南京晓庄学院

毕业设计(论文)外文资料翻译

|  |  |
| --- | --- |
| **学 院：** | 信息工程学院 |
| **专 业：** | 计算机科学与技术 |
| **姓 名：** | 顾佳凯 |
| **学 号：** | 19130104 |
| **外文出处：** | Architectural Styles and the Design of Network-based Software Architectures |
|  |  |
|  |  |
|  |  |
|  |  |
| **附 件：** | 1.外文资料翻译译文；2.外文原文。 |

|  |
| --- |
| 指导教师评语： |
| |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | 2022 | 年 | 11 | 月 | 30 | 日 |   签名： |

**附件1：外文资料翻译译文**

**表述性状态转移（REST）**

本章介绍并详细描述了为分布式超媒体系统设计的表述性状态转移（REST）架构风格， 描述了指导 REST 的软件工程原则和选择用来支持这些原则的交互约束，并将它们与其他架 构风格的约束进行了对比。REST 是从第 3 章描述的几种基于网络的架构风格中衍生出来的 一种混合架构风格，并且添加了一些额外的约束，用来定义统一的连接器接口。我使用第 1 章中的软件架构框架来定义 REST 的架构元素，并检查原型架构的过程样本、连接器和数据视图。

**5.1 推导 REST**

Web 架构背后的设计基本原理，能够被描述为由一组应用于架构中元素之上的约束组成 的架构风格。当将每个约束添加到进化中的风格时，会产生一些影响。通过检查这些影响， 我们就能够识别出 Web 的约束所导致的属性。然后就能够应用额外的约束来形成一种新的 架构风格，这种风格能够更好地反映出现代 Web 架构所期待的属性。本节通过简述 REST 作 为架构风格的推导过程，提供了关于 REST 的总体概览，后面各节将会详细描述组成 REST 风格的各种特定约束。

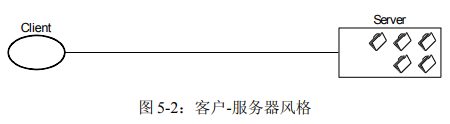
**5.1.1 从“空”风格开始**

无论是建筑还是软件，人们对架构设计的过程都有着两种常见的观点。第一种观点认为 设计师一切从零开始——一块空的石板、白板、或画板——并使用熟悉的组件建造出一个架 构，直到该架构满足希望的系统需求为止。第二种观点则认为设计师从作为一个整体的系统 需求出发，此时没有任何约束，然后增量地识别出各种约束，并将它们应用于系统的元素之 上，以便对设计空间加以区分，并允许影响到系统行为的力量（forces）与系统协调一致， 自然地流动。第一种观点强调创造性和无限的想象力，而第二种观点则强调限制和对系统环 境的理解。REST 是使用后一种过程发展而成的。随着增量地应用一组约束，已应用的约束 会将架构的过程视图区分开，图 5-1 至 5 -8 以图形化的方式依次描述了这个过程。 “空”风格（图 5-1）仅仅是一个空的约束集合。从架构的观点来看，空风格描述了一个 组件之间没有明显边界的系统。这就是我们描述 REST 的起点。



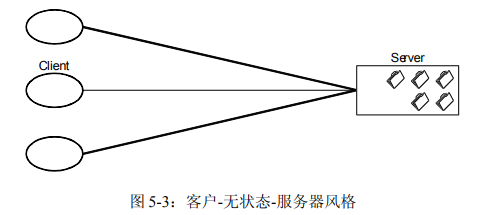
**5.1.2 客户-服务器**

首先被添加到我们的混合风格中的约束来自 3.4.1 小节描述的客户-服务器架构风格（图 5-2）。客户-服务器约束背后的原则是分离关注点。通过分离用户接口和数据存储这两个关 注点，我们改善了用户接口跨多个平台的可移植性；同时通过简化服务器组件，改善了系统 的可伸缩性。然而，对于 Web 来说，最重要的是这种关注点的分离允许组件独立地进化， 从而支持多个组织领域的 Internet 规模的需求。



**5.1.3 无状态**

我们接下来再为客户-服务器交互添加一个约束：通信必须在本质上是无状态的，如 3.4.3 小节中的客户-无状态-服务器（CSS）风格那样，因此从客户到服务器的每个请求都必 须包含理解该请求所必需的所有信息，不能利用任何存储在服务器上的上下文，会话状态因 此要全部保存在客户端。

