Lateral Stack Transfer

Complex conversations
in a single request

Reggie Wilbanks
reggie@wilbanks.info

Introduction

This article introduces and describes the Lateral Stack Transfer (LAST) architectural style for the communication of processing instructions to distributed hypermedia systems. The LAST architectural style is intended to supplement other styles such as Representational State Transfer (REST)[1], and makes use of standards and recommendations such as Hypertext Transfer Protocol (HTTP)[2] (specifically the GET and POST methods), Extensible Markup Language (XML)[3], and ECMAScript for XML (E4X)[4].

Although, best efforts have been made to describe any associated standards and processes where applicable regarding their relationships pertaining to the LAST architectural style, it is suggested that readers familiarize themselves with all related topics before delving too deeply into LAST.

The LAST architectural style makes no attempt to define the actual processing of instructions; however, it does put constraints on what processing instructions should look like when passed to a "processor". It also proposes how a processor internally makes use of processing instructions, and how processed instructions should look like as returned from a processor in order to conform to the "stacked" nature of the style. It is best to visualize LAST as encapsulating the beginning and ending of a request transaction, while providing a structure by which the middle "processing" stage of the transaction should process instructions, without attempting to define the logic or validity of the processing itself.

The elegance of a distributed hypermedia system (the Internet for example), lies in its ability to efficiently present data as "resources" to the requester. A simple example of this data presentation transaction is the request and retrieval of static content from a web site:

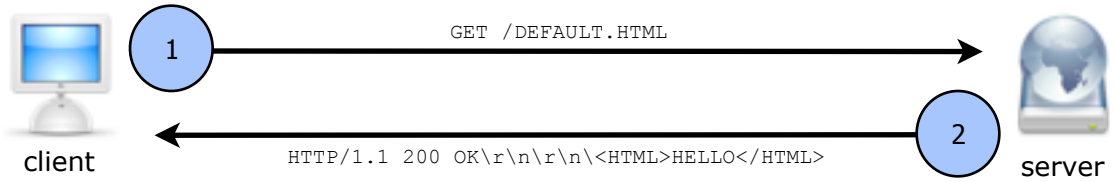

Figure 1. A static content request

In the figure above, the request for a static content page is handled by the processor (in this case, a web server) as a urinary parameter request, where the only parameter is the Uniform Resource Locator (URL) to the static resource "/DEFAULT.HTML". The processor simply retrieves the static content and returns it to the requester.

While static content is justified as a valuable resource in distributed hypermedia systems, at some point, dynamic content (derived from one or more parameters) is required to more fully exploit the functionality of a distributed hypermedia system. These requests are often assembled with parameters using a name-value pairing syntax such as "Name=Value" in the query portion of the request URL:

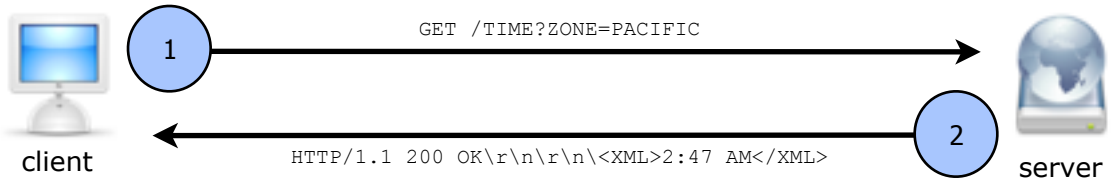

Figure 2. A name-value pair query content request

In the figure above, the name-value pair query is handled by the processor in some logical manner, and a dynamic result (which may or may not be composed of static content) is constructed and returned to the requester. In a content request query, one or more name-value pairs may be presented beginning with the question mark query designator, and delimited by ampersands. Typically, the parameters may appear in any order, and under normal conditions, the processor first analyzes all of the passed parameters, and then constructs a single resource as its response.

The name-value pairing syntax works well for autonomous requests that focus on retrieving a single resource from, or performing a single action (or intentional side-effect) on, a target system. However, this syntax begins to break down in more complex scenarios, often requiring a wasteful use of multiple requests, and an intermediary (script, language, or process) to compose new requests based on the results of previous requests.

The evolution of the distributed hypermedia system has led to complex processing environments, such as Web Services and Rich Internet Applications (RIA). These environments often employ heavy logical processing on the server, with light weight user interfaces on the client. Use of these environments all but guarantees a requirement that server and client maintain a "conversation" of requests and as a result, these environments also suffer from intermediary composition inefficiencies.

