Application Flow Control with iRules – HTTP

James Quinby– MC

David Larsen– Presenter

Stephen Anderson – Presenter

John Eudy - Presenter

John Alam – Presenter

Intro:

WHY HTTP IRULES?

WHAT'S IN IT FOR ME?

WHO ARE YOU TO TELL ME HOW TO DO IRULES AND IS THIS WORTH MY TIME?

Agenda: Application Flow Control with iRules - HTTP

- •iRules Level Set
- •HTTP Protocol Review
- •HTTP Request Side Overview
- •HTTP Response Side Overview
- •HTTP Related Events
- •HTTP Headers
- •STREAM Command

iRules: Level Set

- **First rule of iRules – make sure you need an iRule.**
- **Comment code**
- **Use appropriate looping**
- **Use efficient criteria selection methods**
- **Use datagroups**

Protocol

THIS JOKE IS FUNNY BECAUSE THERE IS A MINION
YOU MUST LAUGH NOW IT IS PROTOCOL

The HyperText Transfer Protocol

- HTTP is the ***stateless***, text-based protocol responsible for the ***World Wide Web***, and has just one purpose: the transfer and delivery of HTTP messages.

- The basic message units are ***requests*** and ***responses***.

- Inside those messages are:

  `Methods / Status / Addressing / Version`

  `Headers`

  `Payload`

The HyperText Transfer Protocol

- Let's start by reviewing a basic HTTP message transfer and delivery.

- A client will make a request:

```
GET /index.html HTTP/1.1
Host: www.example.com
Accept: text/html
Accept-Encoding: x-zip; x-compress
User-Agent: libwww/1.3.1
If-Modified-Since: Thu, 07-Apr-2011 12:00:00 GMT
```

- And a server will respond:

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Server: Apache
Date: Thu, 07 Apr 2011 19:10:53 GMT
Content-Encoding: gzip

<html><head><title>…
```

The HyperText Transfer Protocol: request

- To retrieve a document (via request message) from the following resource:

$$\underset{\text{scheme}}{\texttt{http:}}\underbrace{\texttt{//x.y.com}}_{\text{address}}\underbrace{\texttt{/path/file.html}}_{\text{path/resource}}\underbrace{\texttt{?user=123\&is=me}}_{\text{query string}}\underbrace{\texttt{\#sometimes}}_{\text{fragment}}$$

- Your HTTP (browser) client will generate a request message:

$$\underbrace{\texttt{GET}}_{\text{method}}\ \underbrace{\underbrace{\texttt{/path/file.html}}_{\text{path}}\underbrace{\texttt{?user=123\&is=me}}_{\text{query}}}_{\text{uri}}\ \underbrace{\texttt{HTTP/1.1}}_{\text{version}}$$

The HyperText Transfer Protocol: request

- This resource request line will be followed by a series of ***headers*** - directives in name/value pairs that tell the server about the client.

```
GET /path/file.html?user=123&is=me HTTP/1.1
Host: x.y.com
User-Agent: libwww/1.3.0
Accept: text/html
Accept-Encoding: gzip, deflate
Keep-Alive: 300
Cookie: mycookie=12345
If-Modified-Since: Thu, 07 Apr 2011 12:00:00 GMT
```

The HyperText Transfer Protocol: request

- And if there's any payload data, that will commence after a single empty line.

```
GET /path/file.html?user=123&is=me HTTP/1.1
Host: x.y.com
User-Agent: libwww/1.3.0
Accept: text/html
Accept-Encoding: gzip, deflate
Keep-Alive: 300
Cookie: mycookie=12345
If-Modified-Since: Thu, 07 Apr 2011 12:00:00 GMT

# Other request methods will include payload data at this point
```

The HyperText Transfer Protocol: request

- Request for Comments (RFC) 2616 defines 8 HTTP request methods.

```
GET, POST, PUT, DELETE, TRACE, HEAD, OPTIONS, CONNECT
```

- A ***GET*** request is generally accompanied by an empty body because the request URI has enough information to complete the method. Semantically, a GET means to ***give me a resource***.

- A ***POST*** request typically contains a body of URL-encoded name/value pairs and does not usually contain large query strings. Semantically, a POST means to ***put a resource***.