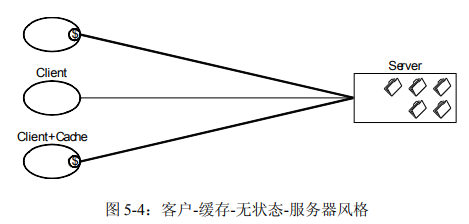


这个约束导致了可见性、可靠性和可伸缩性三个架构属性。改善了可见性是因为监视系 统不必为了确定一个请求的全部性质而去查看该请求之外的多个请求。改善了可靠性是因为 它减轻了从局部故障[133]中恢复的任务量。改善了可伸缩性是因为不必在多个请求之间保 存状态，从而允许服务器组件迅速释放资源，并进一步简化其实现，因为服务器不必跨多个 请求管理资源的使用。

与大多数架构上抉择一样，无状态这一约束反映出设计上的权衡。其缺点是：由于不能 将状态数据保存在服务器上的共享上下文中，因此增加了在一系列请求中发送的重复数据 （每次交互的开销），可能会降低网络性能。此外，将应用状态放在客户端还降低了服务器 对于一致的应用行为的控制，因为这样一来，应用就得依赖于跨多个客户端版本（译者注： 例如多个浏览器窗口）的语义的正确实现。

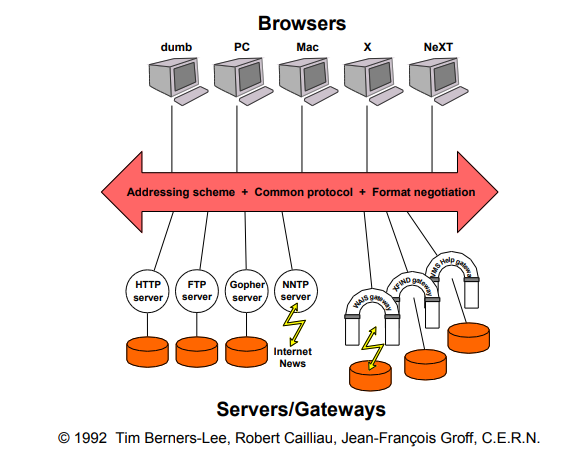
**5.1.4 缓存**

为了改善网络的效率，我们添加了缓存约束，从而形成了 3.4.4 小节描述的客户-缓存无状态-服务器风格（图 5-4）。缓存约束要求一个请求的响应中的数据被隐式地或显式地标 记为可缓存的或不可缓存的。如果响应是可缓存的，那么客户端缓存就可以为以后的相同请 求重用这个响应的数据。



添加缓存约束的好处在于，它们有可能部分或全部消除一些交互，从而通过减少一系列 交互的平均延迟时间，来提高效率、可伸缩性和用户可觉察的性能。然而，付出的代价是， 如果缓存中陈旧的数据与将请求直接发送到服务器得到的数据差别很大，那么缓存会降低可 靠性。

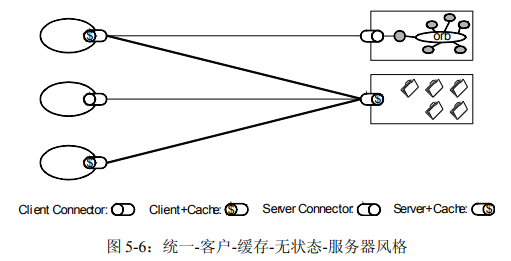
早期的 Web 架构，如图 5-5 所示[11]，是通过客户-缓存-无状态-服务器的约束集合来定 义的。也就是说，1994 年之前的 Web 架构的设计基本原理聚焦于在 Internet 上交换静态文档 的无状态的客户-服务器交互。交互的通信协议仅包含了对非共享缓存的初步支持，但是并 没有限定接口要对所有的资源提供一组一致的语义。相反，Web 依赖于使用一个公共的客户服务器实现库（CERN 的 libwww）来维护 Web 应用之间的一致性。



Web 实现的开发者早已经超越了这种早期的设计。除了静态的文档之外，请求还能够识 别出动态生成响应的服务，例如图像地图（image-maps）[Kevin Hughes]和服务器端脚本 （server-side scripts）[Rob McCool]。人们也以代理[79]和共享缓存[59]的形式开展了对中间 组件的研究，但是为了使中间组件能够可靠地通信，还需要对现有的协议进行扩展。以下几 小节描述了添加到 Web 架构风格中的约束，以便用来对形成现代 Web 架构的扩展加以指导。

