CHAPTER 5

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) 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.

Figure 5-1. Null Style

5.1.2 Client-Server

The first constraints added to our hybrid style are those of the client-server architectural style (Figure 5-2), described in Section 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.

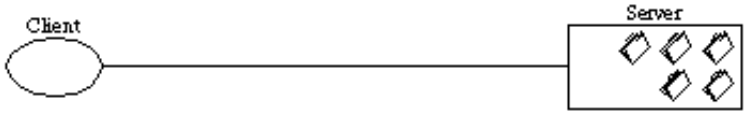

Figure 5-2. Client-Server

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 (Figure 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.

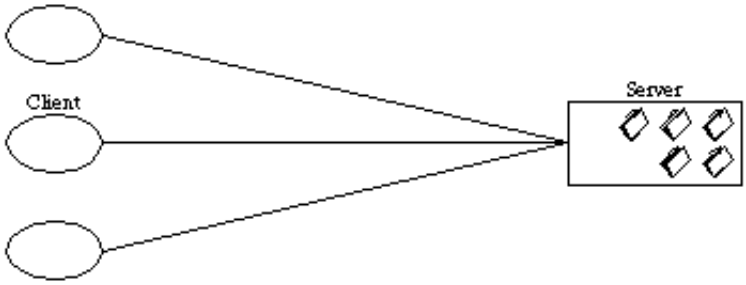

Figure 5-3. Client-Stateless-Server

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]. 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 (Figure 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.

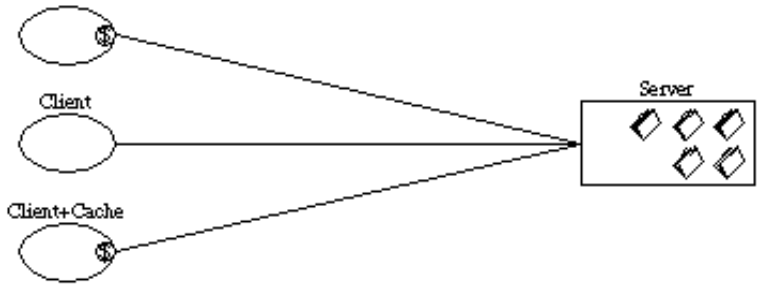

Figure 5-4. Client-Cache-Stateless-Server

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 [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.

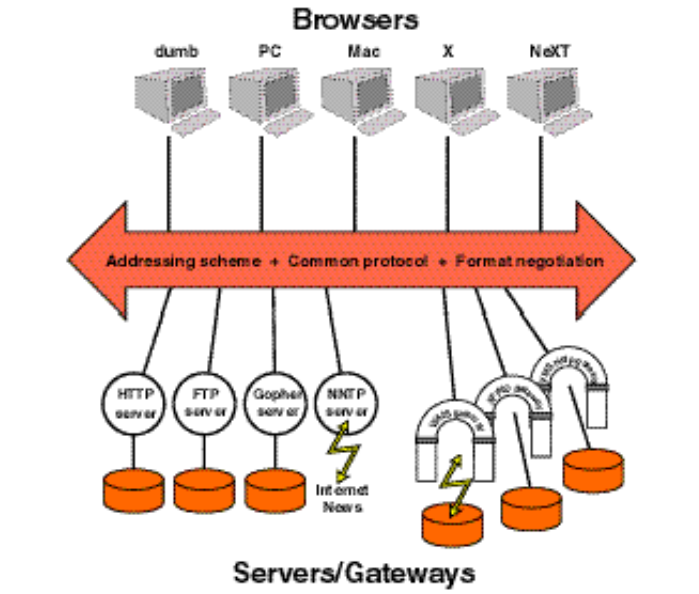

Figure 5-5. Early WWW Architecture Diagram

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] and shared caches [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). 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.