To help further explain the inefficiency of a typical client/server conversation, lets examine a Geographic Information System (GIS) and the sequence of events involved in calculating the number of buildings some distance away from a target building. When given the location of the target building in the form of a street address, assume it must first be "geocoded" (converted to an x,y coordinate) as the buffering process that determines the count of buildings at some distance, uses an x,y coordinate as its location parameter.

Figure 3. Target building buffered, showing four buildings at some distance

The required client/server conversation to achieve the desired result from *Figure 3* can be shown as:

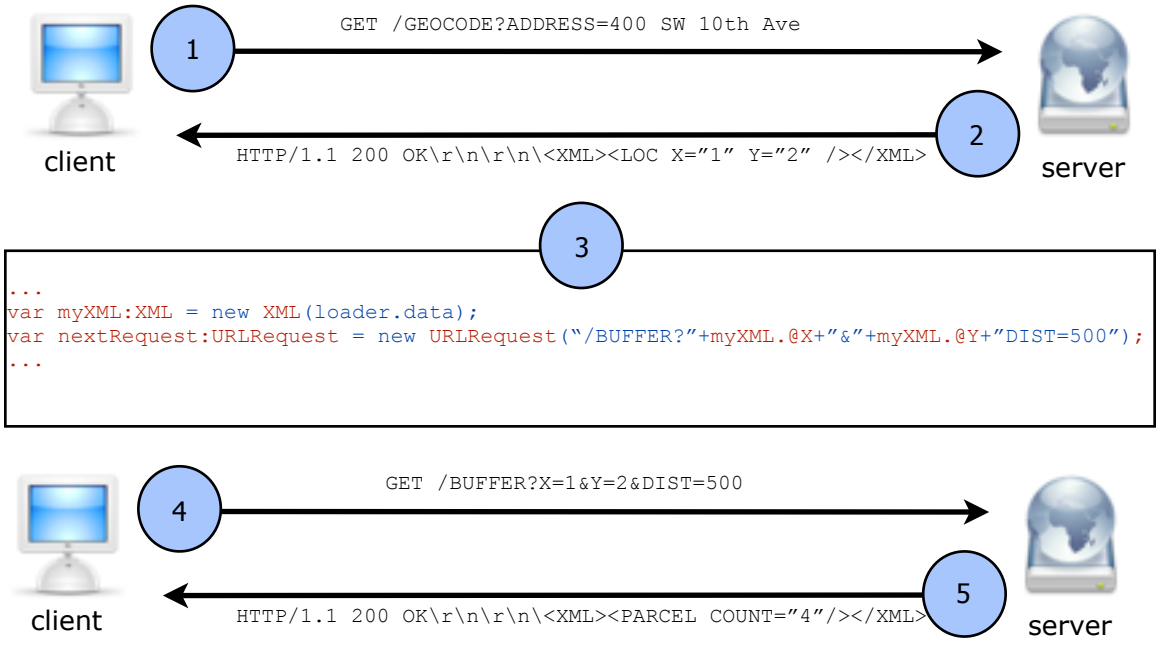

Figure 4. An example request conversion

As illustrated above, the result of the first request is not useful as a parameter to the second request until prepared (at least trivially) by an intermediary process (step 3). The nuisance of this approach is twofold: First, multiple requests are required even though no user interaction is. Second, at least some part of the entire request conversation must be "managed" outside of the URL to be useful; thus, the elegance of HTTP is lost by requiring us to step out of the protocol to "massage" the data, only to step back into the protocol to use the data.

The LAST architectural style is intended to correct conversational inefficiencies that heavy logic processing server systems impose on HTTP relating to network traffic "chatter", and the inherit requirement of an intermediary to compose useful request parameters. This is achieved by providing a functional syntax that allows for the bundling of multiple request actions, that result in reusable response elements, into a single request.

The syntactical apparatus defined by the LAST architectural style allows for the specification of one or more request actions in a single HTTP GET method request. Each of these request actions may (or may not) require parameters, and the result (or a sub element of the result) of an action may in turn, be used as a parameter in a subsequent request action of the same HTTP GET method request.

In the next section, *Defining LAST*, we break down the "LAST" acronym as well as give some examples of where LAST is useful--and perhaps more importantly, where it is not.

Defining LAST

The LAST architectural style offers a methodology by which multiple related, as well as autonomous request actions may be "bundled" into a single HTTP request for server side processing. In its simplest form, a LAST Application Programming Interface (API) request resembles a typical resource request, while more complex forms offer functionality not otherwise possible without client side intermediary processing.

