WHITE PAPER

Cookies, Sessions, and Persistence

UPDATED JANUARY 19, 2018

Introduction

HTTP (HyperText Transfer Protocol) was designed to support a stateless, request-response model of transferring data from a server to a client. Its first version, 1.0, supported a purely 1:1 request to connection ratio (that is, one request-response pair was supported per connection).

Version 1.1 expanded that ratio to be N:1—that is, many requests per connection. This was done to address the growing complexity of web pages, including the many objects and elements that need to be transferred from the server to the client.

With the adoption of 2.0, HTTP continued to support a many-request-per-connection model. Its most radical changes involve the exchange of headers and a move from text-based transfer to binary.

Somewhere along the line, HTTP became more than just a simple mechanism for transferring text and images from a server to a client; it became a platform for applications. The ubiquity of the browser, cross-platform nature, and ease with which applications could be deployed without the heavy cost of supporting multiple operating systems and environments was certainly appealing. Unfortunately, HTTP was not designed to be an application transport protocol. It was designed to transfer documents. Even modern uses of HTTP such as that of APIs assume a document-like payload. A good example of this is JSON, a key-value pair data format transferred as text. Though documents and application protocols are generally text-based, the resemblance ends there.

Traditional applications require some way to maintain their state, while documents do not. Applications are built on logical flows and processes, both of which require that the application know where the user is at the time, and that requires state. Despite the inherently stateless nature of HTTP, it has become the de facto application transport protocol of the web. In what is certainly one of the most widely accepted, useful hacks in technical history, HTTP was given the means by which state could be tracked throughout the use of an application. That "hack" is where sessions and cookies come into play.

Sessions

Transforming the Stateless into the Stateful

Sessions are the way in which web and application servers maintain state. These simple chunks of memory are associated with every TCP connection made to a web or application server, and serve as in-memory storage for information in HTTP-based applications.

When a user connects to a server for the first time, a session is created and associated with that connection. Developers then use that session as a place to store bits of application-relevant data. This data can range from important information such as a customer ID to less consequential data such as how you like to see the front page of the site displayed.

The best example of session usefulness is shopping carts, because nearly all of us have shopped online at one time or another. Items in a shopping cart remain over the course of a "session" because every item

customization applications. These "mini" applications enable you to browse a set of options and select them; at the end, you are usually shocked by the estimated cost of all the bells and whistles you added. As you click through each "screen of options," the other options you chose are stored in the session so they can be easily retrieved, added, or deleted.

Modern applications are designed to be stateless, but their architectures may not comply with that principle. Modern methods of scale often rely on architectural patterns like sharding, which requires routing requests based on some indexable data, like username or account number. This requires a kind of stateful approach, in that the indexable data is carried along with each request to ensure proper routing and application behavior. In this regard, modern “stateless” applications and APIs often require similar care and feeding as their stateful predecessors.

The problem is that sessions are tied to connections, and connections left idle for too long time out. Also, the definition of "too long" for connections is quite a bit different than when it is applied to sessions. The default configuration for some web servers, for example, is to close a connection once it has been idle—that is, no more requests have been made—for 15 seconds. Conversely, a session in those web servers, by default, will remain in memory for 300 seconds, or 5 minutes. Obviously the two are at odds with one another, because once the connection times out, what good is the session if it's associated with the connection?

You might think you could simply increase the connection time-out value to match the session and address this disparity. Increasing the time-out means that you're potentially going to expend memory to maintain a connection that may or may not be used. This can decrease the total concurrent user capacity of your server as well as ultimately impede its performance. And you certainly don't want to decrease the session timeout to match the connection time out, because most people take more than five minutes to shop around or customize their new toy.

Important note: While HTTP/2 addresses some of these issues, it introduces others related to maintaining state. While not required by the specification, major browsers only allow HTTP/2 over TLS/SSL. Both protocols require persistence to avoid the performance cost of renegotiation, which in turn requires session awareness, a.k.a stateful behavior.

Thus, what you end up with is sessions that remain as memory on the server even after their associated connections have been terminated due to inactivity, chewing up valuable resources and potentially angering users for whom your application just doesn't work.

