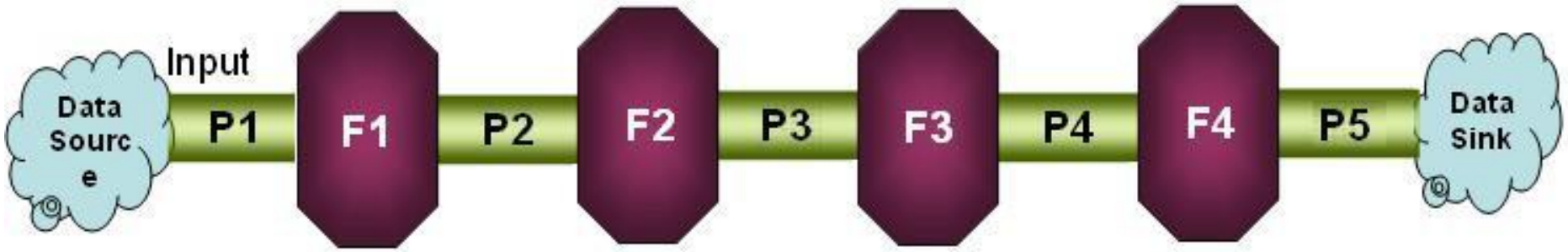
管道过滤器

- 一个流水线由一连串的处理元素组成，每个元素的输出是下一个元素的输入。通常，在连续的元素之间提供一定量的缓冲。



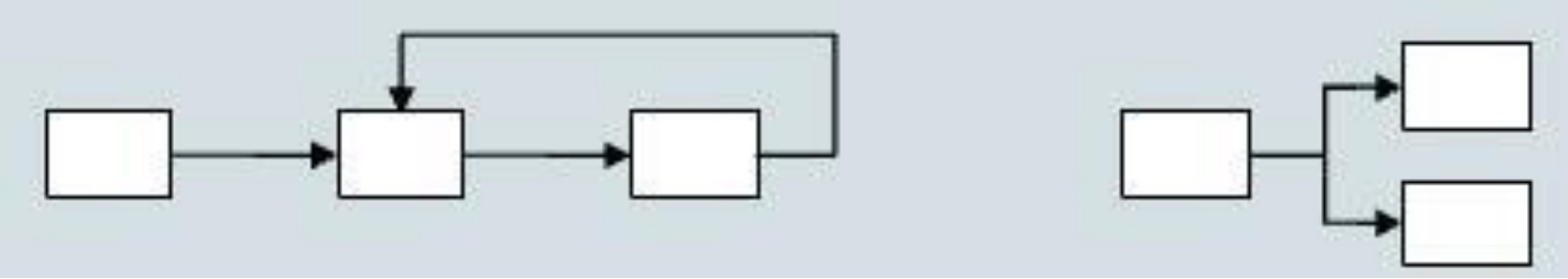
管道过滤器

- 组件：用于数据处理的过滤器
  - 读取其输入的数据流，并在其输出上产生一个数据流
  - 相互独立
- 连接器：用于数据转换和运输的管道
  - 将过滤器产生的输出传输给其他过滤器



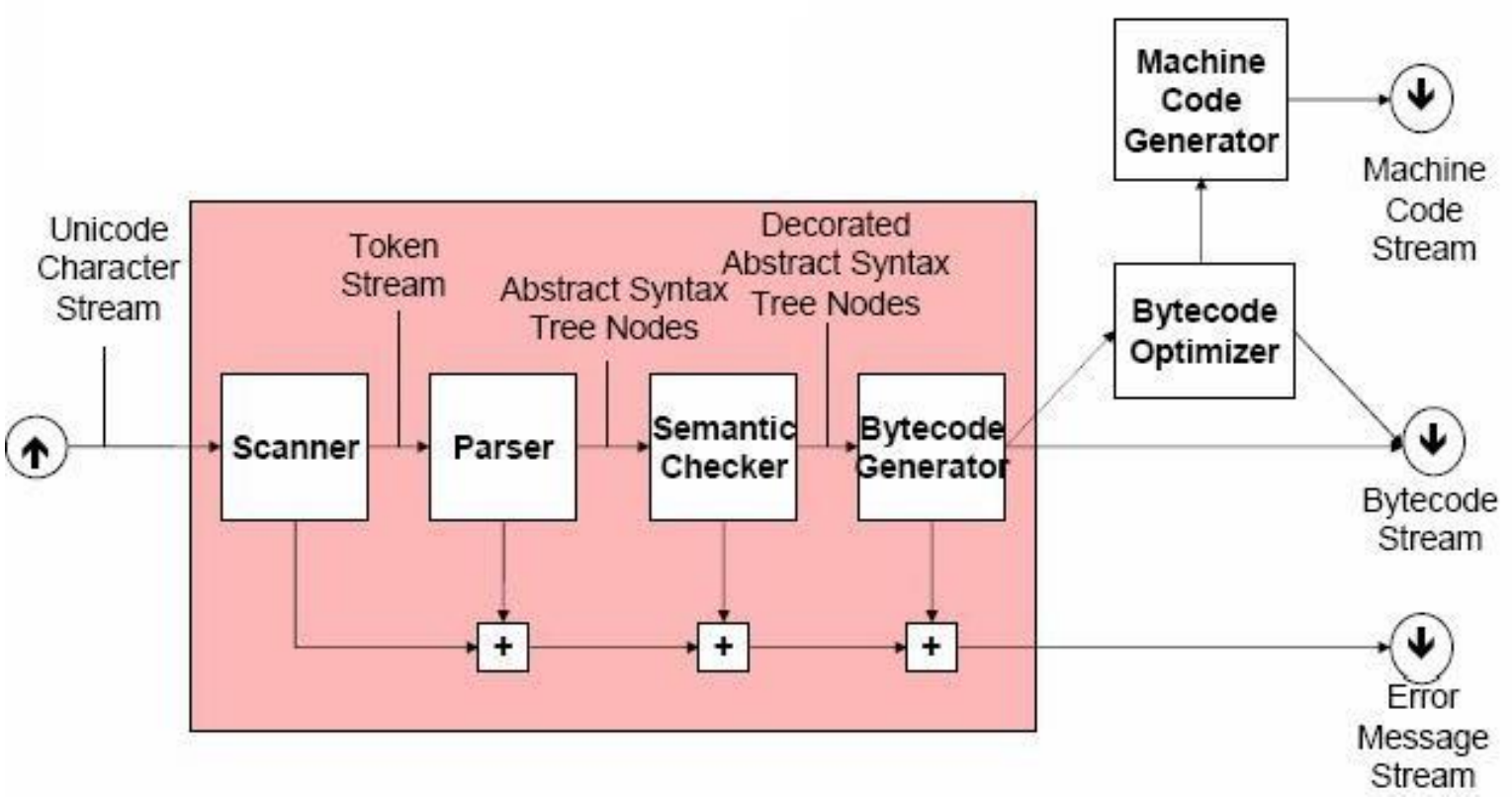
管道过滤器

- 拓扑结构：带有反馈回路或分管的线性结构



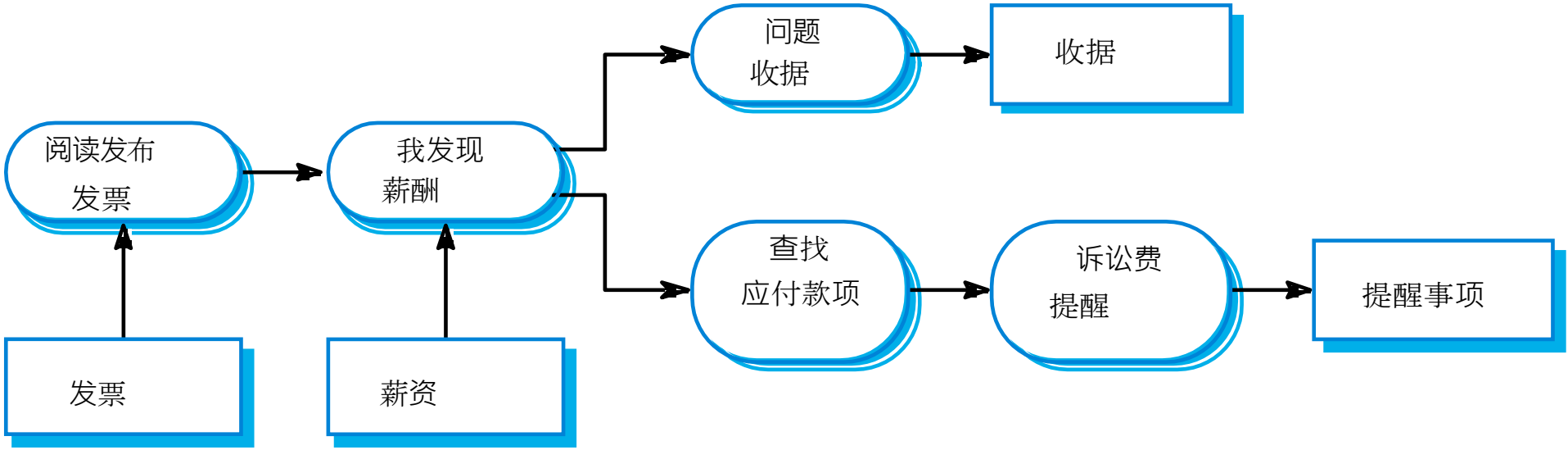
例子 13

- 编译器


Machine Code Generator
Machine Code Stream
Unicode Character Stream
Token Stream
Abstract Syntax Tree Nodes
Decorated Abstract Syntax Tree Nodes
Bytecode Optimizer
Scanner
Parser
Semantic Checker
Bytecode Generator
Bytecode Stream
Error Message Stream

例子 14

- 发票处理系统



优势

- 高内聚性：过滤器是自我包含的处理服务，执行特定的功能，因此它的内聚性相当高。
- 低耦合性：过滤器仅通过管道进行通信，因此在耦合性方面受到 "一定程度 "的限制
- 简单性：过滤器可以很容易地用于设计一个并发的或顺序的系统
- 可重用性：现有的过滤器可以重用，无需考虑其他过滤器
- 灵活性：现有的过滤器可以很容易地被重新定义和/或重新路由
- 可扩展性：在不考虑其他过滤器的情况下，可以

优势

很容易地添加新的过滤器

劣势

- 需要一个共同的格式来沿管道传输数据
- 难以将各种输入分配给相应的过滤器（在支持基于事件的交互方面）。

存储库模型风格

存储库模型

- 子系统之间的数据交换可以通过两种方式进行
  - 共享数据保存在一个中央数据库或存储库中，所有子系统都可以访问。
  - 每个子系统维护自己的数据库，并将数据明确地传递给其他子系统
- 当需要共享大量数据时，最常使用的是第

存储库模型

一种选择

存储库模型

- 组件：数据存储和处理数据的子系统
- 连接器：互动协议
  - 所有子系统都在一个单一的数据存储上工作
  - 对数据存储的任何改变都可能影响所有或部分子系统
- 病人处理系统、税务处理系统、库存