The name "Lateral Stack Transfer" accurately describes this architectural style, as seen by examining each term individually:

Lateral – to move laterally or sideways; this term depicts the direction in which request query terms appear in an HTTP GET method request, for example:

```
GET /LAST-PROCESSOR?1=MAKEPOINT&X=10&Y=20
```

The name-value pairs 1=MAKEPOINT, X=10, and Y=20 are expressed laterally in the request above.

Stack – typically, a linear list arranged so that the last item stored is the first item retrieved; Here, this term refers to the list of name-value pairs which make up one or more request "actions" to be processed in sequence by "action index" (described later):

```
GET /LAST-PROCESSOR?1=TIME&ZONE=PACIFIC&2=TIME2&ZONE2=EASTERN
```

The request above packages two autonomous "time" request actions, the first for the Pacific Time zone, while the second is for the Eastern Time zone. Even though in the above example each request action is autonomous, it is important to note that the stacked nature of the LAST architectural style requires that the first action requested (indicated by the action index "1") be processed before the second (indicated by the action index "2"). The significance of this requirement will be explained shortly.

Transfer – to be moved from one place to another; denoting the action of conveying requests from client to server, and accepting results from server to client.

To further clearly define LAST, it is important at this point to discuss what LAST is not, and to expound on conditions to which a LAST API may be appropriate, compared to other API architectures. LAST is not intended to be a replacement for simple static request/response systems:

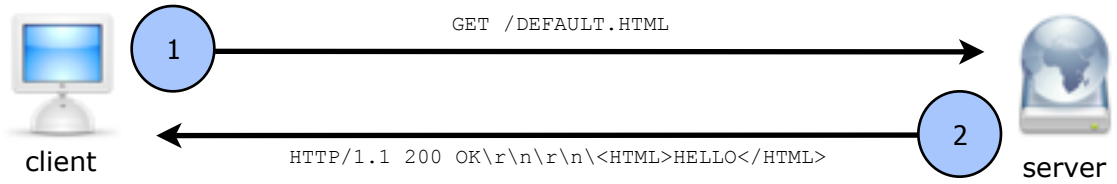

Figure 5. Simple static request/response system; bad LAST candidate

Nor, is the LAST architectural style extremely useful in purely autonomous request systems, where the requests do not relate or require parameters based on previous responses:

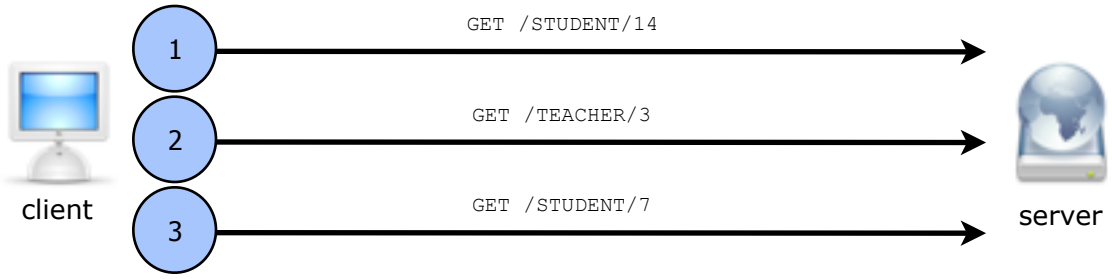

Figure 6. Unrelated requests; less than optimal candidate for LAST

The usefulness of the LAST architectural style applied to the situation above could be debated; as while the HTTP inefficiencies of sending multiple requests would be eliminated, some powerful elements of LAST are never used, such as its stacked nature and result interrogation qualities (both of which will be described shortly).

The most useful application for the LAST architectural style is to facilitate optimal use of enterprise systems that expose Web Services containing functionality that use custom types as input parameters, and have sequential, progressive, processing paths to an expected outcome or result.

Web based GIS is a perfect example of an enterprise system that can benefit from the implementation of a LAST API, satisfying both the dependence on custom types (a geographic point, for instance) and progressive processing path requirements; locate a parcel owned by a specify individual, *then* buffer that parcel some distance based on some criteria, *then* generate a map image of the target area with a graphical representation of the buffered area including a pushpin graphic at the centroid of the located parcel.