```
POST /formlogin.php HTTP/1.1
<headers>

abc=123&is=me&you=arecool
```

The HyperText Transfer Protocol: response

- The web server then responds with an HTTP response message:

$$\underbrace{\underbrace{\texttt{HTTP/1.1}}_{\text{version}}\ \underbrace{\texttt{200 OK}}_{\text{status code}}}_{\text{status line}}$$

- The status code indicates to the client how the request was processed, and optionally gives an indication for further actions.

The HyperText Transfer Protocol: response

- This response status line will be followed by a series of ***headers*** - directives in name/value pairs that tell the client about the response.

```
HTTP/1.1 200 OK
Date: Thu, 07 Apr 2011 19:12:35 GMT
Server: Apache/2.2.9
Last-Modified: Thu, 31 Mar 2011 12:32:00 GMT
Content-Length: 3600
Content-Type: text/html
Content-Encoding: gzip
Set-Cookie: mycookie=12345; path=/;
```

The HyperText Transfer Protocol: response

- And if there’s any payload data, that will commence after a double CRLF.

```
HTTP/1.1 200 OK
Date: Thu, 07 Apr 2011 19:12:35 GMT
Server: Apache/2.2.9
Last-Modified: Thu, 31 Mar 2011 12:32:00 GMT
Content-Length: 3600
Content-Type: text/html
Content-Encoding: gzip
Set-Cookie: mycookie=12345; path=/;

<html><head><title>Test Page</title></head><body>…
```

The HyperText Transfer Protocol: response

- Per the HTTP RFCs, there are a fair number of response codes. The following are a small few:

```
100 Continue
200 OK
301 Moved Permanently
302 Moved Temporarily
304 Not Modified
404 Not Found
500 Internal Server Error
```

and

Events

Traffic Flow Occurrences

Direct and Manage traffic chronologically

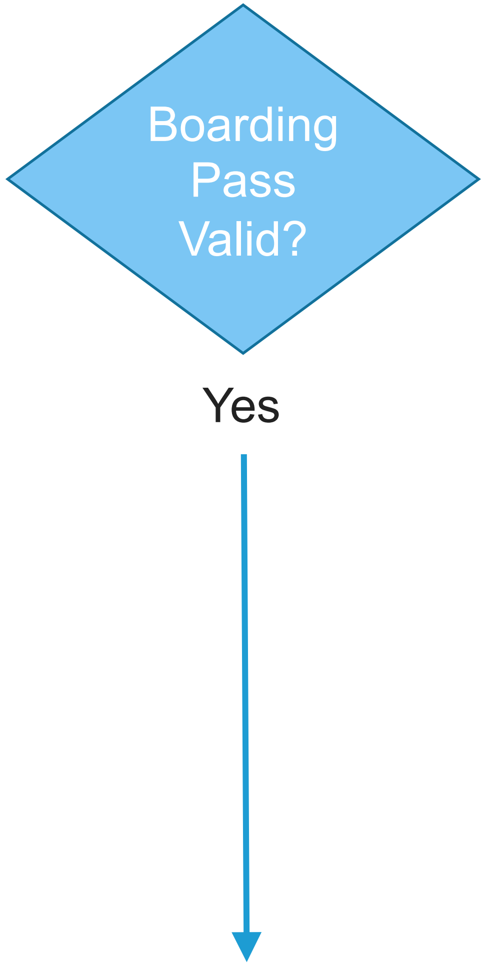
Dynamic decision making based on the state of the Flow.