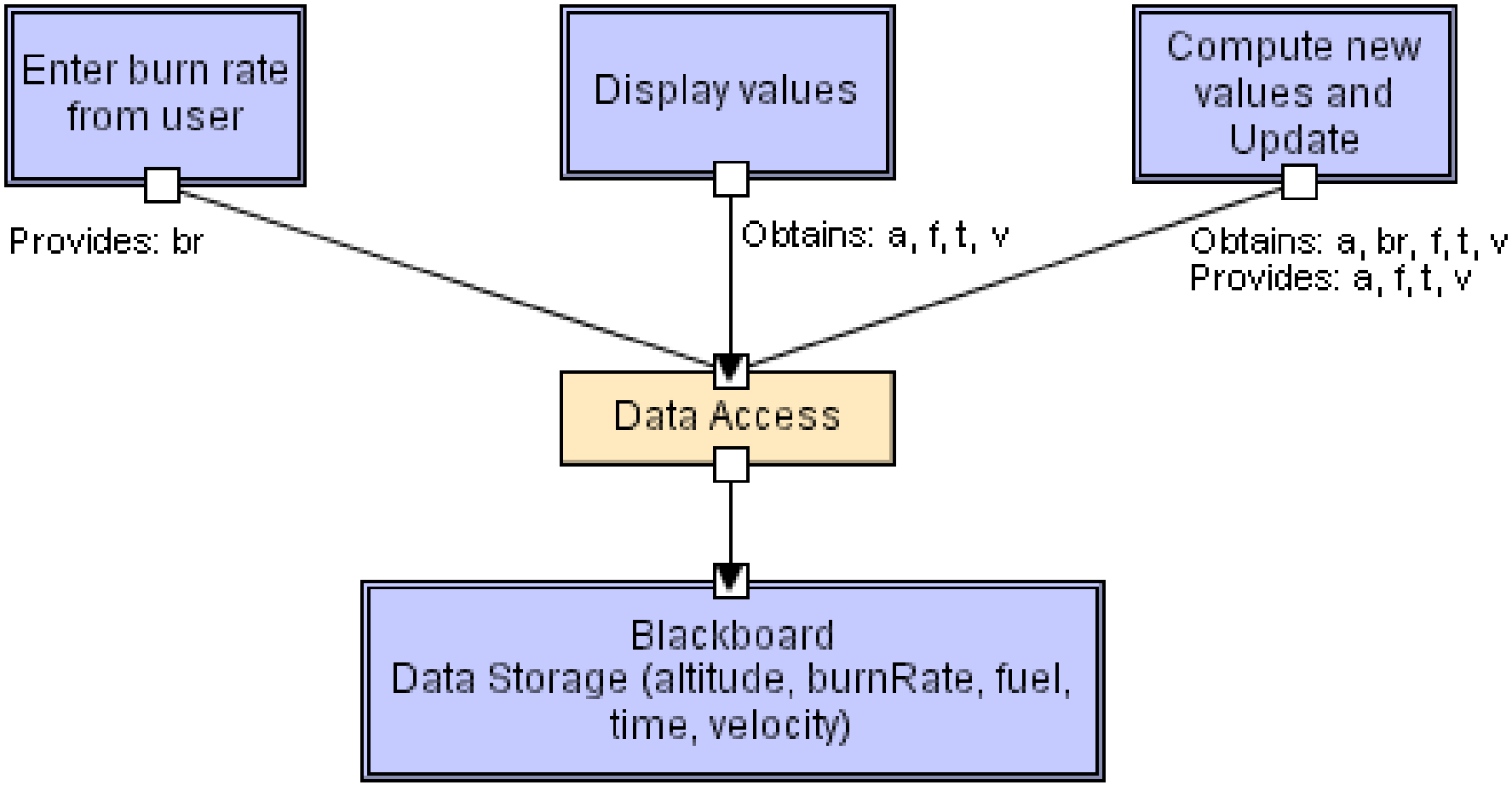
存储库模型

控制系统；等等。

两种变化

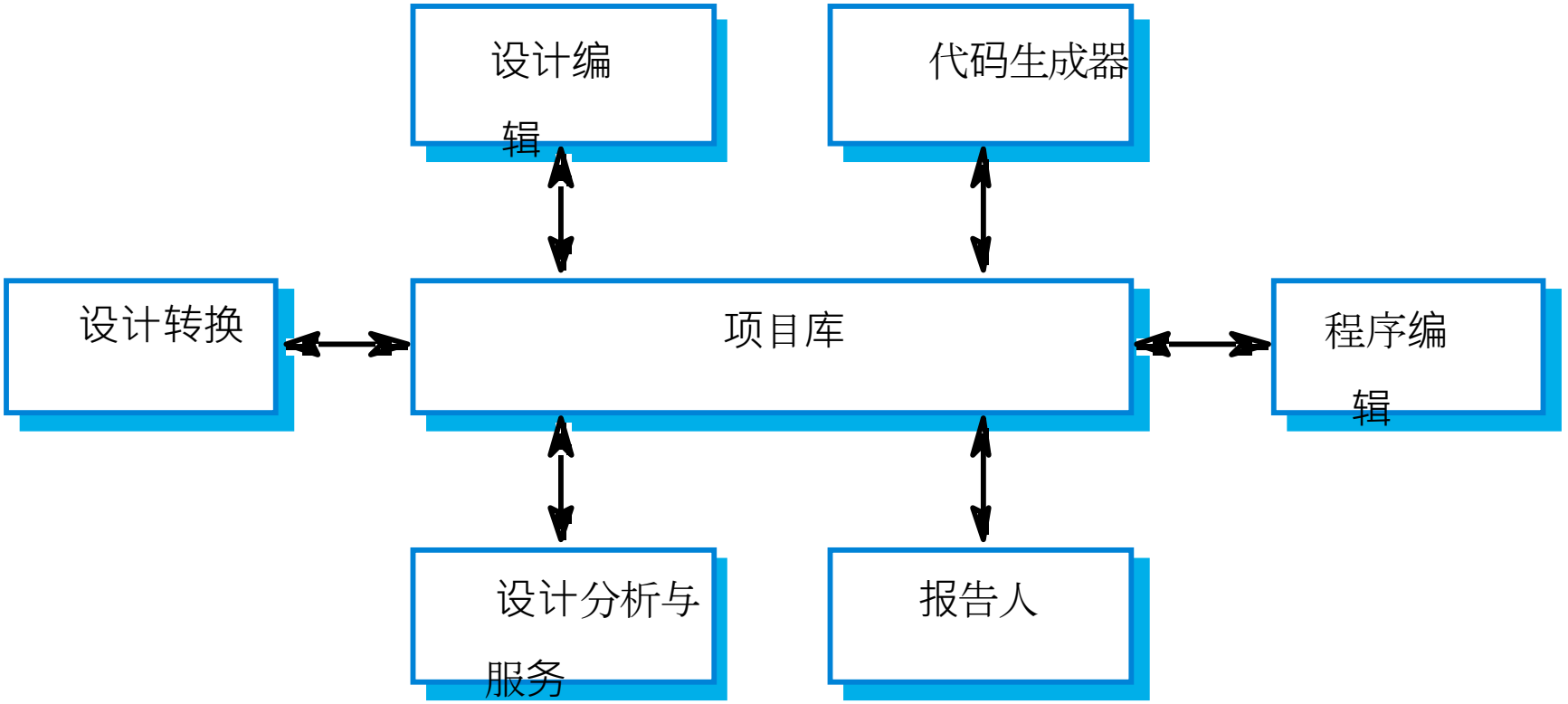
- 黑板风格：每当有变化的事件发生时，数据存储会提醒参与的子系统
- 存储库风格：参与的子系统检查数据存储的变化

黑板风格。例子


Enter burn rate from user
Display values
Compute new values and Update
Provides: br
Obtains: a, f, t, v
Obtains: a, br, f, t, v
Provides: a, f, t, v
Data Access
Blackboard
Data Storage (altitude, burnRate, fuel, time, velocity)

存储库风格。例子

- 计算机辅助软件工程（CASE）工具集架构



优势

- 独立的子系统在其内部是有凝聚力的，耦合仅限于共享数据。
- 单一的数据存储使得数据的共享变得高效。

劣势

- 对共享数据的任何改变都需要达成协议，并有可能改变所有或部分子系统。
  - 数据演变是困难和昂贵的
- 如果数据存储发生故障，所有的子系统都会受到影响，可能不得不停止。
  - 需要有冗余的数据库和/或良好的备份和恢复程序

客户端-服务器风格

客户端-服务器

- 组件：客户端和服务器
  - 客户端：提出请求（向服务器）并处理与系统环境的输入/输出的子系统
  - 服务器：响应客户请求的子系统
- 连接器：网络交互协议
  - 客户端依赖服务器
  - 客户端知道服务器的身份，而服务器不知道客