In the classic request/response architecture, the process described above would take several calls to the server, each requiring intermediate client processing:

1. Client constructs a request to retrieve the target parcel centroid X and Y coordinates, based on owner name criteria;
2. Server receives and processes the request, generating a point as the response;
3. Client stores the point result of the request through some scripting language;
4. Client constructs and sends a new request to buffer a certain distance from the X and Y components of the stored point variable;
5. Server receives and processes the request, generating a polygon “buffer zone” as the response;
6. Client stores the resulting polygon of the request via some scripting language;
7. Client constructs and sends a new request to generate a map image that fully contains the polygon resulting from the previous request;
8. Server receives and processes the request, returning a URL to the generated image;
9. Client stores the resulting URL through some scripting language;

10. Client constructs a new request to update the generated image with a pushpin graphic at the target parcel centroid;
11. Server receives and processes the request, returning a URL to the updated image;
12. Client receives and finally consumes the resulting image.

Using the LAST architectural style, the entire process is reduced to a single HTTP GET method request, composed of multiple, reusable actions;

1. Client constructs a LAST API request that generates a centroid based on owner name criteria, a buffer based on the centroid, an image based on the buffer, and a pushpin graphic embedding in an image, based on the first image and the original centroid;
2. Server receives and processes the request, returning a URL to the image;
3. Client receives and consumes the resulting image.

Compared to the classic request/response architecture, the LAST architecture style can significantly reduce both network "chatter", and intermediate client side processing when implemented for complex conversations.

In the next section, *Elements of a LAST Request*, we define "actions" and "parameters"--the two element types used to construct a LAST request.

Elements of a LAST Request

The name-value pair elements that follow the question mark in a LAST request, compose its "query" and are classified as either actions or parameters. These element classifications are explained in greater detail below:

Action – a job or task to be completed. Actions are the core elements that define what processing is required of the request. In a LAST request, all actions are specified using the following name-value pair convention:

action index=action name[group suffix]

"action index" is a unique (per request) numeric value, and is used to determine the order in which this action will be processed when multiple actions are specified in a single request. LAST processors process actions in ascending action index order, starting with the lowest value action index and completing with the highest.

"action name" is the name used to identify the action to be processed. This is comparable to a method name in an application programming interface, and instructs the LAST processor as to which task is should perform. Action names may consist of any valid URL safe characters, but may not end with a numeric value, as this conflicts with the optional group suffix notation described below.

"group suffix" is an optional grouping identifier consisting of a unique (per group) numeric value that is only required when the names of parameters for any two action calls conflict. This allows multiple instances of the same action, or different actions that have coincidental parameter names to be specified in the request without causing ambiguity. By using the same group suffix for the action name, and each of its parameters, each action (and its parameters) are "grouped", and thus, do not conflict with action calls having coincidental parameters.

For instance (line breaks have been added for clarity),

```
GET /LAST-PROCESSOR?
                    1=MAKEPOINT&X=10&Y=20&
                    2=BUFFER&ADDRESS=100 SW MAIN&DIST=500
```

Because the actions MAKEPOINT and BUFFER do not have coincidental parameter names, a group suffix is not required. However, if BUFFER had a second prototype using an X,Y coordinate instead of a street address,

```
GET /LAST-PROCESSOR?
                    1=MAKEPOINT1&X1=10&Y1=20&
                    2=BUFFER2&X2=50&Y2=400&DIST2=500
```

Here, the group suffix notation is required to avoid ambiguity between X's and Y's. It is important to note that this example the action index and group suffix are identical, but, this is purely coincidental as the same request could have been written,

```
GET /LAST-PROCESSOR?
                    1=MAKEPOINT42&X42=10&Y42=20&
                    2=BUFFER8&X8=50&Y8=400&DIST8=500
```

Also note when group suffix's are required, one instance of an action (or parameter) may exclude the group suffix and still maintain uniqueness as long as all other conflicting instances declare a group suffix. In the example below, action 1 (and its parameters) exclude a group suffix, but the syntax is still valid as actions 2 and 3 declare group suffix's.

```
GET /LAST-PROCESSOR?
                    1=MAKEPOINT&X=10&Y=20&
                    2=BUFFER40&X40=50&Y40=400&DIST40=500
                    3=MAKPOINT26&X26=300&Y26=750
```

Parameter – a piece of data used to "feed" the requirements of a request action. In a LAST request, all parameters are specified using the following name-value pair convention:

parameter name[group suffix]=value

"parameter name" is the name used to identify the parameter expected by the related action. This is comparable to a parameter name in an application programming interface, and "feeds" the LAST processor with information needed to complete the related action. Parameter names may consist of any valid URL safe characters, but may not end with a numeric value, as this conflicts with the optional group suffix notation described above.