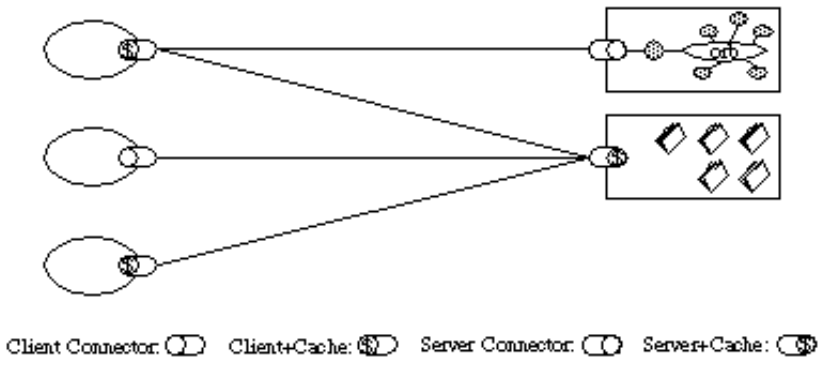

Figure 5-6. Uniform-Client-Cache-Stateless-Server

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.

5.1.6 Layered System

In order to further improve behavior for Internet-scale requirements, we add layered system constraints (Figure 5-7). As described in Section 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.

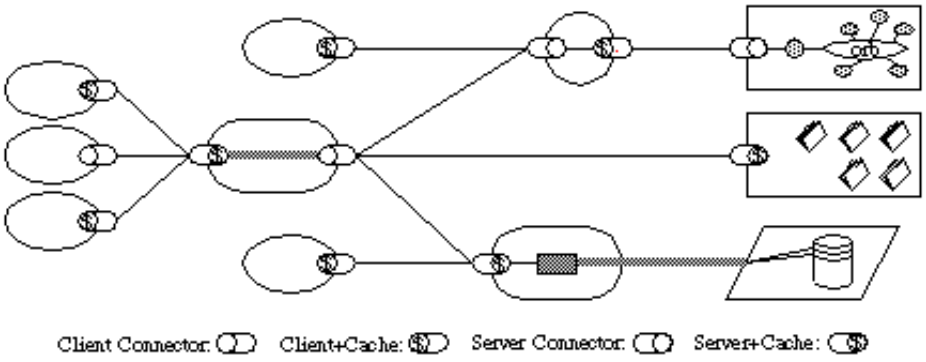

Figure 5-7. Uniform-Layered-Client-Cache-Stateless-Server

The primary disadvantage of layered systems is that they add overhead and latency to the processing of

data, reducing user-perceived performance [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]. Such layers also allow security policies to be enforced on data crossing the organizational boundary, as is required by firewalls [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). 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]. 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 (Figure 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.

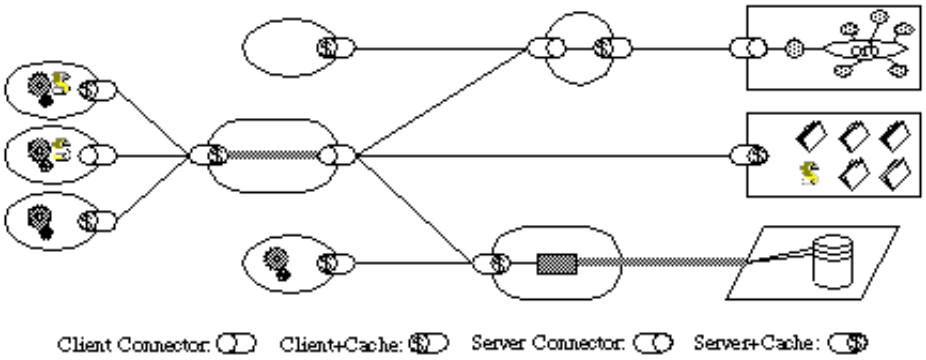

Figure 5-8. REST

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], 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 depicts the derivation of REST's constraints graphically in terms of the network-based architectural styles examined in Chapter 3.