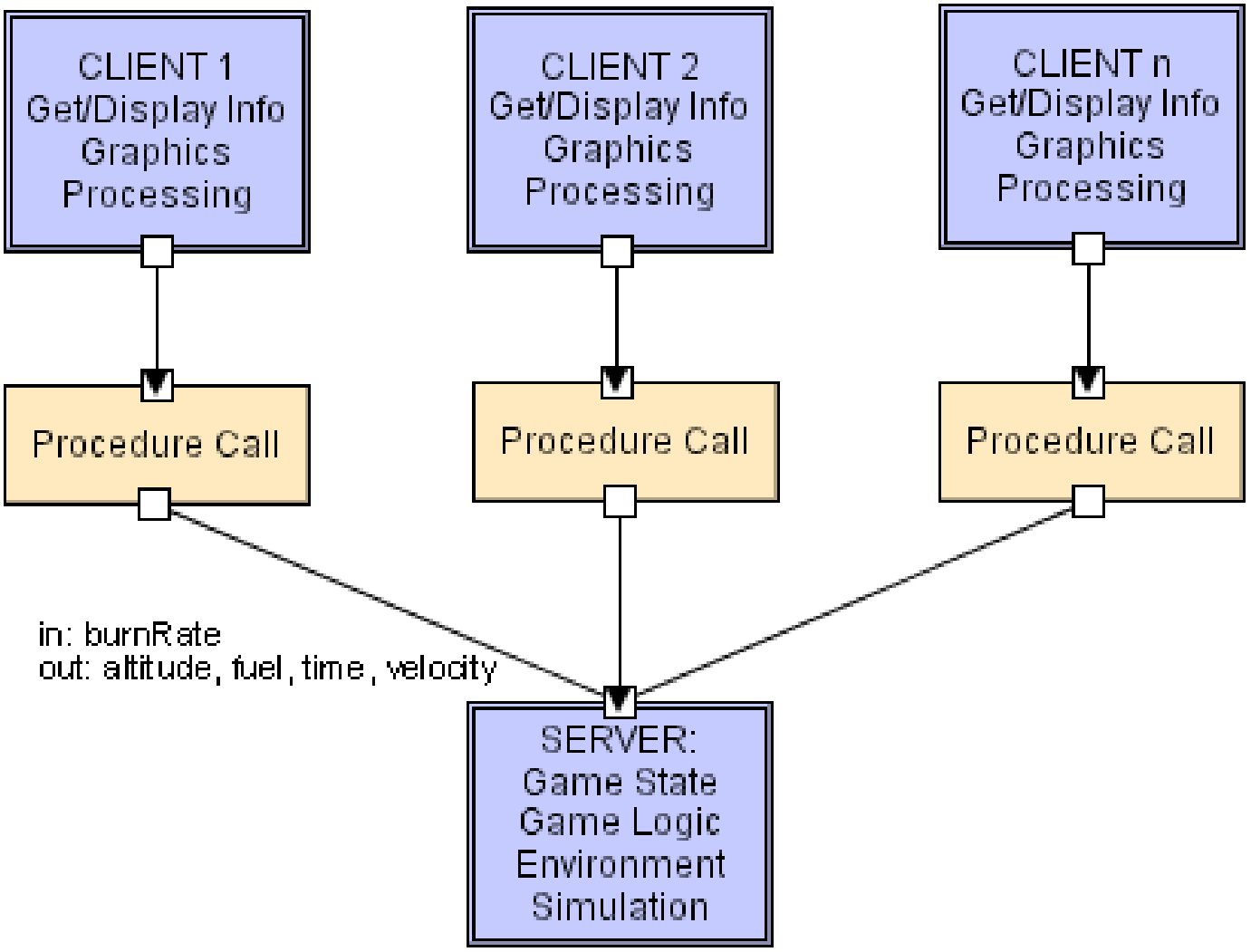
客户端-服务器

户的数量或身份

客户端-服务器

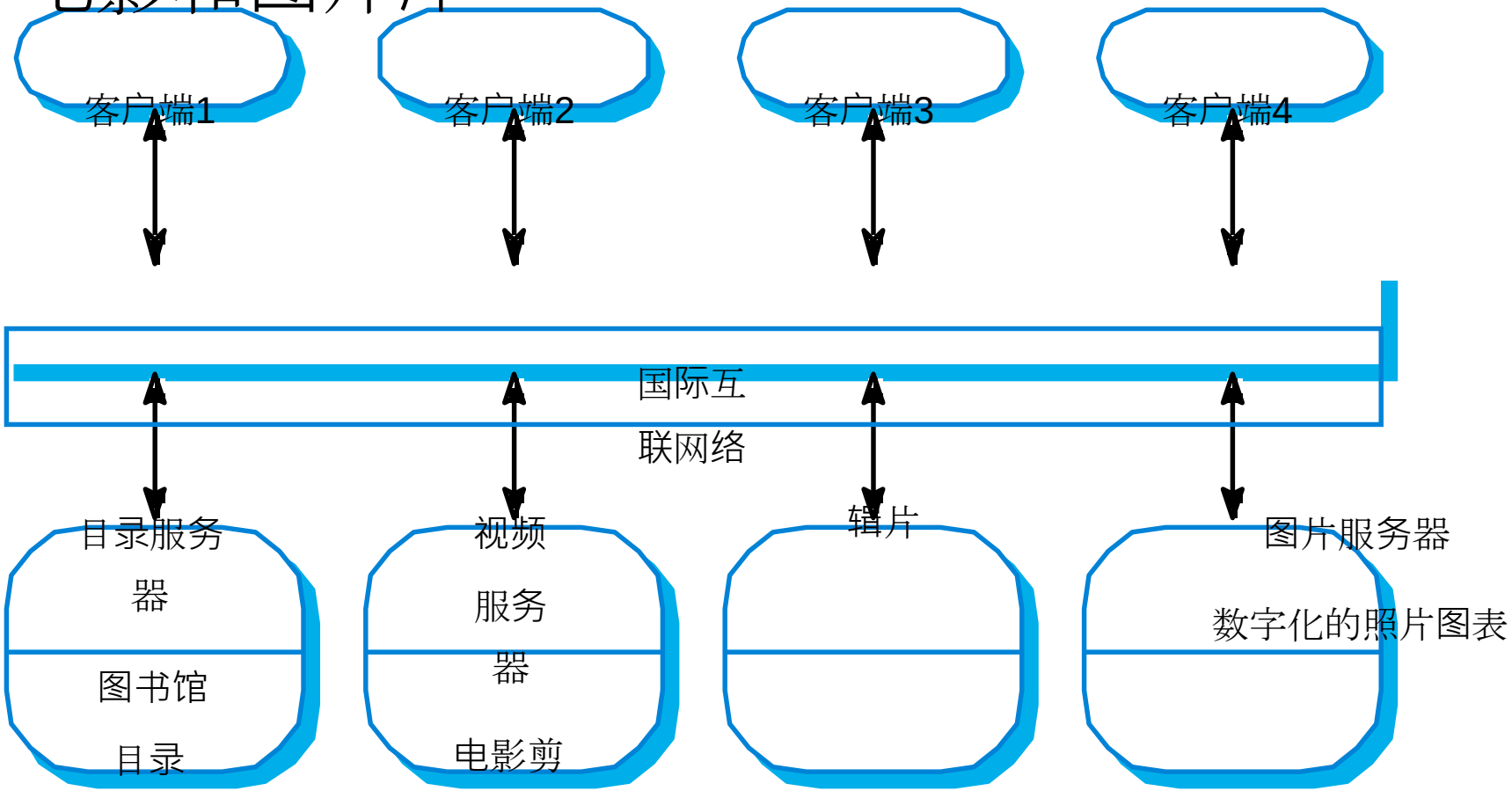
- 拓扑结构：许多客户对许多服务器
  - 客户之间没有联系
- 同步性：同步或异步
- 安全性：通常由服务器控制

例子 29


CLIENT 1
Get/Display Info
Graphics
Processing
CLIENT 2
Get/Display Info
Graphics
Processing
CLIENT n
Get/Display Info
Graphics
Processing
Procedure Call
Procedure Call
Procedure Call
in: burnRate
out: altitude, fuel, time, velocity
SERVER:
Game State
Game Logic
Environment
Simulation

