Hypermedia-Oriented Design

An Approach for Supporting Evolvable Distributed Network Applications

Mike Amundsen

September 2011

Introduction

This paper briefly reviews three common design patterns for distributed network applications and notes examples where these designs make supporting a system that evolves over time problematic. An alternative approach is presented which relies on the concept of "affordances" and Hypermedia Factors. Common use cases are cited to show that this alternative approach can successfully support evolving systems where existing client applications automatically incorporate the modifications without the need to be re-coded and re-deployed. Some areas of continued study are also identified.

A Review of Common Application Designs

Distributed network application implementations rely on a small number of well-known design approaches. This section focuses on three broad categories of design:

- RPC-Oriented Design
- Object-Oriented Design
- URI-Oriented Design

While each of these patterns has advantages, they all focus on sharing understanding between client and server by requiring both parties to use an implementation model based on a predefined list of procedure calls, object graphs, or URI-construction rules. In each of these cases, modifications to the operational elements over time (procedures, objects, URIs) has the potential to invalidate existing implementations and/or be ignored by them.

RPC-Oriented

In RPC-Oriented designs, clients and servers share a list of procedure signatures including the parameters and their data types. Clients are expected to understand the details of the procedure list and use that list to arrange a series of calls in order to complete a desired goal (create an invoice, register a new user, etc.). Servers are expected validate security when each procedure is called, confirm the call contains the proper arguments, and prevent clients from executing procedures out of order.

When server implementors update the list of procedures (add/remove procedures, modify the parameter list for a procedure, change the order in which procedures may be executed, etc.), client developers are expected to retrieve the new RPC list and modify existing code to accommodate the changes. Depending on the implementation details, existing client applications may be able to continue to operate w/o error, but will not support any of the modified calls.

In RPC-Oriented models, the client is "bound" to the procedure list.

Object-Oriented

Object-Oriented designs focus on making sure clients and servers share the same object model (objects, properties, hierarchies, etc.). Clients are expected to maintain a copy of the object model, serialize objects for submitting to the server, and recognize serialized objects sent from the server. It is up to the client to use this object model in order to reach a desired goal. Servers are expected to validate security when handling requests, confirm the schema of the serialized objects it receives, and prevent clients from passing object serializations out of order.

When the server implementation is updated (e.g. the object graph is modified, the order in which objects may be successfully processed changes, etc.), client code is expected to be modified in order to maintain a valid object

graph and change the order in which objects are submitted. It is possible existing clients will continue to function without supporting the changes.

In Object-Oriented designs, the client is "bound" to the object graph.

URI-Oriented

Designs which are URI-Oriented are ones where the shared understanding between client and server is focused on a list of URIs and/or URI-construction rules. This pattern relies on "meaningful" URIs that can reflect either an RPC-Oriented style (/customer/add, /invoice/approve, etc.) or an Object-Oriented style (/invoices/, /customers/, /customer/invoices, etc.). Clients are expected to understand how to construct one or more URIs, know how to compose a valid request body associated with the URI, know which response bodies are expected for each URI, and execute URI-based requests in series to achieve a desired goal. Servers are expected to reject client requests that do not follow the agreed URI construct rules, those that use an improper serialization in the body, are not properly authorized, and/or are not executed in proper order.

Updating the server implementation can result in modification to the URI construction rules (adding, removing, elements of construction, modifying request body requirements, and/or changing process ordering rules, etc) may invalidate existing implementations. Client are expected to maintain an up-to-date list of URI construction rules and associated request bodies in order to continue interacting with the server.

In URI-Oriented designs, the client is "bound" to URI construction rules and their associated payloads.

Evolving Common Implementations

In all three cases, client applications are expected to maintain their own copy of "action lists" (procedures, objects, URIs) and are expected to know which item to execute at what time. They receive little help from the server except to reject client requests that are improperly composed or appear in the wrong order.

Since client applications are "bound" to lists of possible actions, modifying the list of actions (by the server) can invalidate the existing client or at best "freeze" the client at a single implementation point. The most effective strategy for evolving these designs is for servers to declare a new "version" of the action list and establish rules that allow clients to recognize and select only "supported" versions. This versioning isolates the problem of invalidating existing client applications, but does not eliminate it.

The Hypermedia Alternative

In contrast to the previous methods of designing distributed network applications, there is an alternative approach that does not require the client and server to "bind" directly to first-class operational elements (e.g. procedures, objects, URIs). Instead, it is possible to allow clients and servers to share understanding via a set of protocol-level actions which afford both client and server the options needed to interact with each other. In this paper, this alternative is called "Hypermedia-Oriented Design."