Where else might we encounter traffic FLOWS?

Does the traffic follow sequential steps?

Let's look at something we know very well.

American


Boarding Pass Valid?
Yes

Special handling?

Yes

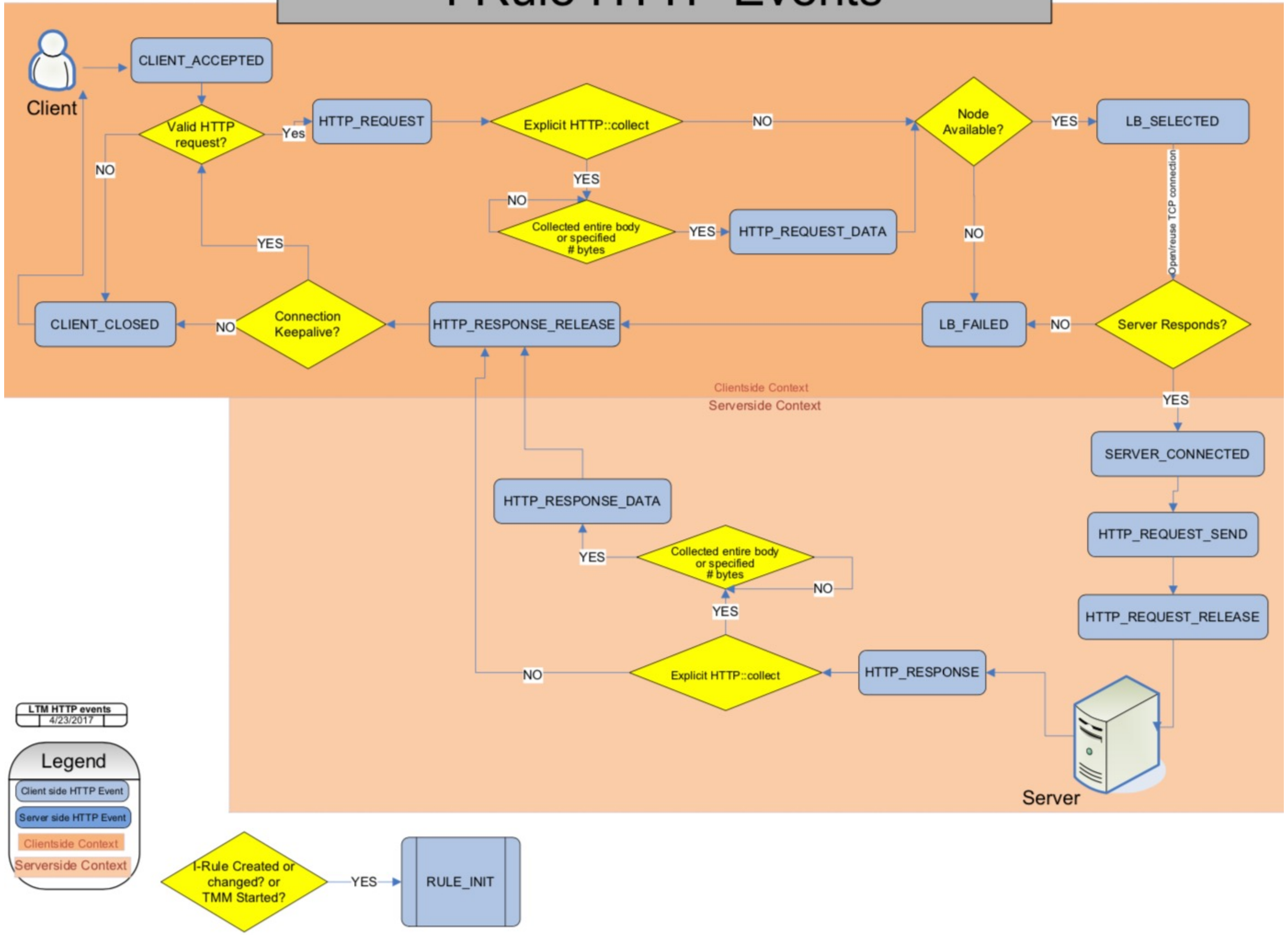
Next connection found?

Yes