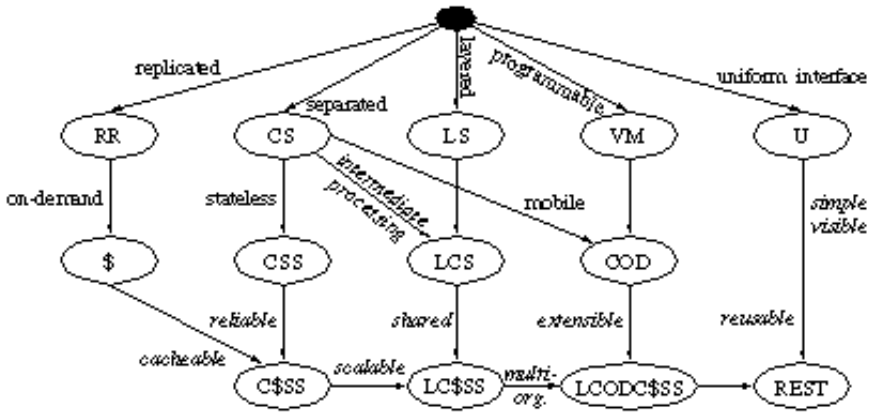

Figure 5-9. REST Derivation by Style Constraints

5.2 REST Architectural Elements

The Representational State Transfer (REST) style is an abstraction of the architectural elements within a distributed hypermedia system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements. It encompasses the fundamental constraints upon components, connectors, and data that define the basis of the Web architecture, and thus the essence of its behavior as a network-based application.

5.2.1 Data Elements

Unlike the distributed object style [31], where all data is encapsulated within and hidden by the processing components, the nature and state of an architecture's data elements is a key aspect of REST. The rationale for this design can be seen in the nature of distributed hypermedia. When a link is selected, information needs to be moved from the location where it is stored to the location where it will be used by, in most cases, a human reader. This is unlike many other distributed processing paradigms [6, 50], where it is possible, and usually more efficient, to move the "processing agent" (e.g., mobile code, stored procedure, search expression, etc.) to the data rather than move the data to the processor.

A distributed hypermedia architect has only three fundamental options: 1) render the data where it is located and send a fixed-format image to the recipient; 2) encapsulate the data with a rendering engine and send both to the recipient; or, 3) send the raw data to the recipient along with metadata that describes the data type, so that the recipient can choose their own rendering engine.

Each option has its advantages and disadvantages. Option 1, the traditional client-server style [31], allows all information about the true nature of the data to remain hidden within the sender, preventing assumptions from being made about the data structure and making client implementation easier. However, it also severely restricts the functionality of the recipient and places most of the processing load on the sender, leading to scalability problems. Option 2, the mobile object style [50], provides information hiding while enabling specialized processing of the data via its unique rendering engine, but limits the functionality of the recipient to what is anticipated within that engine and may vastly increase the amount of data transferred. Option 3 allows the sender to remain simple and scalable while minimizing the bytes transferred, but loses the advantages of information hiding and requires that both sender and recipient understand the same data types.

REST provides a hybrid of all three options by focusing on a shared understanding of data types with metadata, but limiting the scope of what is revealed to a standardized interface. REST components communicate by transferring a representation of a resource in a format matching one of an evolving set

of standard data types, selected dynamically based on the capabilities or desires of the recipient and the nature of the resource. Whether the representation is in the same format as the raw source, or is derived from the source, remains hidden behind the interface. The benefits of the mobile object style are approximated by sending a representation that consists of instructions in the standard data format of an encapsulated rendering engine (e.g., Java [45]). REST therefore gains the separation of concerns of the client-server style without the server scalability problem, allows information hiding through a generic interface to enable encapsulation and evolution of services, and provides for a diverse set of functionality through downloadable feature-engines.

REST's data elements are summarized in Table 5-1.

Table 5-1: REST Data Elements

Data Element	Modern Web Examples
resource	the intended conceptual target of a hypertext reference
resource identifier	URL, URN
representation	HTML document, JPEG image
representation metadata	media type, last-modified time
resource metadata	source link, alternates, vary
control data	if-modified-since, cache-control

5.2.1.1 Resources and Resource Identifiers

The key abstraction of information in REST is a *resource*. Any information that can be named can be a resource: a document or image, a temporal service (e.g. "today's weather in Los Angeles"), a collection of other resources, a non-virtual object (e.g. a person), and so on. In other words, any concept that might be the target of an author's hypertext reference must fit within the definition of a resource. A resource is a conceptual mapping to a set of entities, not the entity that corresponds to the mapping at any particular point in time.

More precisely, a resource R is a temporally varying membership function $M_R(t)$, which for time t maps to a set of entities, or values, which are equivalent. The values in the set may be *resource representations* and/or *resource identifiers*. A resource can map to the empty set, which allows references to be made to a concept before any realization of that concept exists -- a notion that was foreign to most hypertext systems prior to the Web [61]. Some resources are static in the sense that, when examined at any time after their creation, they always correspond to the same value set. Others have a high degree of variance in their value over time. The only thing that is required to be static for a resource is the semantics of the mapping, since the semantics is what distinguishes one resource from another.