例子 30

- 电影和图片库


客户端1
客户端2
客户端3
客户端4
国际互
联网络
目录服务
器
图书馆
目录
视频
服务
器
电影剪
辑片
图片服务器
数字化的照片图表

例子 31

网络　　服务器

电影和照片资料。

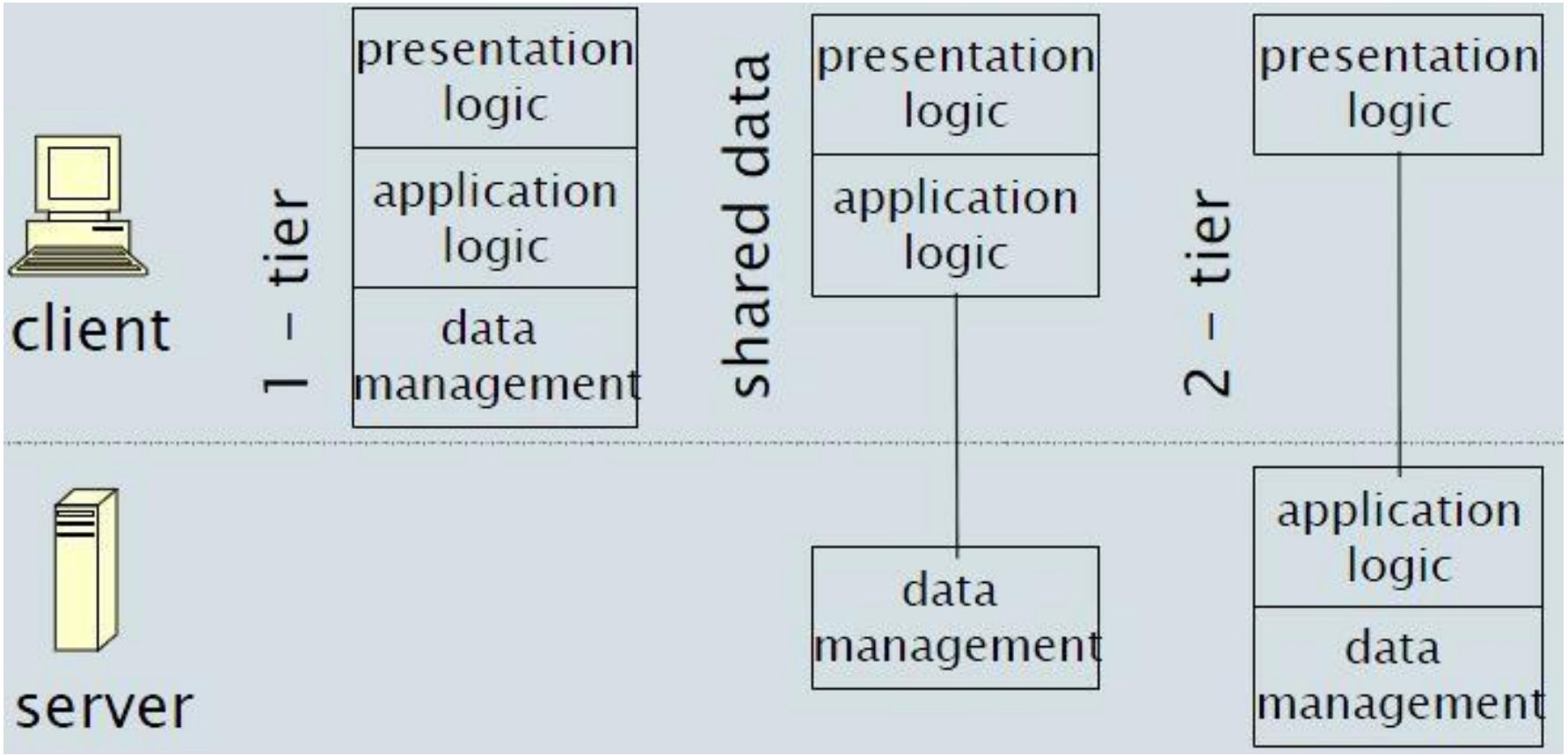
两种变化

- 两层楼
  - 瘦客户机
  - 厚重的客户端
- 三层楼
  - 瘦客户机
  - 厚重的客户端

两层的客户-服务器

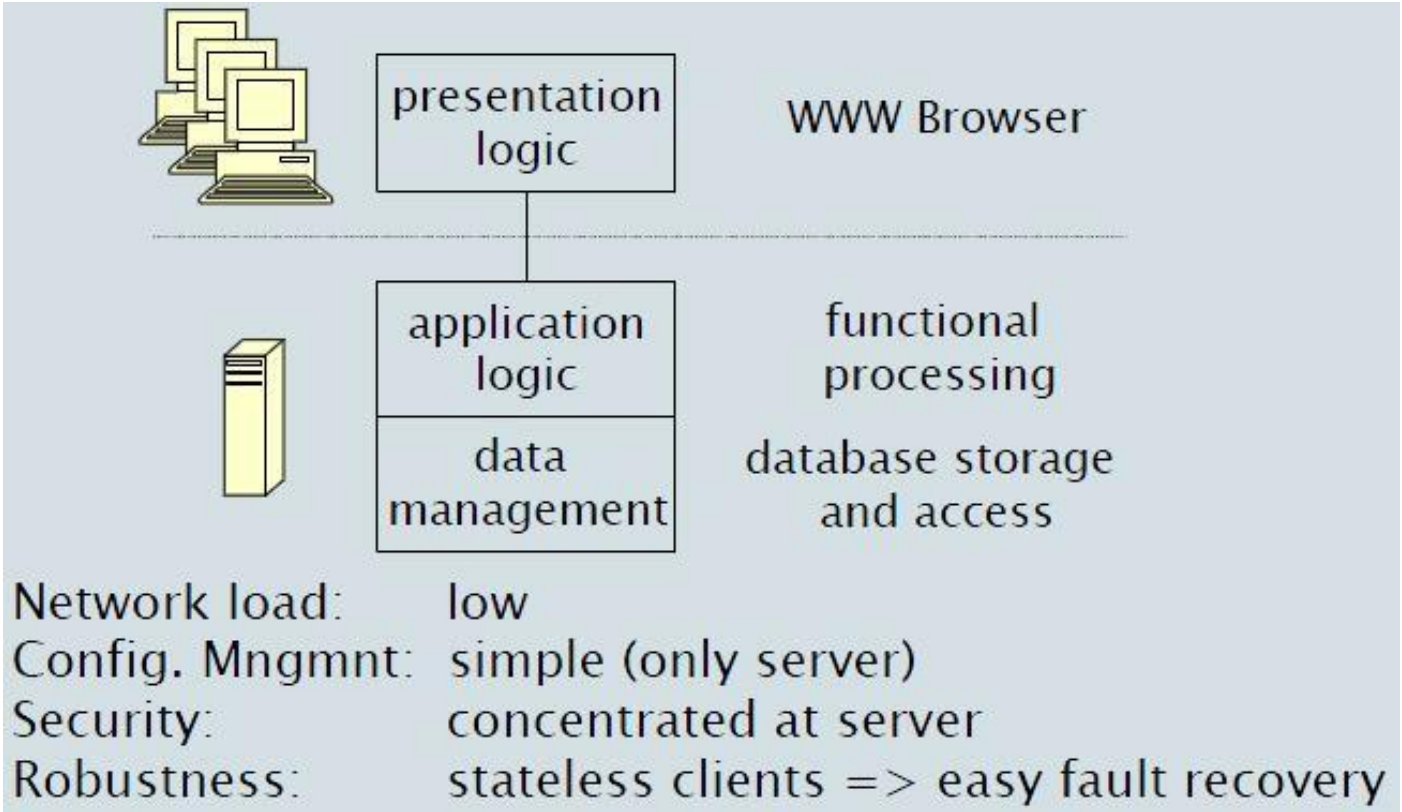
- 第1层：客户中本地计算机环境的用户界面
- 第2层：服务器中的应用和数据库管理

两层的客户-服务器

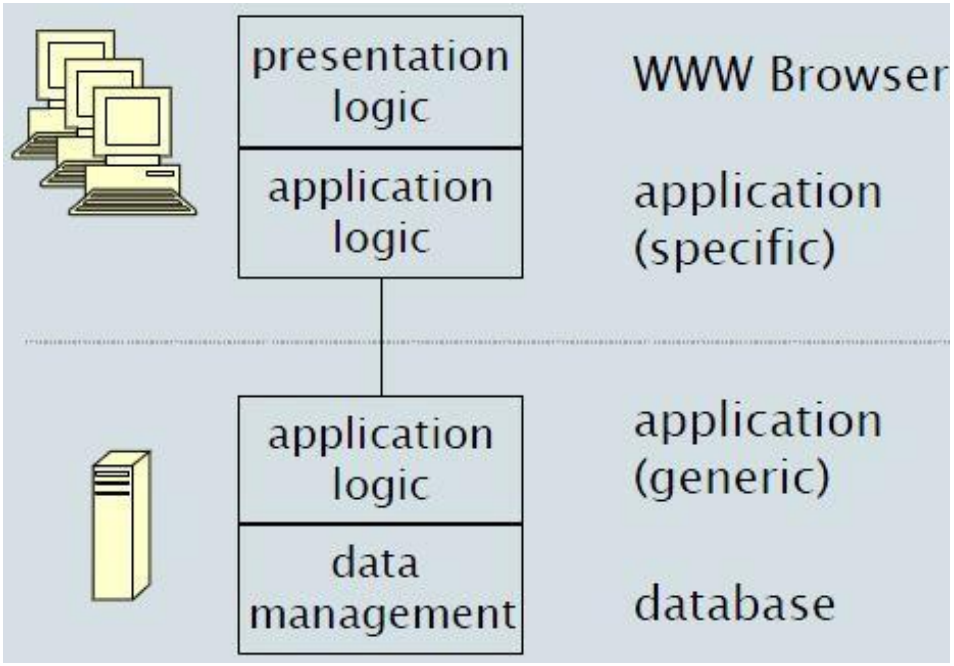

client
server
1 – tier
presentation logic
application logic
data management
shared data
presentation logic
application logic
data management
2 – tier
presentation logic
application logic
data management