Hypermedia-Oriented

Hypermedia-Oriented designs are based on a shared understanding of a defined set of hypermedia elements in representations. These elements allow clients to initiate requests to navigate to new states, embed responses within the current document, and transfer one or more data items to the server for processing. Clients are expected to recognize, parse, and (as needed) render and/or activate these hypermedia elements if and when they appear within a representation. Servers are responsible for emitting appropriate hypermedia affordances with each reply to the client. Servers are also expected to validate client requests for proper authentication, confirm request bodies are valid, and reject requests that cannot be processed.

It should be noted that, in a Hypermedia-Oriented design, clients and servers do not need to share object models, procedure details, or even URI patterns. All that is needed is that clients know at least one valid URI and that the client and server share an understanding of the possible hypermedia affordances that might appear.

For Hypermedia-Oriented designs, sever modifications (new procedures, new objects, new URIs, etc.) do not increase the risk of invalidating client applications. Servers are also free to change which affordances appear in

responses (or whether they appear at all) since clients are designed to simply recognize, parse, and present/activate affordances as they are encountered. The only risk to client applications is adding new, undefined affordances to responses. In such cases, clients will simply ignore them and continue processing responses anyway.

In Hypermedia-Oriented designs, the client is "bound" to the list of possible affordances which may appear in server responses.

Affordances, Factors, and Types

The primary elements in hypermedia designs are the affordances within each response representation. These affordances are elements that clients must know ahead of time and which provide clients the ability to retrieve data from, and/or send data to, another location.

In distributed network applications all hypermedia affordances have four common characteristics. These characteristics can be combined in various ways to produce Hypermedia Factors (H-Factors) which clients can easily recognize, parse and render/activate as needed.

When using HTTP as the transfer protocol, clients and servers negotiate for support of various hypermedia factors using MIME Media types. Not all media types support H-Factors. For example XML, JSON, and CSV do not contain native hypermedia elements. However, HTML, VoiceXML, and Atom are examples of MIME Media types with built-in hypermedia affordances. In Hypermedia designs, clients and servers agree upon which hypermedia sets (Media Types) can be used for client-server interaction.

Hypermedia Affordances

The concept of affordances was first outlined by James J. Gibson in his 1977 article "The Theory of Affordances." Later, in his 1986 book "The Ecological Approach to Visual Perception" Gibson wrote:

> "The affordances of the environment are what it offers ... what it provides or furnishes, either for good or ill. The verb 'to afford' is found in the dictionary, but the noun 'affordance' is not. I have made it up."[1]

Later, in 1988, Donald Norman used the term in his "Design of Everyday Things":

> "[T]he term affordance refers to the perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used."[2]

And, in 2008, Roy T. Fielding employed the term when describing his use of the word hypertext:

> "When I say Hypertext, I mean the simultaneous presentation of information and controls such that the information becomes the affordance through which the user obtains choices and selects actions."[3]

For the purposes of applying affordances to hypermedia, there are four important aspects to consider:

Mutability
: The affordance either supports mutability or it does not (i.e. it is immutable). For example, in HTML the `form` element affords mutability; the `link` element does not.

Presentation
: The affordance either presents the related hypermedia (the response from the target URI) as a navigational experience (i.e. HTML's a tag) or a transclusion experience (i.e. HTML's `img` tag).

Idempotence
: The affordance represents either an idempotent action or a non-idempotent action. For example, when the HTML `form` element has the `method` property set to `get` the element affords an idempotent action. When the same property is set to `post` the same element affords a non-idempotent action.

[1] "The Ecological Approach to Visual Perception", 1986, pg 126.
[2] "Design of Everyday Things", 1988, pg 9.
[3] "REST APIs Must Be Hypertext Driven", 2008, (blog)

Safety — The affordance represents either a "safe" or "unsafe" action. The HTTP protocol, for example, supports a number of "safe" actions including `HEAD`, and `GET`. The HTTP methods `PUT`, `POST`, and `DELETE` are categorized as "unsafe" actions.

Two of the aspects (Mutability and Presentation) are the responsibility of the client application (i.e. the client is expected to honor the presentation aspect of the affordance supplied by the server). The other to aspects (Idempotence and Safety) are the responsibility of the server (i.e. the server is expected to treat the client's request as "safe" when the client activates an affordance exhibiting that aspect, etc.).

Hypermedia Factors

Hypermedia Factors (H-Factors)[4] are document elements which exist within the MIME media type to provide hypermedia affordances. Each H-Factor has all four of the aspects described above (mutability, presentation, idempotence, and safety). Most H-Factors have static affordance aspects (i.e. the element always represents an immutable action) but some H-Factors may support inline settings (usually via properties or attributes) that support changing one or more affordance aspects at runtime (i.e the element may, at one point, represent a safe action and, at another point, the same element will represent an unsafe action). The details of what affordances appear in a MIME media type and how those affordances can be modified is a decision made by the media type designer.