For example, the "authors' preferred version" of an academic paper is a mapping whose value changes over time, whereas a mapping to "the paper published in the proceedings of conference X" is static. These are two distinct resources, even if they both map to the same value at some point in time. The distinction is necessary so that both resources can be identified and referenced independently. A similar example from software engineering is the separate identification of a version-controlled source code file when referring to the "latest revision", "revision number 1.2.7", or "revision included with the Orange release."

This abstract definition of a resource enables key features of the Web architecture. First, it provides generality by encompassing many sources of information without artificially distinguishing them by type or implementation. Second, it allows late binding of the reference to a representation, enabling content negotiation to take place based on characteristics of the request. Finally, it allows an author to reference the concept rather than some singular representation of that concept, thus removing the need to change all existing links whenever the representation changes (assuming the author used the right identifier).

all existing links whenever the representation changes (assuming the author used the right identifier). REST uses a resource identifier to identify the particular resource involved in an interaction between components. REST connectors provide a generic interface for accessing and manipulating the value set of a resource, regardless of how the membership function is defined or the type of software that is handling the request. The naming authority that assigned the resource identifier, making it possible to reference the resource, is responsible for maintaining the semantic validity of the mapping over time (i.e., ensuring that the membership function does not change).

Traditional hypertext systems [61], which typically operate in a closed or local environment, use unique node or document identifiers that change every time the information changes, relying on link servers to maintain references separately from the content [135]. Since centralized link servers are an anathema to the immense scale and multi-organizational domain requirements of the Web, REST relies instead on the author choosing a resource identifier that best fits the nature of the concept being identified. Naturally, the quality of an identifier is often proportional to the amount of money spent to retain its validity, which leads to broken links as ephemeral (or poorly supported) information moves or disappears over time.

5.2.1.2 Representations

REST components perform actions on a resource by using a representation to capture the current or intended state of that resource and transferring that representation between components. A representation is a sequence of bytes, plus representation metadata to describe those bytes. Other commonly used but less precise names for a representation include: document, file, and HTTP message entity, instance, or variant.

A representation consists of data, metadata describing the data, and, on occasion, metadata to describe the metadata (usually for the purpose of verifying message integrity). Metadata is in the form of name-value pairs, where the name corresponds to a standard that defines the value's structure and semantics. Response messages may include both representation metadata and resource metadata: information about the resource that is not specific to the supplied representation.

Control data defines the purpose of a message between components, such as the action being requested or the meaning of a response. It is also used to parameterize requests and override the default behavior of some connecting elements. For example, cache behavior can be modified by control data included in the request or response message.

Depending on the message control data, a given representation may indicate the current state of the requested resource, the desired state for the requested resource, or the value of some other resource, such as a representation of the input data within a client's query form, or a representation of some error condition for a response. For example, remote authoring of a resource requires that the author send a representation to the server, thus establishing a value for that resource that can be retrieved by later requests. If the value set of a resource at a given time consists of multiple representations, content negotiation may be used to select the best representation for inclusion in a given message.

The data format of a representation is known as a media type [48]. A representation can be included in a message and processed by the recipient according to the control data of the message and the nature of the media type. Some media types are intended for automated processing, some are intended to be rendered for viewing by a user, and a few are capable of both. Composite media types can be used to enclose multiple representations in a single message.

The design of a media type can directly impact the user-perceived performance of a distributed hypermedia system. Any data that must be received before the recipient can begin rendering the representation adds to the latency of an interaction. A data format that places the most important rendering information up front, such that the initial information can be incrementally rendered while the rest of the information is being received, results in much better user-perceived performance than a data format that must be entirely received before rendering can begin.

For example, a Web browser that can incrementally render a large HTML document while it is being received provides significantly better user-perceived performance than one that waits until the entire document is completely received prior to rendering, even though the network performance is the same. Note that the rendering ability of a representation can also be impacted by the choice of content. If the dimensions of dynamically-sized tables and embedded objects must be determined before they can be rendered, their occurrence within the viewing area of a hypermedia page will increase its latency.