两层的客户-服务器。瘦客户机

- 在服务器端处理的最大部分



两层的客户-服务器。厚客户机



- 在客户端进行大量的处理

网络负荷：高

两层的客户-服务器。厚客户机

配置。管理：复杂（包括客户端和服务器） 安全：

复杂（包括客户端和服务器）

稳健性：客户有状态=>复杂的故障恢复

两层的客户-服务器。优势

- 有效利用网络资源
- 易于添加新客户/服务器或升级现有客户/服务器
- 允许多个客户端之间共享数据

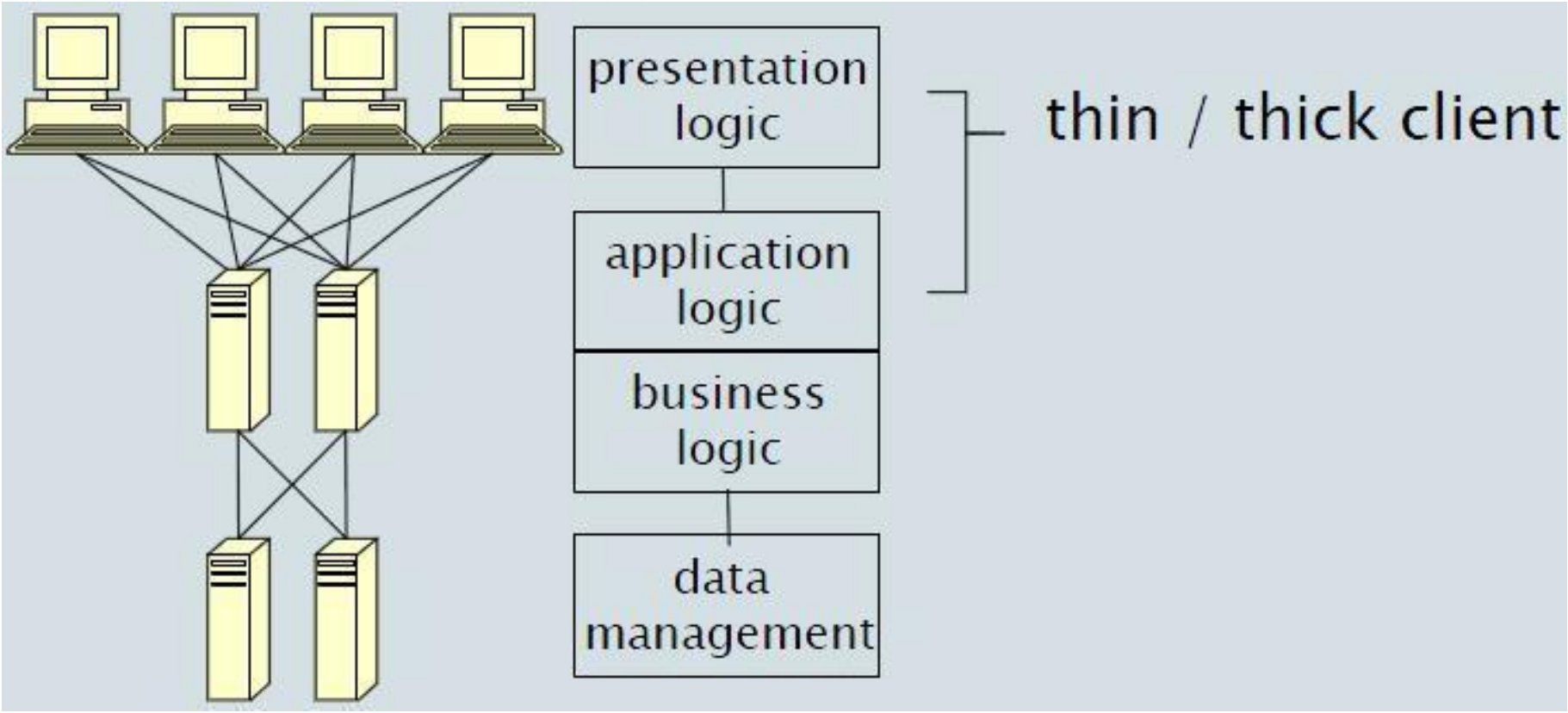
两层的客户-服务器。劣势

- 需要高质量的硬件和每个服务器的冗余管理
- 很难发现有哪些服务器和服务可用
  - 没有名称和服务的中央登记
- 性能和可扩展性受到服务器和网络容量的限制

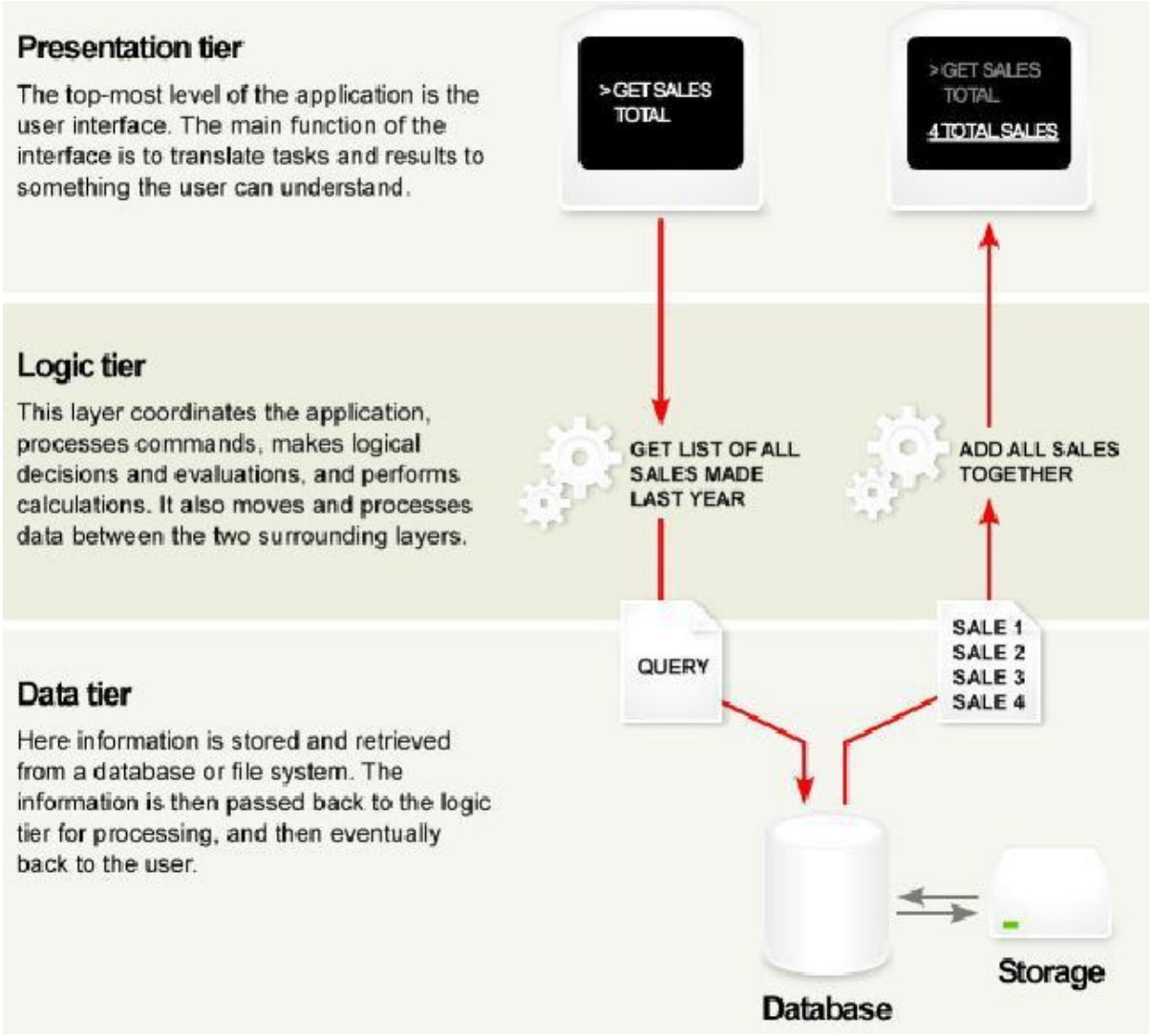
三层客户-服务器

- 演示层：负责显示用户界面
- 数据访问层。负责检索、修改和删除数据，并将结果发送到演示层。
- 业务逻辑层。负责处理检索到的数据并发送至演示层

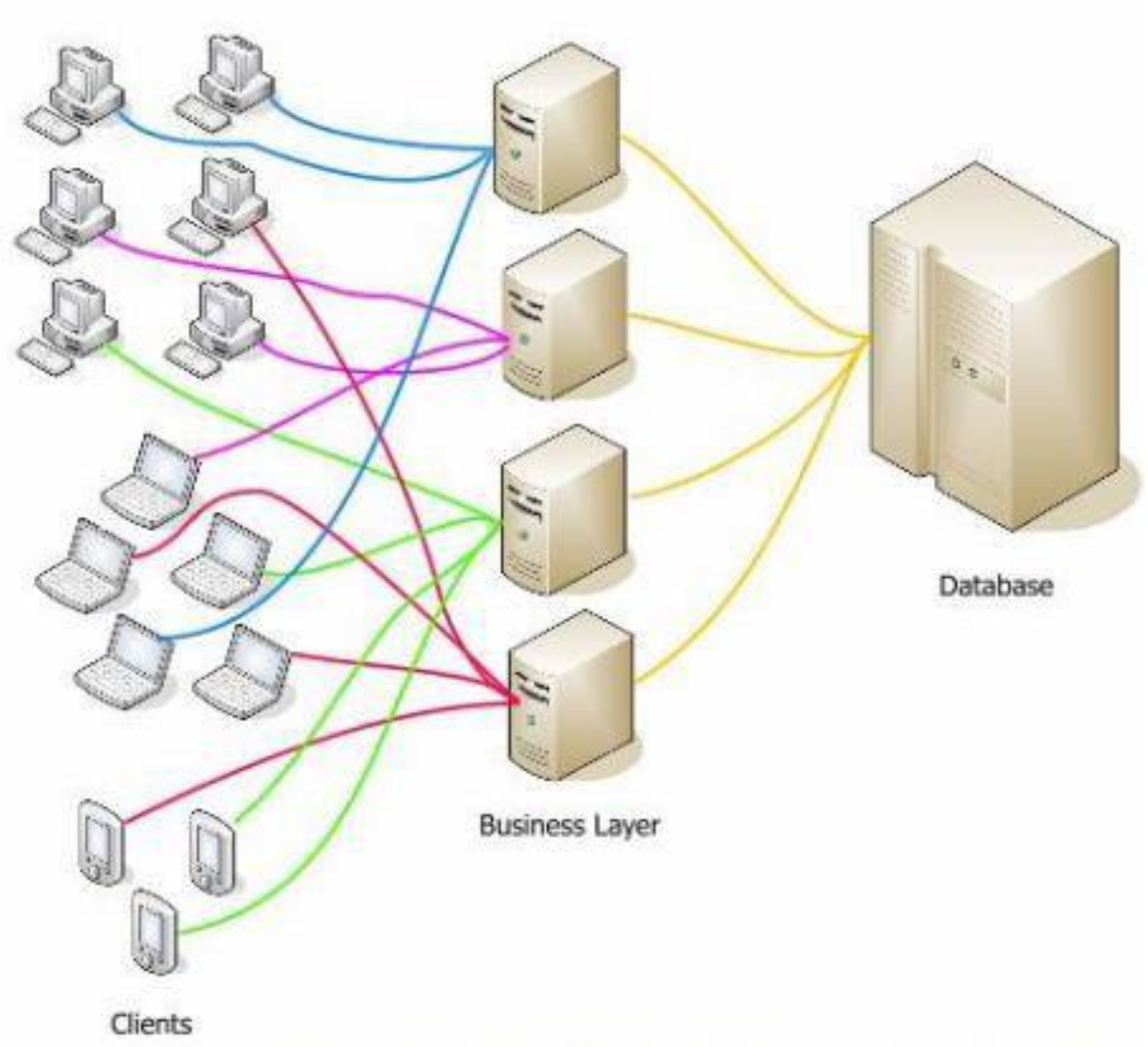
三层客户-服务器


presentation logic
application logic
business logic
data management
thin / thick client

三层客户-服务器。例子


Presentation tier
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.
>GET SALES TOTAL
>GET SALES TOTAL
4 TOTAL SALES
Logic tier
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.
GET LIST OF ALL SALES MADE LAST YEAR
ADD ALL SALES TOGETHER
Data tier
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.
QUERY
SALE 1
SALE 2
SALE 3
SALE 4
Storage
Database

三层客户-服务器。部署



三层客户-服务器。优势

- 更好的性能和可扩展性
- 安全措施可以集中分配
- 不同层级的平行发展

三层的客户-服务器。劣势

- 双层客户-服务器风格也是如此

模型-视图-控制器(MVC)风格

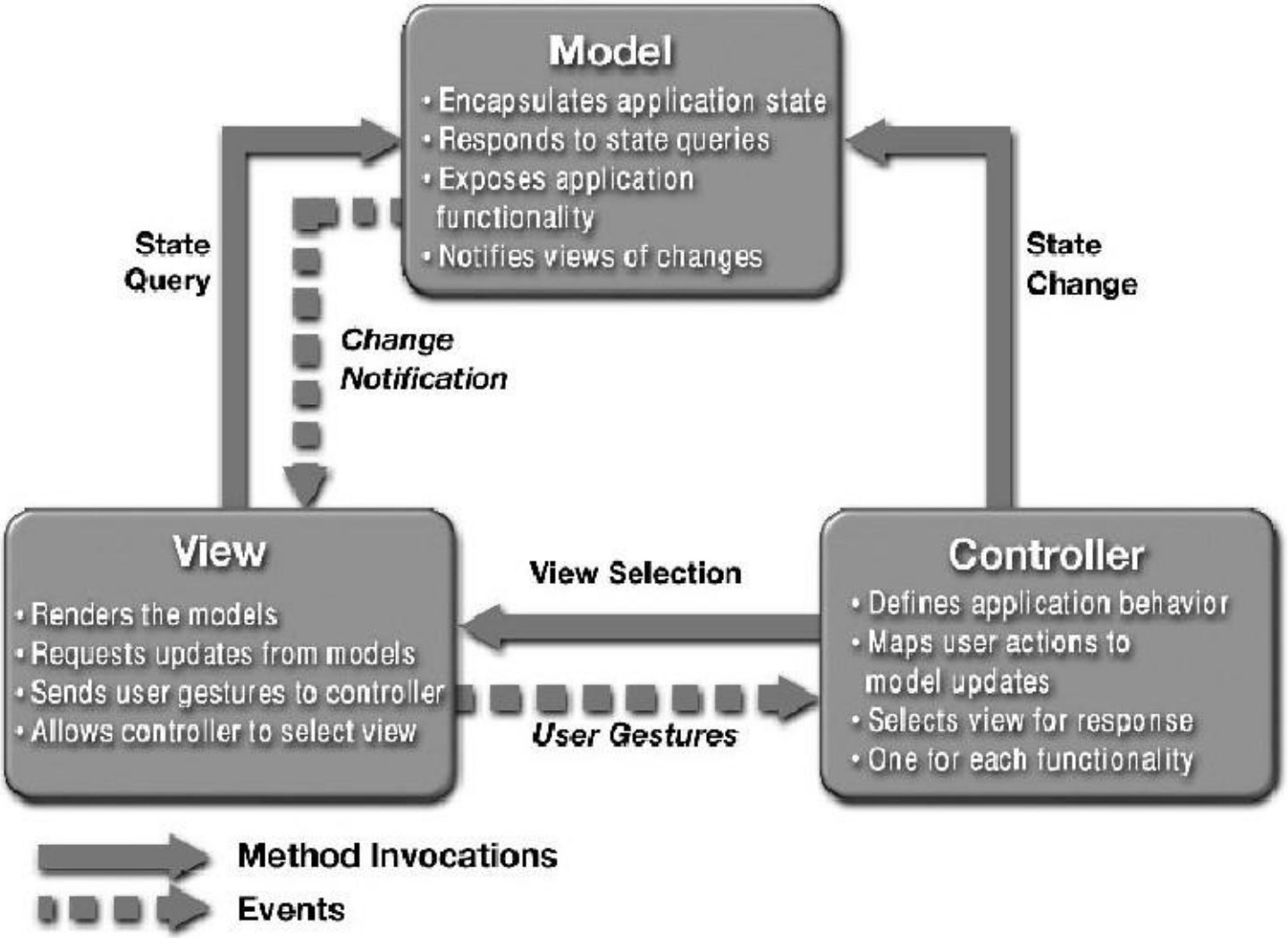
模型-视图-控制器(MVC)

演示者
2016-07-19 16:51:30

表现层=视图+控制器；业务逻辑层+数据访问层=模型

- 组件：模型、视图和控制器
  - 将领域的建模、展示和基于用户输入的行动分离成三个独立的组别
- 连接器：互动协议

模型-视图-控制器(MVC)