**5.1.5 统一接口**

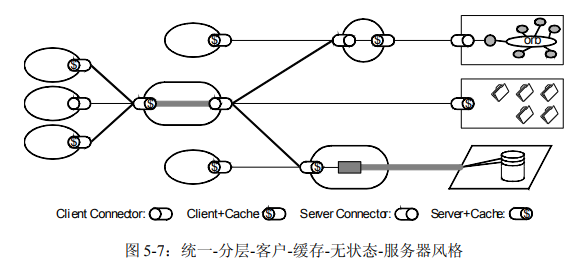
使 REST 架构风格区别于其他基于网络的架构风格的核心特征是，它强调组件之间要有 一个统一的接口（图 5-6）。通过在组件接口上应用通用性的软件工程原则，整体的系统架 构得到了简化，交互的可见性也得到了改善。实现与它们所提供的服务是解耦的，这促进了 独立的可进化性。然而，付出的代价是，统一接口降低了效率，因为信息都使用标准化的形 式来转移，而不能使用特定于应用的需求的形式。REST 接口被设计为可以高效地转移大粒 度的超媒体数据，并针对 Web 的常见情况做了优化，但是这也导致了该接口对于其他形式 的架构交互并不是最优的。



为了获得统一的接口，需要有多个架构约束来指导组件的行为。REST 由四个接口约束 来定义：资源的识别（identification of resources）、通过表述对资源执行的操作、自描述的 消息（self-descriptive messages）、以及作为应用状态引擎的超媒体。这些约束将在 5.2 节中 讨论。

**5.1.6 分层系统**

为了进一步改善与 Internet 规模的需求相关的行为，我们添加了分层的系统约束（图 5- 7）。正如 3.4.2 小节中所描述的那样，分层系统风格通过限制组件的行为（即，每个组件只 能“看到”与其交互的紧邻层），将架构分解为若干等级的层。通过将组件对系统的知识限 制在单一层内，为整个系统的复杂性设置了边界，并且提高了底层独立性。我们能够使用层 来封装遗留的服务，使新的服务免受遗留客户端的影响，通过将不常用的功能转移到一个共 享的中间组件中，从而简化组件的实现。中间组件还能够通过支持跨多个网络和处理器的负 载均衡，来改善系统的可伸缩性。

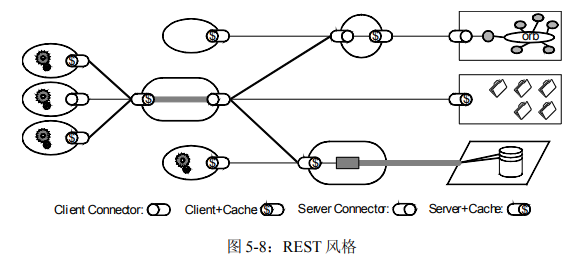


分层系统的主要缺点是：增加了数据处理的开销和延迟，因此降低了用户可觉察的性能 [32]。对于一个支持缓存约束的基于网络的系统来说，可以通过在中间层使用共享缓存所获 得的好处来弥补这一缺点。在组织领域的边界设置共享缓存能够获得显著的性能提升 [136]。这些中间层还允许我们对跨组织边界的数据强制执行安全策略，例如防火墙所要求 的那些安全策略[79]。

分层系统约束和统一接口约束相结合，导致了与统一管道和过滤器风格（3.2.2 小节） 类似的架构属性。尽管 REST 的交互是双向的，但是超媒体交互的大粒度的数据流每一个都 能够被当作一个数据流网络来处理，其中包括一些有选择地应用在数据流上的过滤器组件， 以便在数据传递的过程中对它的内容进行转换[26]。在 REST 中，中间组件能够主动地转换 消息的内容，因为这些消息是自描述的，并且其语义对于中间组件是可见的。

**5.1.7 按需代码**

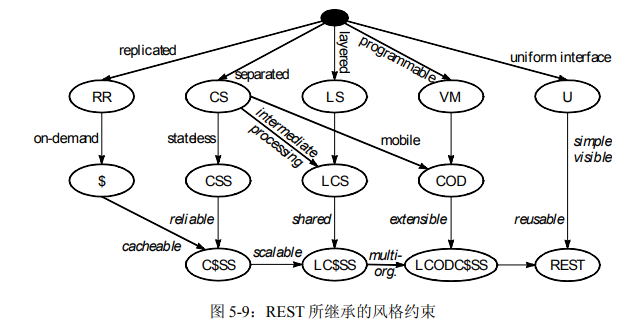
我们为 REST 添加的最后的约束来自于 3.5.3 小节中描述的按需代码风格（图 5-8）。通 过下载并执行 applet 形式或脚本形式的代码，REST 允许对客户端的功能进行扩展。这样， 通过减少必须被预先实现的功能的数目，简化了客户端的开发。允许在部署之后下载功能代 码也改善了系统的可扩展性。然而，这也降低了可见性，因此它只是 REST 的一个可选的约束。