5.2.2 Connectors

REST uses various connector types, summarized in Table 5-2, to encapsulate the activities of accessing resources and transferring resource representations. The connectors present an abstract interface for component communication, enhancing simplicity by providing a clean separation of concerns and hiding the underlying implementation of resources and communication mechanisms. The generality of the interface also enables substitutability: if the users' only access to the system is via an abstract interface, the implementation can be replaced without impacting the users. Since a connector manages network communication for a component, information can be shared across multiple interactions in order to improve efficiency and responsiveness.

Table 5-2: REST Connectors

Connector	Modern Web Examples
client	libwww, libwww-perl
server	libwww, Apache API, NSAPI
cache	browser cache, Akamai cache network
resolver	bind (DNS lookup library)
tunnel	SOCKS, SSL after HTTP CONNECT

All REST interactions are stateless. That is, each request contains all of the information necessary for a connector to understand the request, independent of any requests that may have preceded it. This restriction accomplishes four functions: 1) it removes any need for the connectors to retain application state between requests, thus reducing consumption of physical resources and improving scalability; 2) it allows interactions to be processed in parallel without requiring that the processing mechanism understand the interaction semantics; 3) it allows an intermediary to view and understand a request in isolation, which may be necessary when services are dynamically rearranged; and, 4) it forces all of the information that might factor into the reusability of a cached response to be present in each request.

The connector interface is similar to procedural invocation, but with important differences in the passing of parameters and results. The in-parameters consist of request control data, a resource identifier indicating the target of the request, and an optional representation. The out-parameters consist of response control data, optional resource metadata, and an optional representation. From an abstract viewpoint the invocation is synchronous, but both in and out-parameters can be passed as data streams. In other words, processing can be invoked before the value of the parameters is completely known, thus avoiding the latency of batch processing large data transfers.

The primary connector types are client and server. The essential difference between the two is that a client initiates communication by making a request, whereas a server listens for connections and responds to requests in order to supply access to its services. A component may include both client and server connectors.

A third connector type, the cache connector, can be located on the interface to a client or server connector in order to save cacheable responses to current interactions so that they can be reused for later requested interactions. A cache may be used by a client to avoid repetition of network communication, or by a server to avoid repeating the process of generating a response, with both cases serving to reduce interaction latency. A cache is typically implemented within the address space of the connector that uses

it.

Some cache connectors are shared, meaning that its cached responses may be used in answer to a client other than the one for which the response was originally obtained. Shared caching can be effective at reducing the impact of "flash crowds" on the load of a popular server, particularly when the caching is arranged hierarchically to cover large groups of users, such as those within a company's intranet, the customers of an Internet service provider, or Universities sharing a national network backbone. However, shared caching can also lead to errors if the cached response does not match what would have been obtained by a new request. REST attempts to balance the desire for transparency in cache behavior with the desire for efficient use of the network, rather than assuming that absolute transparency is always required.

A cache is able to determine the cacheability of a response because the interface is generic rather than specific to each resource. By default, the response to a retrieval request is cacheable and the responses to other requests are non-cacheable. If some form of user authentication is part of the request, or if the response indicates that it should not be shared, then the response is only cacheable by a non-shared cache. A component can override these defaults by including control data that marks the interaction as cacheable, non-cacheable or cacheable for only a limited time.

A resolver translates partial or complete resource identifiers into the network address information needed to establish an inter-component connection. For example, most URI include a DNS hostname as the mechanism for identifying the naming authority for the resource. In order to initiate a request, a Web browser will extract the hostname from the URI and make use of a DNS resolver to obtain the Internet Protocol address for that authority. Another example is that some identification schemes (e.g., URN [124]) require an intermediary to translate a permanent identifier to a more transient address in order to access the identified resource. Use of one or more intermediate resolvers can improve the longevity of resource references through indirection, though doing so adds to the request latency.

The final form of connector type is a tunnel, which simply relays communication across a connection boundary, such as a firewall or lower-level network gateway. The only reason it is modeled as part of REST and not abstracted away as part of the network infrastructure is that some REST components may dynamically switch from active component behavior to that of a tunnel. The primary example is an HTTP proxy that switches to a tunnel in response to a CONNECT method request [71], thus allowing its client to directly communicate with a remote server using a different protocol, such as TLS, that doesn't allow proxies. The tunnel disappears when both ends terminate their communication.