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

优势

- 视图、控制器和模型是独立的组件，允许在不明显干扰其他组件的情况下进行修改。
  - 即使模型停机，视图和控制器也能继续部分运作

劣势

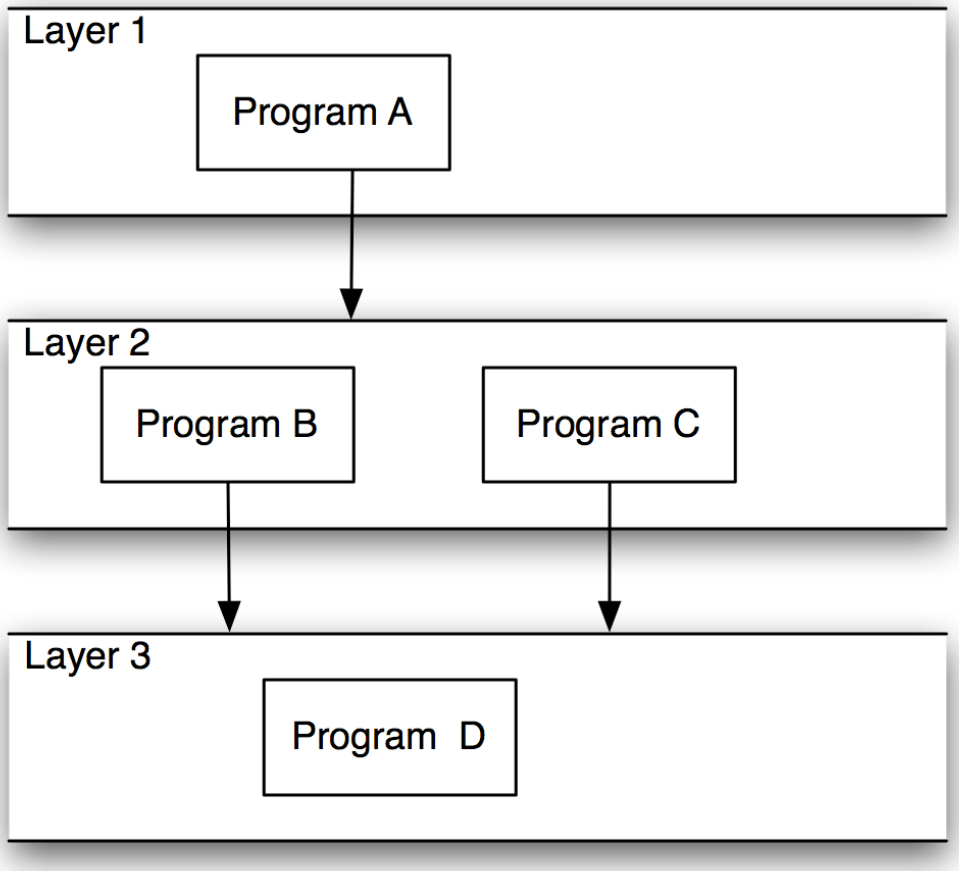
- 严重依赖开发和生产系统环境以及与MVC架构相匹配的工具（如TomCat、.Net、Rail等）。

分层系统风格

分层系统

- 组成部分：层
  - 每一层都提供一套相关的服务
- 连接器：各层之间的互动协议
  - 每一层只能使用它下面的一层（几层）。

分层系统


Layer 1
Program A
Layer 2
Program B
Program C
Layer 3
Program D

两种变化

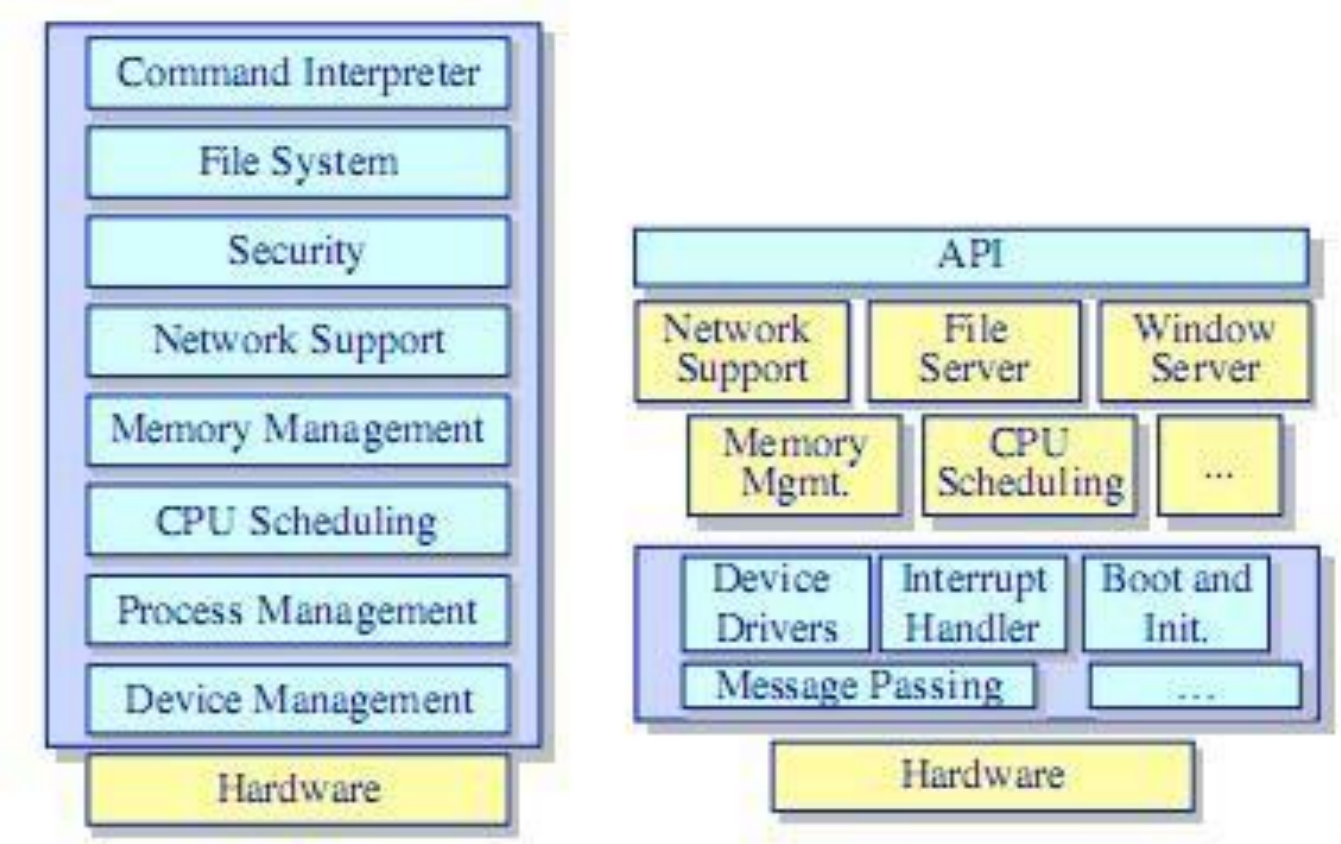
- 严格的分层系统风格：如果任何一层只使用其正下方的一层
- 宽松的层系统风格：如果一个层可以使用它下面的任何层

"多层次客户-服务器"

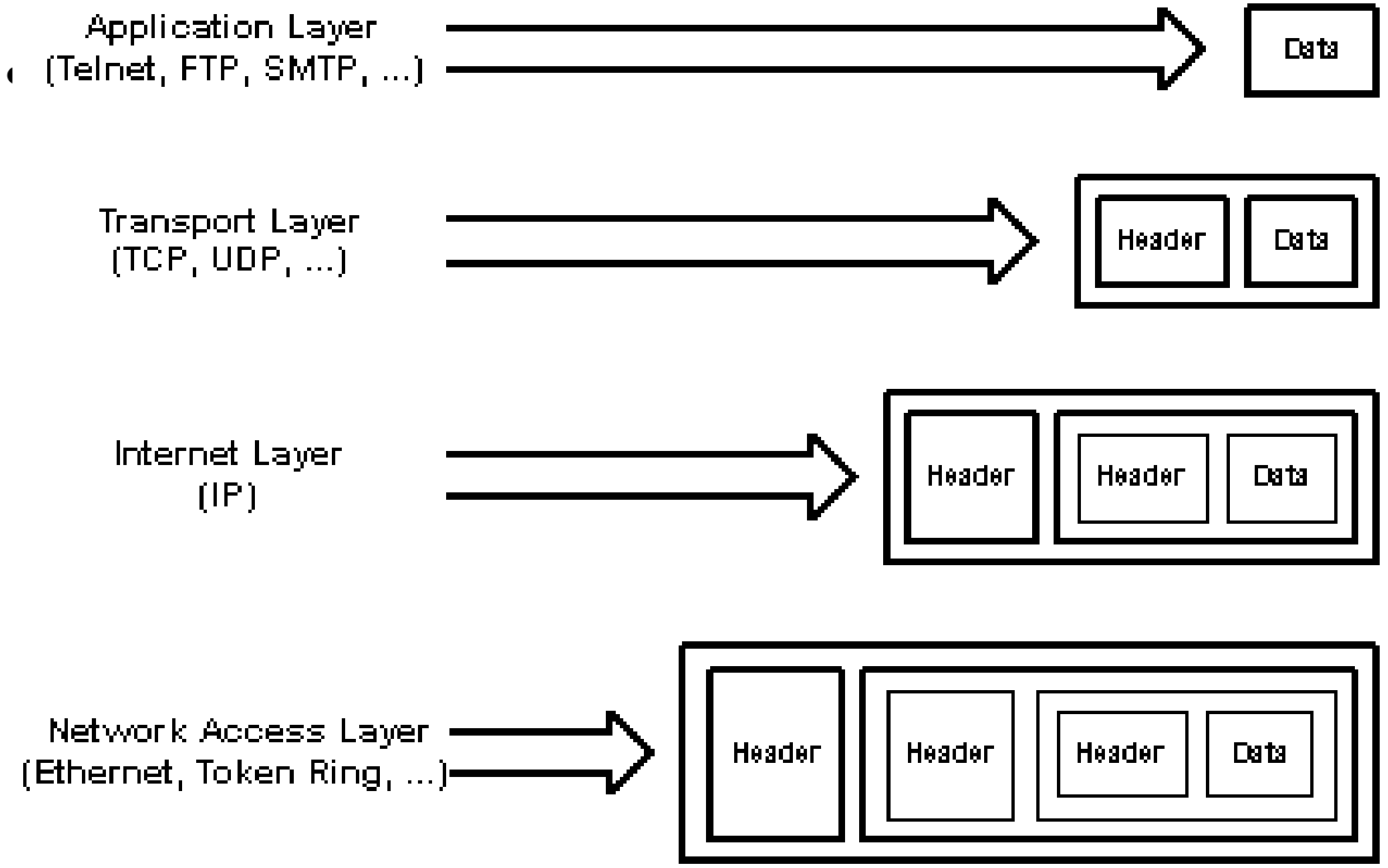
- 每一层都作为一个
  - 服务器：为 "上面 "的层提供服务
  - 客户：" 下面 "的层的服务消费者