可选的约束的想法似乎有些矛盾。然而，在设计一个包含多个组织边界的系统的架构时，它确实是有用的。这意味着只有当已知对于整个系统的某些领域有效的情况下，架构才会从 可选的约束得到好处（或蒙受损失）。例如，如果已知一个组织中的所有客户端软件都支持 Java applet[45]，那么该组织中的服务就能够构造为可以通过下载 Java 类来增强客户端的功 能，以便从可选的约束得到好处。然而，与此同时，该组织的防火墙可能会阻止转移来自外 部资源的 Java applet，因此对于 Web 的其余部分来说，这些客户端似乎是不支持按需代码的。 一个可选的约束允许我们设计在一般的场合下支持期待的行为的架构，但是我们需要理解， 这些行为可能在某些环境中无法使用。

**5.1.8 风格推导小结**

REST 由一组选择用来在候选架构上导致想要得到的属性的架构约束组成。尽管这些约 束每一个都能够独立加以考虑，但是根据它们在通用的架构风格中的来源来对它们进行描述， 使得我们理解选择它们背后的基本原理更加容易。图 5-9 根据第 3 章中调查过的基于网络的 架构风格图形化地描述了 REST 约束的来源。



**附件2：外文原文**

# Representational State Transfer (REST)

This chapter introduces and elaborates the Representational State Transfer (REST) architectural style for distributed hypermedia systems, describing the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, while contrasting them to the constraints of other architectural styles. REST is a hybrid style derived from several of the network-based architectural styles described in Chapter 3 and combined with additional constraints that define a uniform connector interface. The software architecture framework of Chapter 1 is used to define the architectural elements of REST and examine sample process, connector, and data views of prototypical architectures.

## 5.1 Deriving REST

The design rationale behind the Web architecture can be described by an architectural style consisting of the set of constraints applied to elements within the architecture. By examining the impact of each constraint as it is added to the evolving style, we can identify the properties induced by the Web's constraints. Additional constraints can then be applied to form a new architectural style that better reflects the desired properties of a modern Web architecture. This section provides a general overview of REST by walking through the process of deriving it as an architectural style. Later sections will describe in more detail the specific constraints that compose the REST style.

### 5.1.1 Starting with the Null Style

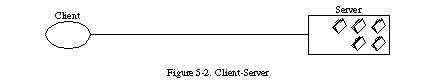
There are two common perspectives on the process of architectural design, whether it be for buildings or for software. The first is that a designer starts with nothing--a blank slate, whiteboard, or drawing board--and builds-up an architecture from familiar components until it satisfies the needs of the intended system. The second is that a designer starts with the system needs as a whole, without constraints, and then incrementally identifies and applies constraints to elements of the system in order to differentiate the design space and allow the forces that influence system behavior to flow naturally, in harmony with the system. Where the first emphasizes creativity and unbounded vision, the second emphasizes restraint and understanding of the system context. REST has been developed using the latter process. Figures 5-1 through 5-8 depict this graphically in terms of how the applied constraints would differentiate the process view of an architecture as the incremental set of constraints is applied.

The Null style ([Figure 5-1](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#fig_5_1)) is simply an empty set of constraints. From an architectural perspective, the null style describes a system in which there are no distinguished boundaries between components. It is the starting point for our description of REST.



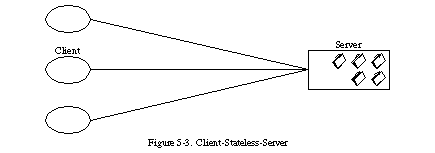
### 5.1.2 Client-Server

The first constraints added to our hybrid style are those of the client-server architectural style ([Figure 5-2](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#fig_5_2)), described in [Section 3.4.1](https://www.ics.uci.edu/~fielding/pubs/dissertation/net_arch_styles.htm#sec_3_4_1). Separation of concerns is the principle behind the client-server constraints. By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components. Perhaps most significant to the Web, however, is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains.