5.2.3 Components

REST components, summarized in Table 5-3, are typed by their roles in an overall application action.

Table 5-3: REST Components

Component	Modern Web Examples
origin server	Apache httpd, Microsoft IIS
gateway	Squid, CGI, Reverse Proxy
proxy	CERN Proxy, Netscape Proxy, Gauntlet
user agent	Netscape Navigator, Lynx, MOMspider

A user agent uses a client connector to initiate a request and becomes the ultimate recipient of the response. The most common example is a Web browser, which provides access to information services and renders service responses according to the application needs.

An origin server uses a server connector to govern the namespace for a requested resource. It is the definitive source for representations of its resources and must be the ultimate recipient of any request that

intends to modify the value of its resources. Each origin server provides a generic interface to its services as a resource hierarchy. The resource implementation details are hidden behind the interface.

Intermediary components act as both a client and a server in order to forward, with possible translation, requests and responses. A proxy component is an intermediary selected by a client to provide interface encapsulation of other services, data translation, performance enhancement, or security protection. A gateway (a.k.a., reverse proxy) component is an intermediary imposed by the network or origin server to provide an interface encapsulation of other services, for data translation, performance enhancement, or security enforcement. Note that the difference between a proxy and a gateway is that a client determines when it will use a proxy.

5.3 REST Architectural Views

Now that we have an understanding of the REST architectural elements in isolation, we can use architectural views [105] to describe how the elements work together to form an architecture. Three types of view--process, connector, and data--are useful for illuminating the design principles of REST.

5.3.1 Process View

A process view of an architecture is primarily effective at eliciting the interaction relationships among components by revealing the path of data as it flows through the system. Unfortunately, the interaction of a real system usually involves an extensive number of components, resulting in an overall view that is obscured by the details. Figure 5-10 provides a sample of the process view from a REST-based architecture at a particular instance during the processing of three parallel requests.

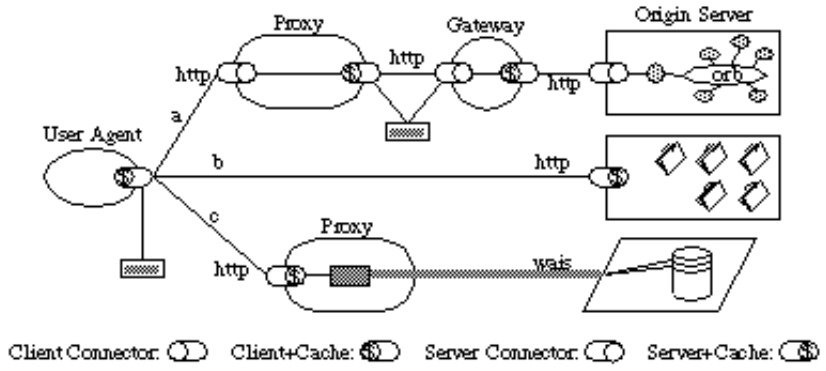

Figure 5-10. Process View of a REST-based Architecture

A user agent is portrayed in the midst of three parallel interactions: a, b, and c. The interactions were not satisfied by the user agent's client connector cache, so each request has been routed to the resource origin according to the properties of each resource identifier and the configuration of the client connector. Request (a) has been sent to a local proxy, which in turn accesses a caching gateway found by DNS lookup, which forwards the request on to be satisfied by an origin server whose internal resources are defined by an encapsulated object request broker architecture. Request (b) is sent directly to an origin server, which is able to satisfy the request from its own cache. Request (c) is sent to a proxy that is capable of directly accessing WAIS, an information service that is separate from the Web architecture, and translating the WAIS response into a format recognized by the generic connector interface. Each component is only aware of the interaction with their own client or server connectors; the overall process topology is an artifact of our view.

REST's client-server separation of concerns simplifies component implementation, reduces the complexity of connector semantics, improves the effectiveness of performance tuning, and increases the scalability of pure server components. Layered system constraints allow intermediaries--proxies, gateways, and firewalls--to be introduced at various points in the communication without changing the interfaces between components, thus allowing them to assist in communication translation or improve performance via large-scale, shared caching. REST enables intermediate processing by constraining messages to be self-descriptive: interaction is stateless between requests, standard methods and media types are used to indicate semantics and exchange information, and responses explicitly indicate

cacheability.