例子 53

- 单片机操作系统和微内核操作系统



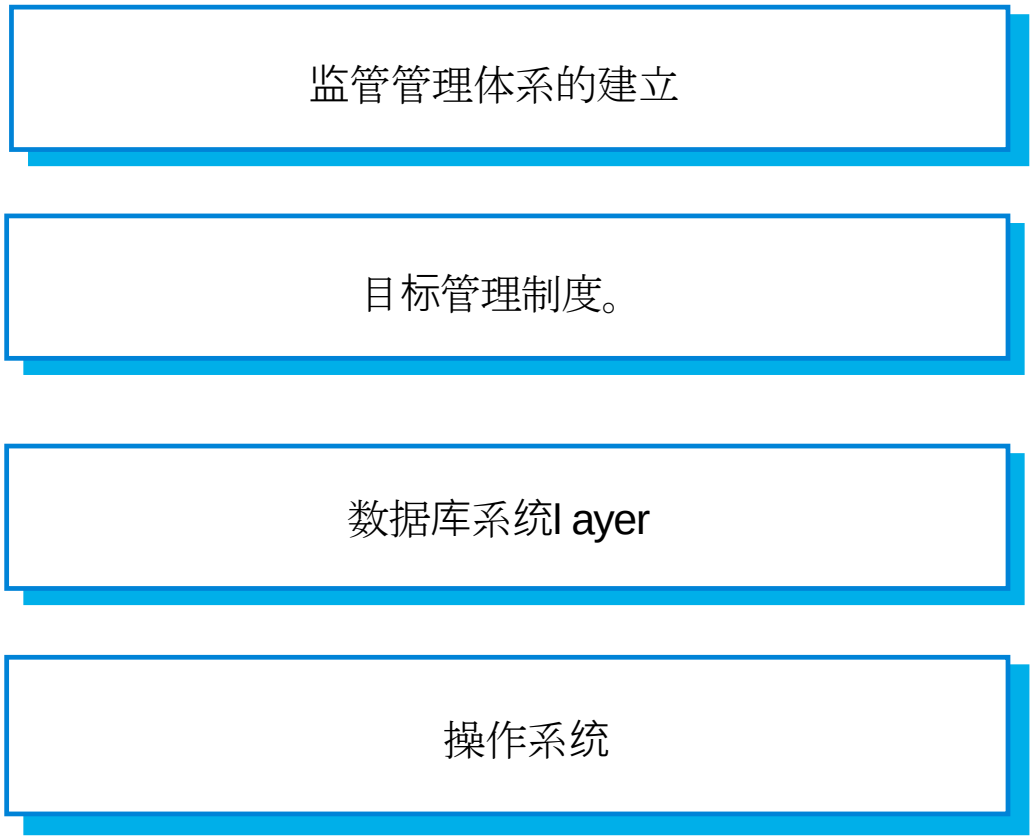
例子 LIV


Application Layer
(Telnet, FTP, SMTP, ...)
Data
Transport Layer
(TCP, UDP, ...)
Header
Data
Internet Layer
(IP)
Header
Header
Data
Network Access Layer
(Ethernet, Token Ring, ...)
Header
Header
Header
Data

例子 LV

例子 56

- 版本管理制度



优势

- 每一层都被选为一组相关的服务，所以它在层内提供了高度的内聚力。
- 层可以只使用较低的层，所以它制约了耦合的数量
- 每一层都是内聚的，并且只与较低的层耦合，使得添加、修改和/或重复使用一个层

优势

更加容易。

劣势

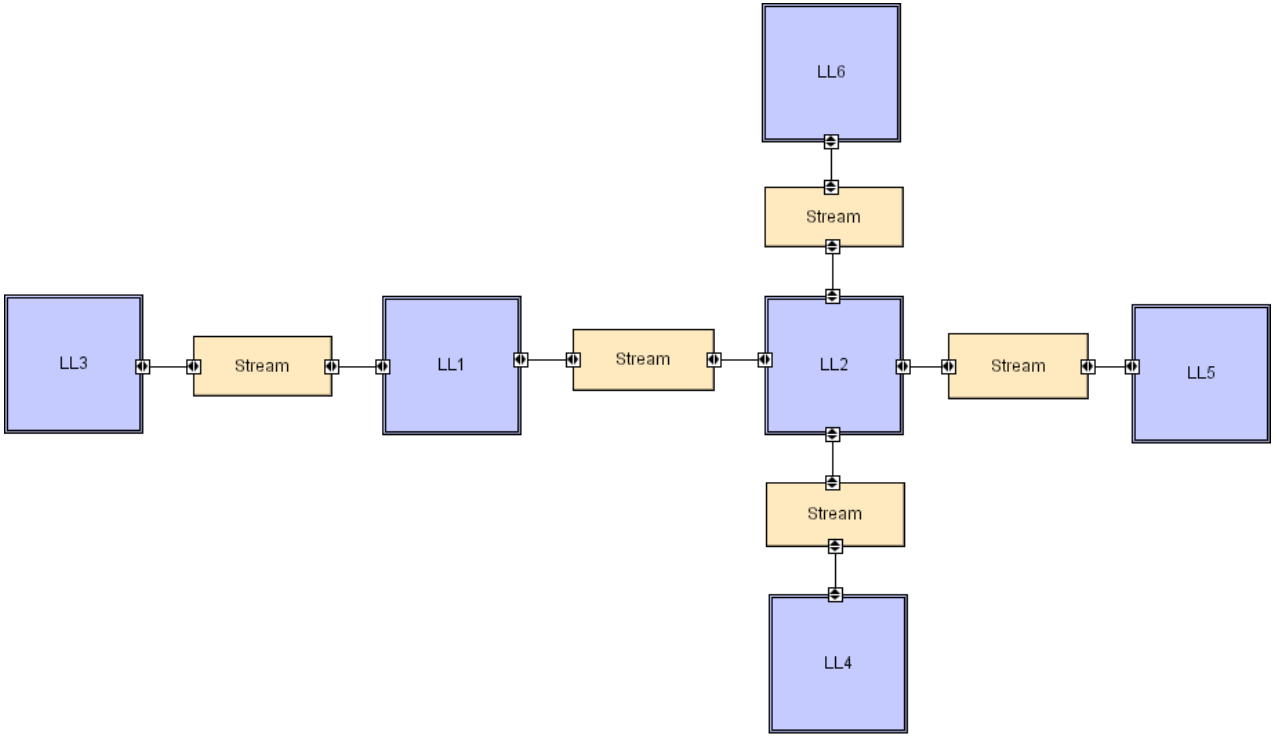
- 严格的分层风格可能会导致性能问题，这取决于层的数量。
- 要设计出干净的层次并不容易

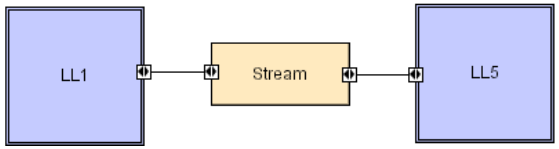
点对点风格

点对点

- 组成部分：同行
  - 每个对等体都维护自己的数据存储，以及其他对等体地址的动态路由表。
  - 同龄人之间没有区别
- 连接器：网络协议
  - 每个对等体可以同时充当服务器和客户端
- BitTorrent

例子


LL6
Stream
LL3
Stream
LL1
Stream
LL2
Stream
LL5
Stream
LL4


LL1
Stream
LL5

事件驱动的风格

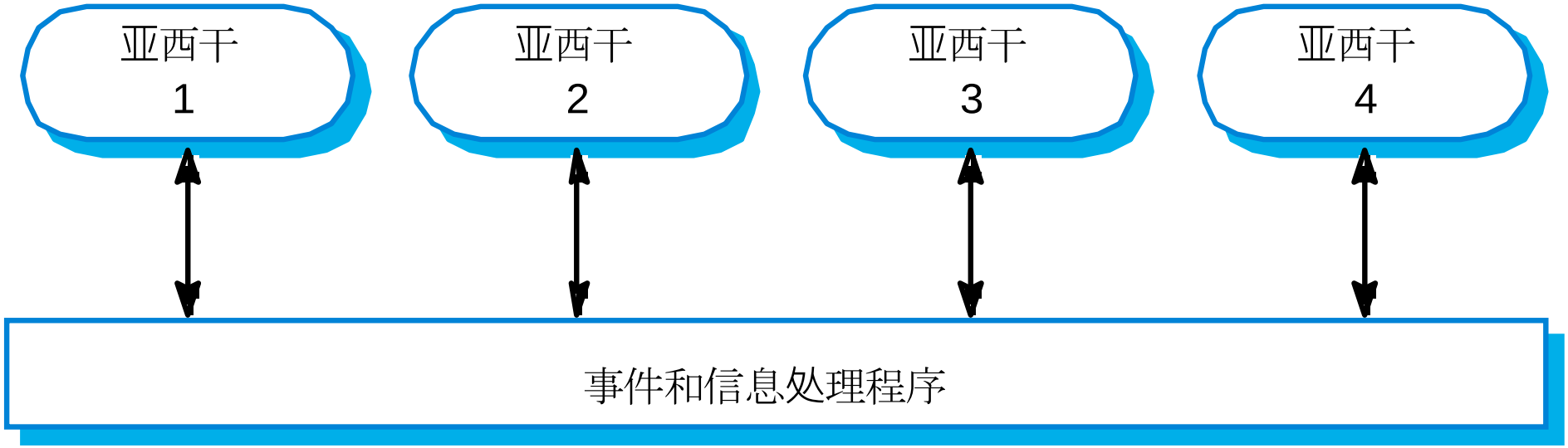
事件驱动

- 组件：事件调度器和处理器
  - 事件可能是一个简单的通知，也可能包括相关的数据
  - 事件可以通过组件进行 "注册 "或 "未注册"。
- 连接器：互动协议
  - 事件可以根据时间等限制因素来确定优先次序。
  - 事件可能需要同步或异步处理
- 通常对实时系统有用，如：飞机控制；医疗设备监控；家庭监控；嵌入式设备控制器；游戏等。