### 5.1.3 Stateless

We next add a constraint to the client-server interaction: communication must be stateless in nature, as in the client-stateless-server (CSS) style of [Section 3.4.3](https://www.ics.uci.edu/~fielding/pubs/dissertation/net_arch_styles.htm#sec_3_4_3) ([Figure 5-3](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#fig_5_3)), such that each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.

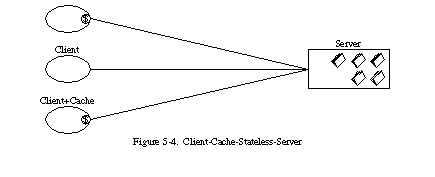


This constraint induces the properties of visibility, reliability, and scalability. Visibility is improved because a monitoring system does not have to look beyond a single request datum in order to determine the full nature of the request. Reliability is improved because it eases the task of recovering from partial failures [[133](https://www.ics.uci.edu/~fielding/pubs/dissertation/references.htm#ref_133)]. Scalability is improved because not having to store state between requests allows the server component to quickly free resources, and further simplifies implementation because the server doesn't have to manage resource usage across requests.

Like most architectural choices, the stateless constraint reflects a design trade-off. The disadvantage is that it may decrease network performance by increasing the repetitive data (per-interaction overhead) sent in a series of requests, since that data cannot be left on the server in a shared context. In addition, placing the application state on the client-side reduces the server's control over consistent application behavior, since the application becomes dependent on the correct implementation of semantics across multiple client versions.

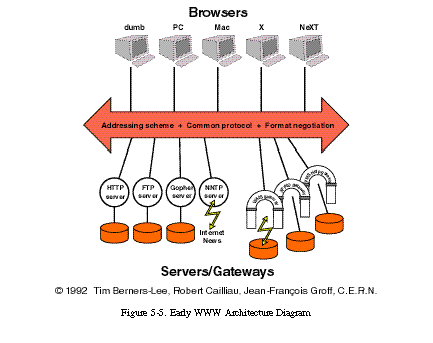
### 5.1.4 Cache

In order to improve network efficiency, we add cache constraints to form the client-cache-stateless-server style of [Section 3.4.4](https://www.ics.uci.edu/~fielding/pubs/dissertation/net_arch_styles.htm#sec_3_4_4) ([Figure 5-4](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#fig_5_4)). Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.



The advantage of adding cache constraints is that they have the potential to partially or completely eliminate some interactions, improving efficiency, scalability, and user-perceived performance by reducing the average latency of a series of interactions. The trade-off, however, is that a cache can decrease reliability if stale data within the cache differs significantly from the data that would have been obtained had the request been sent directly to the server.

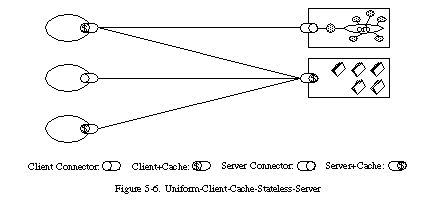
The early Web architecture, as portrayed by the diagram in [Figure 5-5](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#fig_5_5) [[11](https://www.ics.uci.edu/~fielding/pubs/dissertation/references.htm#ref_11)], was defined by the client-cache-stateless-server set of constraints. That is, the design rationale presented for the Web architecture prior to 1994 focused on stateless client-server interaction for the exchange of static documents over the Internet. The protocols for communicating interactions had rudimentary support for non-shared caches, but did not constrain the interface to a consistent set of semantics for all resources. Instead, the Web relied on the use of a common client-server implementation library (CERN libwww) to maintain consistency across Web applications.



Developers of Web implementations had already exceeded the early design. In addition to static documents, requests could identify services that dynamically generated responses, such as image-maps [Kevin Hughes] and server-side scripts [Rob McCool]. Work had also begun on intermediary components, in the form of proxies [[79](https://www.ics.uci.edu/~fielding/pubs/dissertation/references.htm#ref_79)] and shared caches [[59](https://www.ics.uci.edu/~fielding/pubs/dissertation/references.htm#ref_59)], but extensions to the protocols were needed in order for them to communicate reliably. The following sections describe the constraints added to the Web's architectural style in order to guide the extensions that form the modern Web architecture.