Since the components are connected dynamically, their arrangement and function for a particular application action has characteristics similar to a pipe-and-filter style. Although REST components communicate via bidirectional streams, the processing of each direction is independent and therefore susceptible to stream transducers (filters). The generic connector interface allows components to be placed on the stream based on the properties of each request or response.

Services may be implemented using a complex hierarchy of intermediaries and multiple distributed origin servers. The stateless nature of REST allows each interaction to be independent of the others, removing the need for an awareness of the overall component topology, an impossible task for an Internet-scale architecture, and allowing components to act as either destinations or intermediaries, determined dynamically by the target of each request. Connectors need only be aware of each other's existence during the scope of their communication, though they may cache the existence and capabilities of other components for performance reasons.

5.3.2 Connector View

A connector view of an architecture concentrates on the mechanics of the communication between components. For a REST-based architecture, we are particularly interested in the constraints that define the generic resource interface.

Client connectors examine the resource identifier in order to select an appropriate communication mechanism for each request. For example, a client may be configured to connect to a specific proxy component, perhaps one acting as an annotation filter, when the identifier indicates that it is a local resource. Likewise, a client can be configured to reject requests for some subset of identifiers.

REST does not restrict communication to a particular protocol, but it does constrain the interface between components, and hence the scope of interaction and implementation assumptions that might otherwise be made between components. For example, the Web's primary transfer protocol is HTTP, but the architecture also includes seamless access to resources that originate on pre-existing network servers, including FTP [107], Gopher [7], and WAIS [36]. Interaction with those services is restricted to the semantics of a REST connector. This constraint sacrifices some of the advantages of other architectures, such as the stateful interaction of a relevance feedback protocol like WAIS, in order to retain the advantages of a single, generic interface for connector semantics. In return, the generic interface makes it possible to access a multitude of services through a single proxy. If an application needs the additional capabilities of another architecture, it can implement and invoke those capabilities as a separate system running in parallel, similar to how the Web architecture interfaces with "telnet" and "mailto" resources.

5.3.3 Data View

A data view of an architecture reveals the application state as information flows through the components. Since REST is specifically targeted at distributed information systems, it views an application as a cohesive structure of information and control alternatives through which a user can perform a desired task. For example, looking-up a word in an on-line dictionary is one application, as is touring through a virtual museum, or reviewing a set of class notes to study for an exam. Each application defines goals for the underlying system, against which the system's performance can be measured.

Component interactions occur in the form of dynamically sized messages. Small or medium-grain messages are used for control semantics, but the bulk of application work is accomplished via large-grain messages containing a complete resource representation. The most frequent form of request semantics is that of retrieving a representation of a resource (e.g., the "GET" method in HTTP), which

can often be cached for later reuse.

REST concentrates all of the control state into the representations received in response to interactions. The goal is to improve server scalability by eliminating any need for the server to maintain an awareness of the client state beyond the current request. An application's state is therefore defined by its pending requests, the topology of connected components (some of which may be filtering buffered data), the active requests on those connectors, the data flow of representations in response to those requests, and the processing of those representations as they are received by the user agent.

An application reaches a steady-state whenever it has no outstanding requests; i.e., it has no pending requests and all of the responses to its current set of requests have been completely received or received to the point where they can be treated as a representation data stream. For a browser application, this state corresponds to a "web page," including the primary representation and ancillary representations, such as in-line images, embedded applets, and style sheets. The significance of application steady-states is seen in their impact on both user-perceived performance and the burstiness of network request traffic.

The user-perceived performance of a browser application is determined by the latency between steady-states: the period of time between the selection of a hypermedia link on one web page and the point when usable information has been rendered for the next web page. The optimization of browser performance is therefore centered around reducing this communication latency.

Since REST-based architectures communicate primarily through the transfer of representations of resources, latency can be impacted by both the design of the communication protocols and the design of the representation data formats. The ability to incrementally render the response data as it is received is determined by the design of the media type and the availability of layout information (visual dimensions of in-line objects) within each representation.