"value" may be a constant value consisting of any valid URL safe characters, or an E4X reference to a result (or a sub-element of a result) of a previous action in the request stack.

E4X referencing of results is an important facet of the Lateral Stack Transfer architecture style which sets it apart from other URL based API constructions. Using this style not only allows multiple actions to be completed in a single request, but provides a method by which actions can work together in a single request to eliminate both network "chatter" and the need for intermediary processing.

In the next section, *Composing a LAST Request* we take a closer look E4X, and how it's used to isolate data in an action result for use as parameter data for subsequent actions of the same request.

Composing a LAST Request

To fully take advantage of the LAST architecture style, it may be necessary to "re-think" the typical HTTP conversation. While this task may seem daunting at first, once the syntax of this style becomes familiar, any type of conversation (from the simplest request to very complex calculations) can be expressed intuitively in a single request.

At the heart of this syntax, is the ability to isolate reusable element data from a result or a sub-element of a result of a previous action in the request stack. This is accomplished by using E4X notation to traverse the structure of a result "object", and thus imposes the following requirement on a LAST processor: The result of any action that may be used as a parameter to another action, must be expressed in XML.

This stipulation requiring XML results enables all actions to work together in a common format, and shifts any intermediate processing to a single final processing task by either the LAST processor (to return the final result of the final action as something other than XML), or the client (to extract the useful information out of the final XML structure).

The next example demonstrates a conversation between a client application and a LAST processor, in which the client desires a map image highlighting all of the buildings that surround another building and are within 500 feet. Let's assume the LAST processor interface to the GIS that houses this information supports the following actions:

GetBuildingID - This action retrieves the building ID from an assessor database by geocoding based on either an "Address" parameter (the address of the building), or "X" and "Y" parameters (any point that is within the building footprint). Calling an autonomous request of:

```
GET /LAST-PROCESSOR?1=GETBUILDINGID&ADDRESS="100 SW Main ST"
```

or, an equivalent form of,

```
GET /LAST-PROCESSOR?1=GETBUILDINGID&X=10&Y=20
```

returns the XML element,

```
<GETBUILDINGID ADDRESS="100 SW Main ST" X="10" Y="10" ID="62"/>
```

BuildingsThatSurround - This action retrieves a list building IDs from an assessor database by performing a spatial buffer of a target building, based on "ID" parameter, at some distance in feet based on a "DISTANCE" parameter. Calling an autonomous request of:

```
GET /LAST-PROCESSOR?1=BUILDINGSTHATSURROUND&ID=62&DISTANCE=500
```

returns the XML element,

```
<BUILDINGSTHATSURROUND ID="62" DISTANCE="500">
   <ID>21</ID>
   <ID>98</ID>
   <ID>416</ID>
 <BUILDINGSTHATSURROUND/>
```

DisplayBuildingMap - This action displays image of a map, centered on a building described by the “ID” parameter. An optional parameter, “HighlightBuildings” expects a list of one or more building IDs that will be highlighted on the map. Calling an autonomous request of:

```
GET /LAST-PROCESSOR?1=DISPLAYBUILDINGMAP&ID=62
```

returns actual image data,

In this example, the DisplayBuildingMap action is not intended for use by other actions, thus does expose its results in XML. Exposing the result of the final action in a LAST request as XML certainly has merit, however, it does require the client to manage the XML returned. For instance, The DisplayBulidingMap action could be extended with an additional “OUTPUT” parameter to support differing output types. Calling an autonomous request of:

```
GET /LAST-PROCESSOR?1=DISPLAYBUILDINGMAP&ID=62&OUTPUT=PATH
```

now returns the XML element,

```
<DISPLAYBUILDINGMAP ID=”62” OUTPUT=”PATH” PATH=”/TempImages/BuildingMaps/123.JPG”/>
```

instead of actual image data.

```
GET /LAST-PROCESSOR?
                    1=BUILDING&X=10&Y=20&
                    2=BUFFER40&X40=50&Y40=400&DIST40=500
                    3=MAKPOINT26&X26=300&Y26=750
```

References

[1] http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[2] http://www.w3.org/Protocols/rfc2616/rfc2616.html

[3] http://www.w3.org/TR/xml11/

[4] http://www.ecma-international.org/publications/standards/Ecma-357.htm