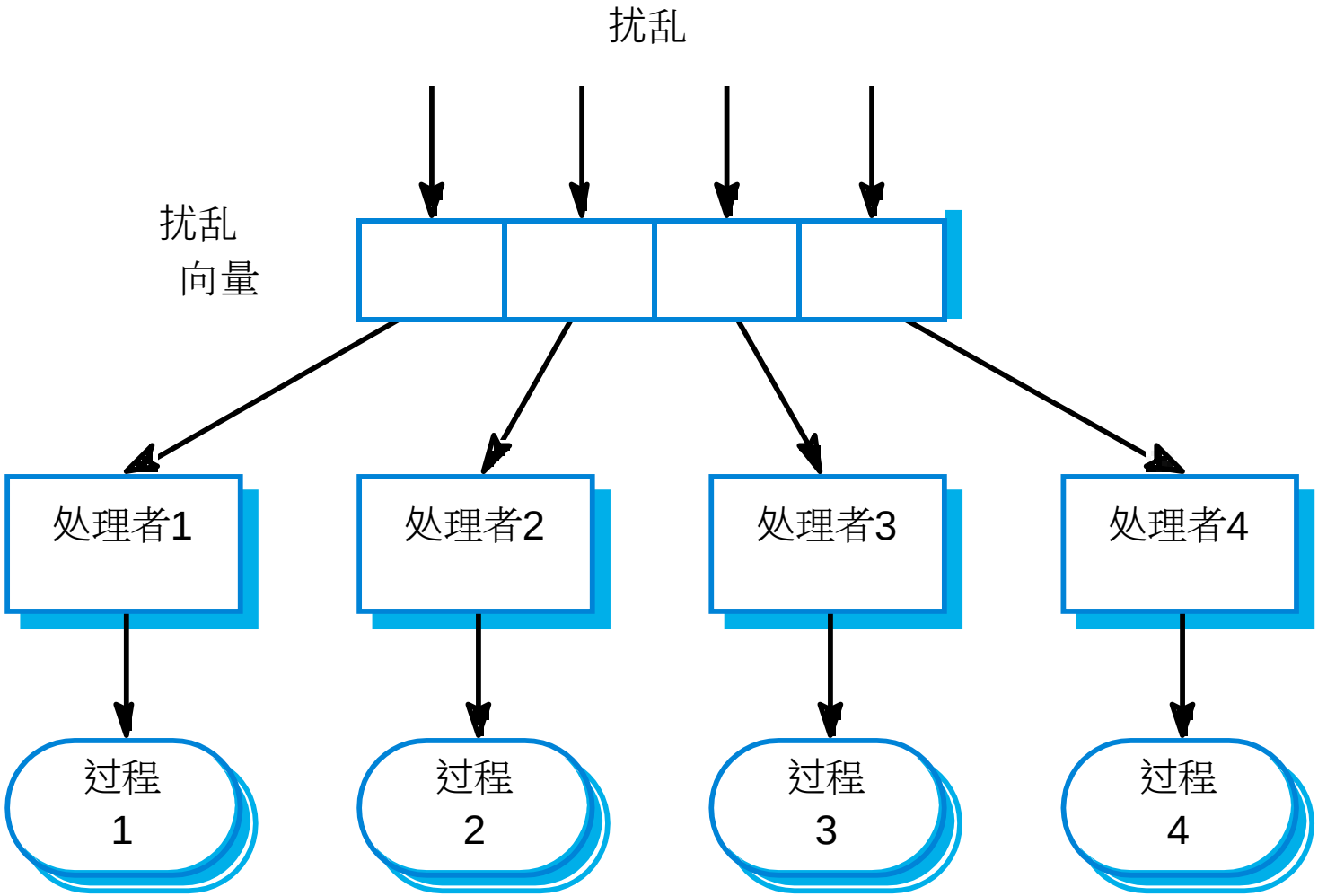
两种变化

- 广播式：一个事件被派发给所有处理器
- 中断驱动的风格：中断被发送到事件分配器（中断处理程序），并传递给一些处理器

广播。例子



中断驱动。例子



优势

- 事件调度器和事件处理器是分开的，保证了低解耦性。
- 任何处理器的故障都不会影响其他处理器

劣势

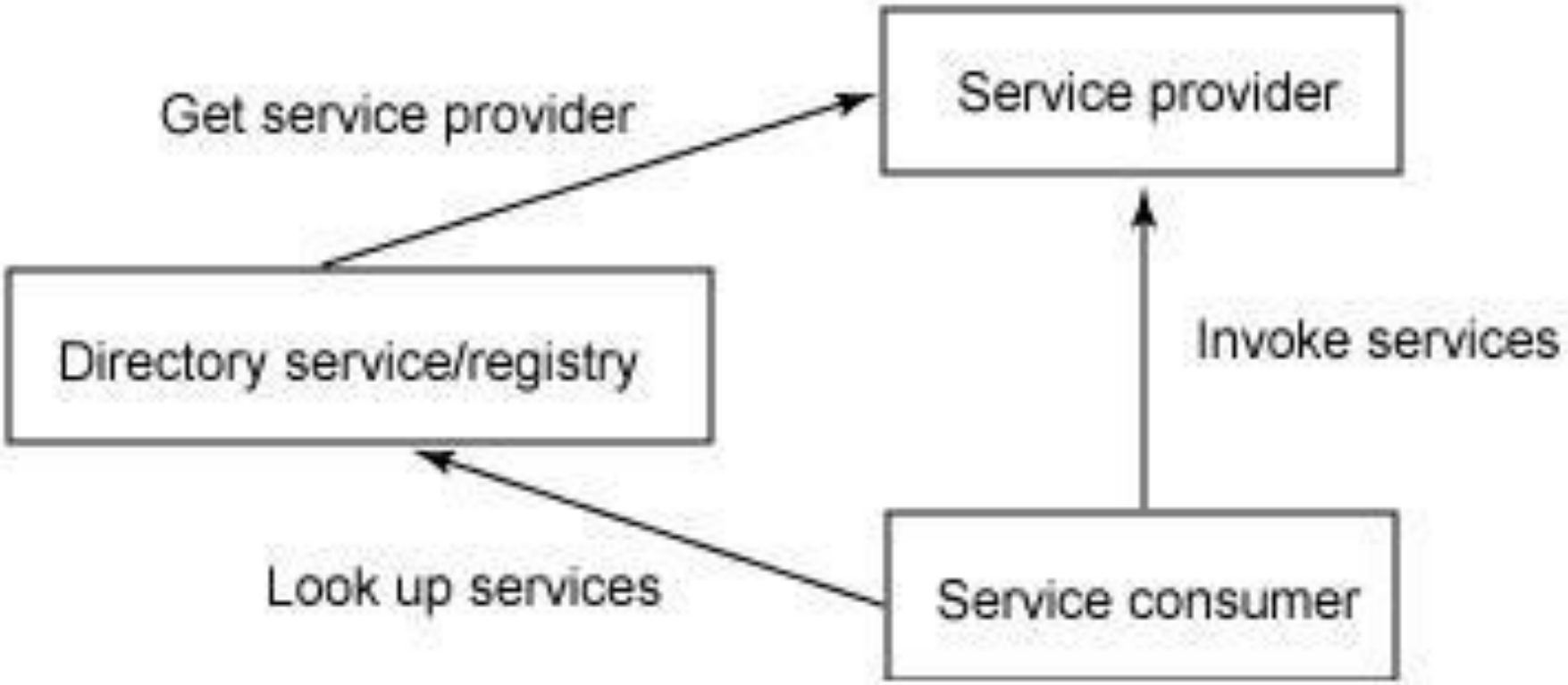
- 调度器是性能瓶颈
- 调度员的故障将使整个系统瘫痪

面向服务的架构(SOA)风格

面向服务的架构（SOA）风格

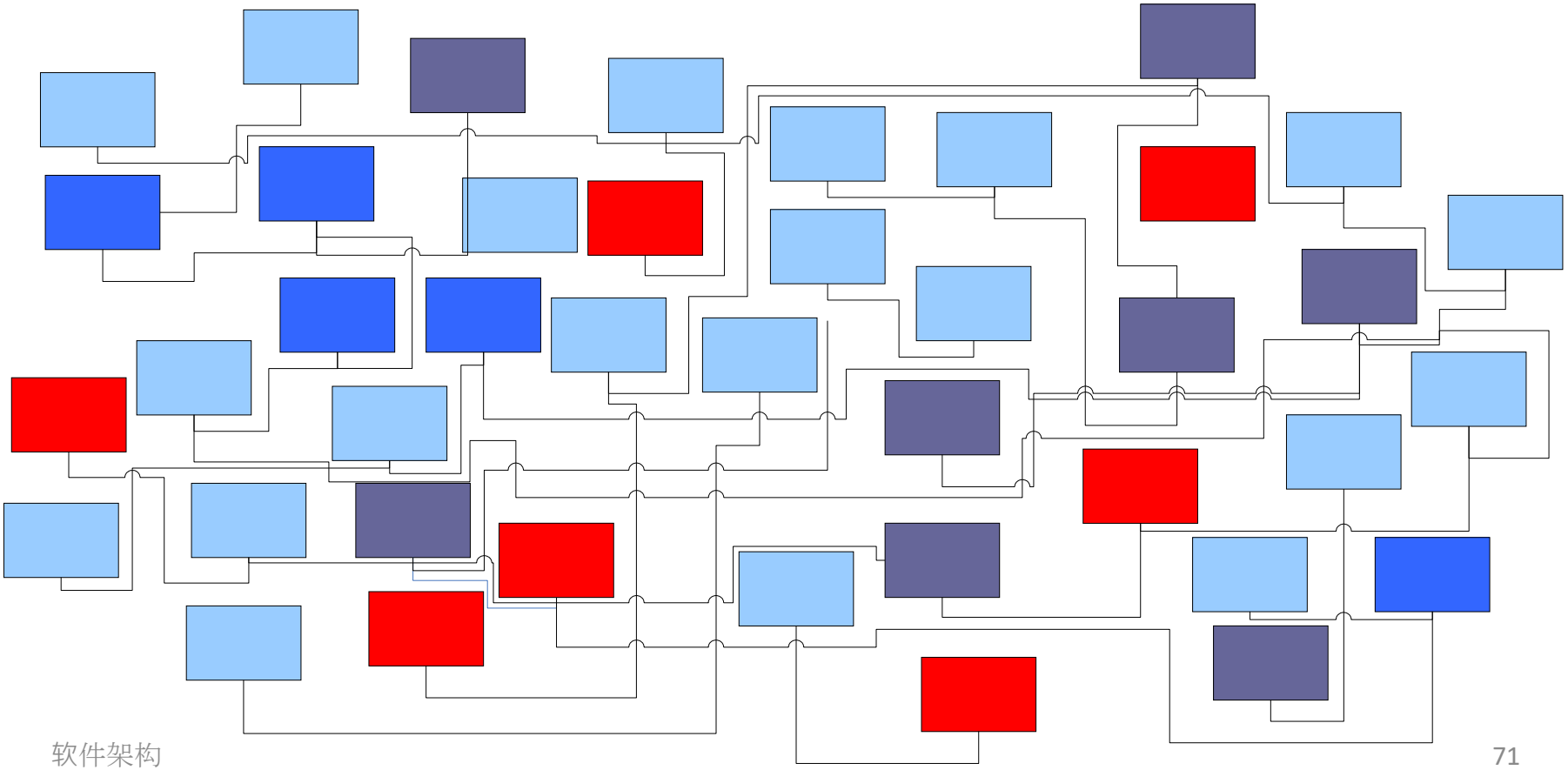
- 组成部分：服务提供者、服务请求者和服务经纪人
  - 服务提供者允许访问服务，创建一个服务的描述，并将其发布到服务经纪商。
  - 服务经纪人主持一个服务描述的登记处
  - 服务请求者通过搜索服务经纪人给出的服务描述来发现一项服务，并与之绑定。
- 连接器：网络服务协议

面向服务的架构（SOA）风格



例子

- 遗产整合



例子

- SOA整合