An interesting observation is that the most efficient network request is one that doesn't use the network. In other words, the ability to reuse a cached response results in a considerable improvement in application performance. Although use of a cache adds some latency to each individual request due to lookup overhead, the average request latency is significantly reduced when even a small percentage of requests result in usable cache hits.

The next control state of an application resides in the representation of the first requested resource, so obtaining that first representation is a priority. REST interaction is therefore improved by protocols that "respond first and think later." In other words, a protocol that requires multiple interactions per user action, in order to do things like negotiate feature capabilities prior to sending a content response, will be perceptively slower than a protocol that sends whatever is most likely to be optimal first and then provides a list of alternatives for the client to retrieve if the first response is unsatisfactory.

The application state is controlled and stored by the user agent and can be composed of representations from multiple servers. In addition to freeing the server from the scalability problems of storing state, this allows the user to directly manipulate the state (e.g., a Web browser's history), anticipate changes to that state (e.g., link maps and prefetching of representations), and jump from one application to another (e.g., bookmarks and URI-entry dialogs).

The model application is therefore an engine that moves from one state to the next by examining and choosing from among the alternative state transitions in the current set of representations. Not surprisingly, this exactly matches the user interface of a hypermedia browser. However, the style does not assume that all applications are browsers. In fact, the application details are hidden from the server by the generic connector interface, and thus a user agent could equally be an automated robot performing information retrieval for an indexing service, a personal agent looking for data that matches certain criteria, or a maintenance spider busy patrolling the information for broken references or modified content [39].

5.4 Related Work

Bass, et al. [9] devote a chapter on architecture for the World Wide Web, but their description only encompasses the implementation architecture within the CERN/W3C developed libwww (client and server libraries) and Jigsaw software. Although those implementations reflect many of the design constraints of REST, having been developed by people familiar with the Web's architectural design and rationale, the real WWW architecture is independent of any single implementation. The modern Web is defined by its standard interfaces and protocols, not how those interfaces and protocols are implemented in a given piece of software.

The REST style draws from many preexisting distributed process paradigms [6, 50], communication protocols, and software fields. REST component interactions are structured in a layered client-server style, but the added constraints of the generic resource interface create the opportunity for substitutability and inspection by intermediaries. Requests and responses have the appearance of a remote invocation style, but REST messages are targeted at a conceptual resource rather than an implementation identifier.

Several attempts have been made to model the Web architecture as a form of distributed file system (e.g., WebNFS) or as a distributed object system [83]. However, they exclude various Web resource types or implementation strategies as being "not interesting," when in fact their presence invalidates the assumptions that underlie such models. REST works well because it does not limit the implementation of resources to certain predefined models, allowing each application to choose an implementation that best matches its own needs and enabling the replacement of implementations without impacting the user.

The interaction method of sending representations of resources to consuming components has some parallels with event-based integration (EBI) styles. The key difference is that EBI styles are push-based. The component containing the state (equivalent to an origin server in REST) issues an event whenever the state changes, whether or not any component is actually interested in or listening for such an event. In the REST style, consuming components usually pull representations. Although this is less efficient when viewed as a single client wishing to monitor a single resource, the scale of the Web makes an unregulated push model infeasible.

The principled use of the REST style in the Web, with its clear notion of components, connectors, and representations, relates closely to the C2 architectural style [128]. The C2 style supports the development of distributed, dynamic applications by focusing on structured use of connectors to obtain substrate independence. C2 applications rely on asynchronous notification of state changes and request messages. As with other event-based schemes, C2 is nominally push-based, though a C2 architecture could operate in REST's pull style by only emitting a notification upon receipt of a request. However, the C2 style lacks the intermediary-friendly constraints of REST, such as the generic resource interface, guaranteed stateless interactions, and intrinsic support for caching.

5.5 Summary

This chapter introduced the Representational State Transfer (REST) architectural style for distributed hypermedia systems. REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems. I described 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.

The next chapter presents an evaluation of the REST architecture through the experience and lessons learned from applying REST to the design, specification, and deployment of the modern Web architecture. This work included authoring the current Internet standards-track specifications of the

Hypertext Transfer Protocol (HTTP/1.1) and Uniform Resource Identifiers (URI), and implementing the architecture through the libwww-perl client protocol library and Apache HTTP server.