Below is a list of H-Factors that may appear within a representation response. This sample list captures many common arrangements of affordance aspects and offers enough possibilities to support most distributed application needs.

Link Embed (LE) — Support for embedded links

```
<!-- HTML -->
<img src="..." title="company logo" />
```

Affordance Aspects: *Immutable, Transclusion, Idempotent, Safe*

Link Outbound (LO) — Support for out-bound navigational links

```
<!-- HTML -->
<a href="..." title="view search page">Search</a>
```

Affordance Aspects: *Immutable, Navigation, Idempotent, Safe*

Link Template (LT) — Support for templated queries

```
<!-- HTML -->
<form method="get" action="...">
  <label>Search term:</label>
  <input name="query" type="text" value="" />
  <input type="submit" />
</form>
```

Affordance Aspects: *Mutable, Navigation, Idempotent, Safe*

Link Non-Idempotent (LN) — Support for non-idempotent updates

```
<!-- HTML -->
<form method="post" action="..."/>
  <label>Keywords:</label>
  <input name="keywords" type="text" value="" />
  <input type="submit" />
</form>
```

Affordance Aspects: *Mutable, Navigation, Non-Idempotent, Unsafe*

Link Idempotent (LI) — Support for idempotent updates

```
<!-- Atom -->
<link rel="edit" href="..."/>
```

[4] "Building Hypermedia APIs with HTML5 and Node", Mike Amundsen, 2011, pg 13

Affordance Aspects: *Mutable, Navigation, Idempotent, Unsafe*

The examples given here are from HTML and Atom, but H-Factors can be found in many MIME Media Types including CSS, SVG, VoiceXML, etc. It is also possible to design new MIME Media Types which contain H-Factors.

Hypermedia Types

MIME media types which contain one or more native H-Factors can be called *Hypermedia Types*.[5] These are media types designed to support hypermedia affordances. Media types need not support all the H-Factors listed here or all the possible affordance combinations in order to be called Hypermedia Types. The existence of at least one element that provides a hypermedia affordance is all that is needed.,

Evolving Hypermedia-Oriented Implementations

Unlike the previously discussed design patterns where client and server share understanding through lists of procedures, objects, or URIs, Hypermedia-Oriented designs only require the client and server to agree upon the list of possible H-Factors to be supported. This means supporting an implementation where operational features evolve over time is usually more feasible.

For example, consider the following scenarios:

- The service adds a new procedure which allows client applications to pre-order upcoming products
- The service adds a new object (Customer Loyalty Services) to the application
- Due to growing popularity, the service must deploy a new set of servers and spread customers across the new server farm resulting in a new set of URIs for all operations.

In all the above cases, Hypermedia-Oriented designs would not require changes to the client application and all existing client applications can automatically support the new features. This is possible because the client applications are not bound to the list of procedures ("pre-order upcoming products"), objects ("Customer Loyalty Services"), or URIs. Instead the clients are bound to the H-Factors and these do not change when a new procedure, object, or URI is added to the system.

Continued Study

Two related areas not yet fully explored are automated user-agents ('bots') and implementations where it is advantageous to retain an existing data-oriented message format and add affordances via an additional related document format.

Automated User Agents and Hypermedia

In human-driven cases, the knowledge of the problem domain resides with the person "driving" the client application. When presented with unexpected affordances in a response representation, it is likely that the human will be able to understand the new options and, based on knowledge of the domain space, make appropriate selections.

In the case of machine-to-machine interactions, unexpected affordances may not be properly understood and the agent may either make a poor choice or simply stop processing altogether. One area worth exploring is the possibility of encoding additional domain knowledge within the agent application and associating that domain knowledge with unexpected affordances in order improve support for evolvability over time.

Implementing Hypermedia as a Separate Document

There is class of commonly-used media types that are not particularly "hypermedia friendly." These are primarily data-oriented media types (CSV, n3, Turtle, etc.). It is not yet clear whether these formats could accommodate Hypermedia-Oriented designs as described here or even whether this is desirable.

[5] "REST: Research to Practice", 2010, Ch. 2 "Hypermedia Types", pg 97

One possible approach is to keep the existing media type focused on delivering data and design a secondary media type that could hold the affordance descriptions. This affordance information could then be "applied" to the associated data representation in a manner similar to the way CSS is used today to modify user interfaces. One of the advantages of this approach is that it allows existing (non-hypermedia) client applications to continue operating as they do today.

Summary

This short paper identified three common patterns employed when designing and implementing distributed network applications (RPC-Oriented, Object-Oriented, URI-Oriented). The paper also introduced an alternative approach (Hypermedia-Oriented) which relies on MIME Media Types which support affordances and H-Factors within the response representation. Some examples where cited where this alternative approach can support safely evolving distributed network applications without the need for re-coding and re-deploying client applications.