### 5.1.5 Uniform Interface

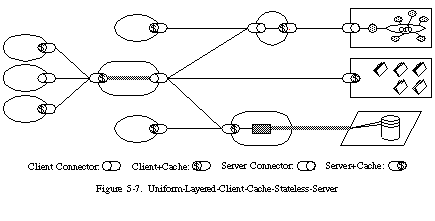
The central feature that distinguishes the REST architectural style from other network-based styles is its emphasis on a uniform interface between components ([Figure 5-6](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#fig_5_6)). By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. Implementations are decoupled from the services they provide, which encourages independent evolvability. The trade-off, though, is that a uniform interface degrades efficiency, since information is transferred in a standardized form rather than one which is specific to an application's needs. The REST interface is designed to be efficient for large-grain hypermedia data transfer, optimizing for the common case of the Web, but resulting in an interface that is not optimal for other forms of architectural interaction.



In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state. These constraints will be discussed in [Section 5.2](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#sec_5_2).

### 5.1.6 Layered System

In order to further improve behavior for Internet-scale requirements, we add layered system constraints ([Figure 5-7](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#fig_5_7)). As described in [Section 3.4.2](https://www.ics.uci.edu/~fielding/pubs/dissertation/net_arch_styles.htm#sec_3_4_2), the layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot "see" beyond the immediate layer with which they are interacting. By restricting knowledge of the system to a single layer, we place a bound on the overall system complexity and promote substrate independence. Layers can be used to encapsulate legacy services and to protect new services from legacy clients, simplifying components by moving infrequently used functionality to a shared intermediary. Intermediaries can also be used to improve system scalability by enabling load balancing of services across multiple networks and processors.

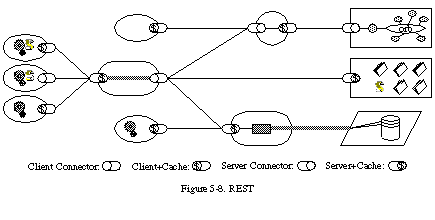


The primary disadvantage of layered systems is that they add overhead and latency to the processing of data, reducing user-perceived performance [[32](https://www.ics.uci.edu/~fielding/pubs/dissertation/references.htm#ref_32)]. For a network-based system that supports cache constraints, this can be offset by the benefits of shared caching at intermediaries. Placing shared caches at the boundaries of an organizational domain can result in significant performance benefits [[136](https://www.ics.uci.edu/~fielding/pubs/dissertation/references.htm#ref_136)]. Such layers also allow security policies to be enforced on data crossing the organizational boundary, as is required by firewalls [[79](https://www.ics.uci.edu/~fielding/pubs/dissertation/references.htm#ref_79)].

The combination of layered system and uniform interface constraints induces architectural properties similar to those of the uniform pipe-and-filter style ([Section 3.2.2](https://www.ics.uci.edu/~fielding/pubs/dissertation/net_arch_styles.htm#sec_3_2_2)). Although REST interaction is two-way, the large-grain data flows of hypermedia interaction can each be processed like a data-flow network, with filter components selectively applied to the data stream in order to transform the content as it passes [[26](https://www.ics.uci.edu/~fielding/pubs/dissertation/references.htm#ref_26)]. Within REST, intermediary components can actively transform the content of messages because the messages are self-descriptive and their semantics are visible to intermediaries.

### 5.1.7 Code-On-Demand

The final addition to our constraint set for REST comes from the code-on-demand style of [Section 3.5.3](https://www.ics.uci.edu/~fielding/pubs/dissertation/net_arch_styles.htm#sec_3_5_3) ([Figure 5-8](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#fig_5_8)). REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility. However, it also reduces visibility, and thus is only an optional constraint within REST.



The notion of an optional constraint may seem like an oxymoron. However, it does have a purpose in the architectural design of a system that encompasses multiple organizational boundaries. It means that the architecture only gains the benefit (and suffers the disadvantages) of the optional constraints when they are known to be in effect for some realm of the overall system. For example, if all of the client software within an organization is known to support Java applets [[45](https://www.ics.uci.edu/~fielding/pubs/dissertation/references.htm#ref_45)], then services within that organization can be constructed such that they gain the benefit of enhanced functionality via downloadable Java classes. At the same time, however, the organization's firewall may prevent the transfer of Java applets from external sources, and thus to the rest of the Web it will appear as if those clients do not support code-on-demand. An optional constraint allows us to design an architecture that supports the desired behavior in the general case, but with the understanding that it may be disabled within some contexts.

### 5.1.8 Style Derivation Summary

REST consists of a set of architectural constraints chosen for the properties they induce on candidate architectures. Although each of these constraints can be considered in isolation, describing them in terms of their derivation from common architectural styles makes it easier to understand the rationale behind their selection. [Figure 5-9](https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm#fig_5_9) depicts the derivation of REST's constraints graphically in terms of the network-based architectural styles examined in Chapter 3.