Luckily, this problem is solved through the use of cookies.

Cookies

The Trail of Crumbs Leads Home

Cookies are bits of data stored on the client by the browser. Cookies can, and do, store all sorts of interesting tidbits about you, your applications, and the sites you visit. The term "cookie" is derived from "magic cookie," a well-known concept in UNIX computing that inspired both the idea and the name. Cookies are created and shared between the browser and the server via the HTTP Header, Cookie.

```
  Cookie: JSESSIONID=9597856473431 Cache-Control: no-cache Host:
127.0.0.2:8080 Connection: Keep-Alive
```

The browser automatically knows it should store the cookie in the HTTP header in a file on your computer, and it keeps track of cookies on a per-domain basis. The cookies for any given domain are always passed to the

server-side of the application.

The session/connection length problem is solved is through a cookie. Almost all modern web applications generate a "session ID" and pass it along as a cookie. This enables the application to find the session on the server even after the connection from which the session was created is closed. Through this exchange of session IDs, state is maintained even for a stateless protocol like HTTP. But what happens when the use of a web application outgrows the capability of a single web or application server? Usually a load balancer, or in today's architectures an Application Delivery Controller (ADC), is introduced to scale the application such that all users are satisfied with the availability and performance.

In modern applications, cookies may still be used but other HTTP headers become critical. API keys for authentication and authorization are often transported via an HTTP header, as well as other custom headers that carry data necessary for routing and proper scale of the backend services. Whether using the traditional “cookie” to carry this data or some other HTTP header is less important than recognizing its importance to the overall architecture.

The problem with this is load balancing algorithms are generally concerned only with distributing requests across servers. Load balancing techniques are based on industry standard algorithms like round robin, least connections, or fastest response time. None of them are stateful, and it is possible for the same user to have each request made to an application be distributed to a different server. This makes all the work done to implement state for HTTP useless, because the data stored in one server's session is rarely shared with other servers in the "pool."

This is where the concept of persistence comes in handy.

Persistence

The Tie that Binds

Persistence—otherwise known as stickiness—is a technique implemented by ADCs to ensure requests from a single user are always distributed to the server on which they started. Some load balancing products and services describe this technique as “sticky sessions”, which is a completely appropriate moniker.

Persistence has long been used in load balancing SSL/TLS-enabled sites because once the negotiation process—a compute intensive one—has been completed and keys exchanged, it would significantly degrade performance to start the process again. Thus, ADCs implemented SSL session persistence to ensure that users were always directed to the same server to which they first connected.

Over the years, browser implementations have necessitated the development of a technique to avoid costly renegotiation of those sessions. That technique is called cookie-based persistence.

Rather than rely on the SSL/TLS session ID, the load balancer would insert a cookie to uniquely identify the session the first time a client accessed the site and then refer to that cookie in subsequent requests to persist the connection to the appropriate server.

The concept of cookie-based persistence has since been applied to application sessions, using session ID information generated by web and application servers to ensure that user requests are always directed to the same server during the same session. Without this capability, applications requiring load balancing would need to find another way to share session information or resort to increasing session and connection time outs to the point that the number of servers needed to support its user base would quickly grow unmanageable.

Although the most common form of persistence is implemented using session IDs passed in the HTTP header, ADCs today can persist on other pieces of data as well. Any data that can be stored in a cookie or derived from the IP, TCP, or HTTP headers can be used to persist a session. In

the browser and a server.

Conclusion

HTTP may be a stateless protocol, but we have managed to force-fit state into the ubiquitous protocol. Using persistence and Application Delivery Controllers, it is possible to architect highly available, performant web applications without breaking the somewhat brittle integration of cookies and sessions required to maintain state.

These features are what give HTTP state, though its implementation and execution remain stateless. Without cookies, sessions, and persistence, we surely would have found a stateful protocol on which to build our applications. Instead, features and functionality found in Application Delivery Controllers mediate between browsers (clients) and servers to provide this functionality, extending the useful of HTTP beyond static web pages and traditional applications to modern microservices-based architectures and the digital economy darling, the API.

[redacted]