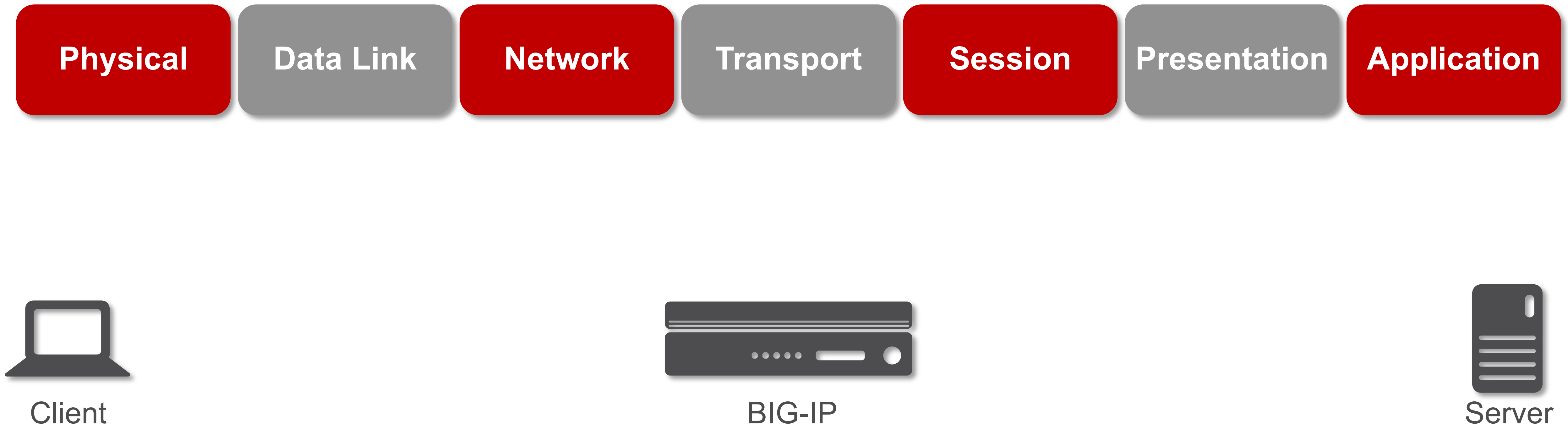
Connection made?

Yes

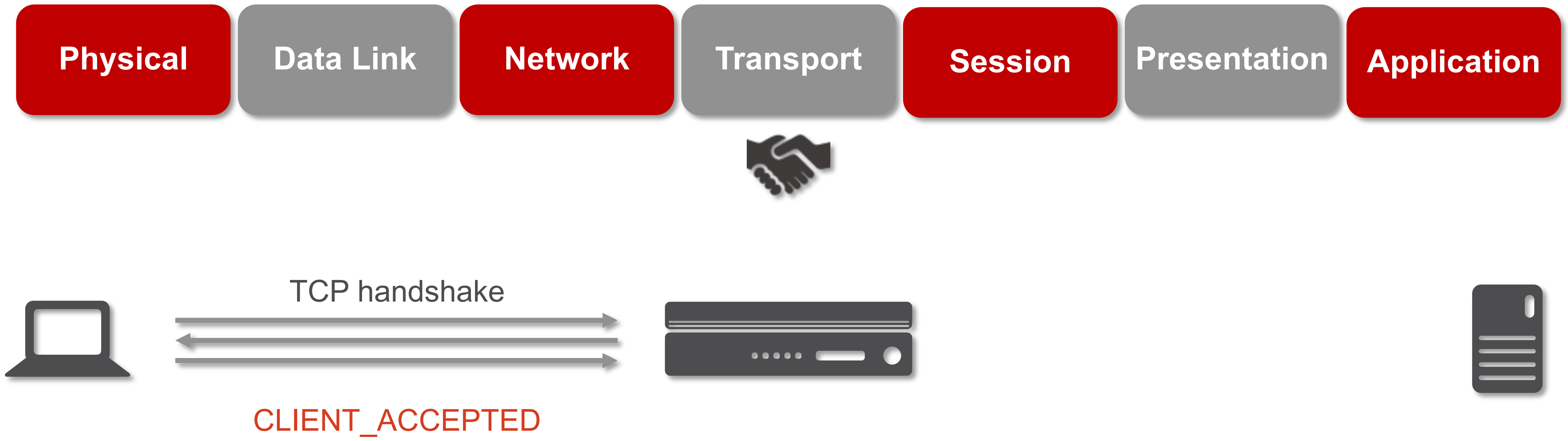
I-Rule HTTP Events



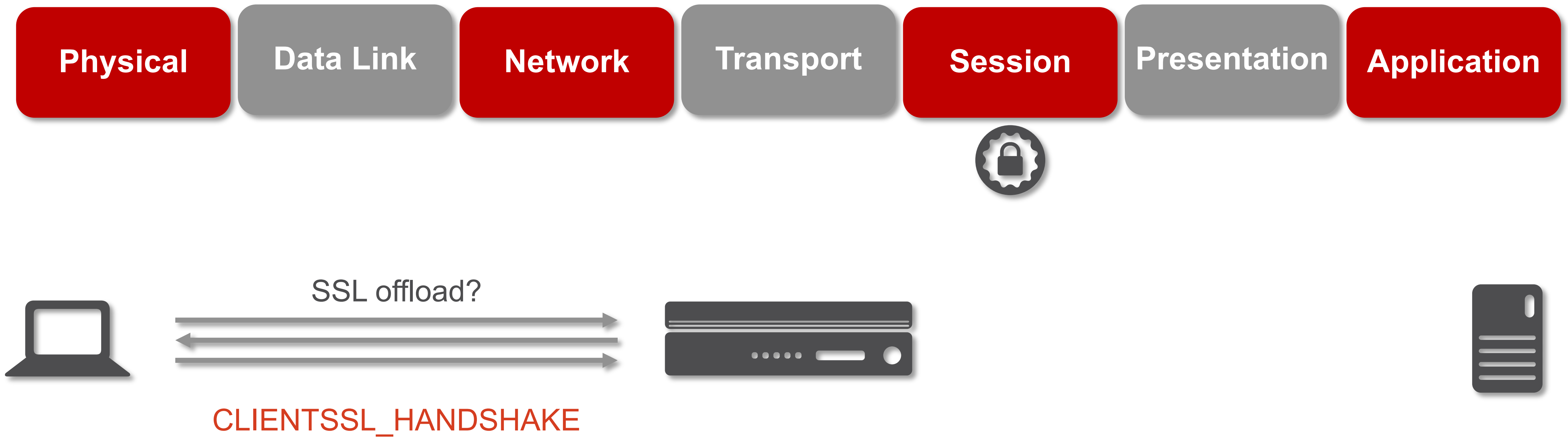
Example: the event life cycle of an HTTP dialog



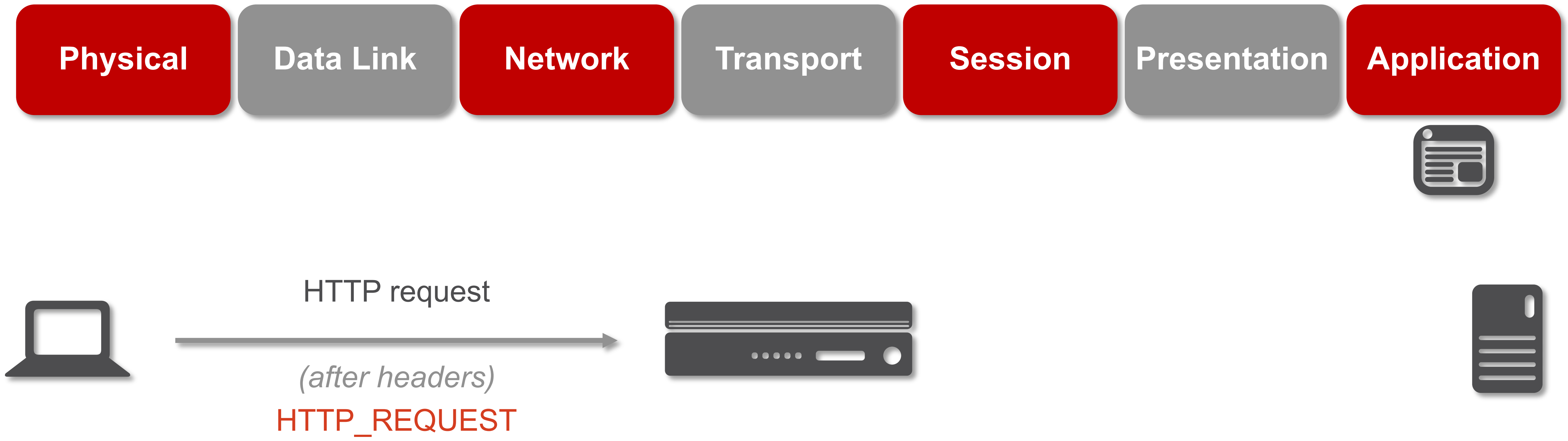
Example: the event life cycle of an HTTP dialog



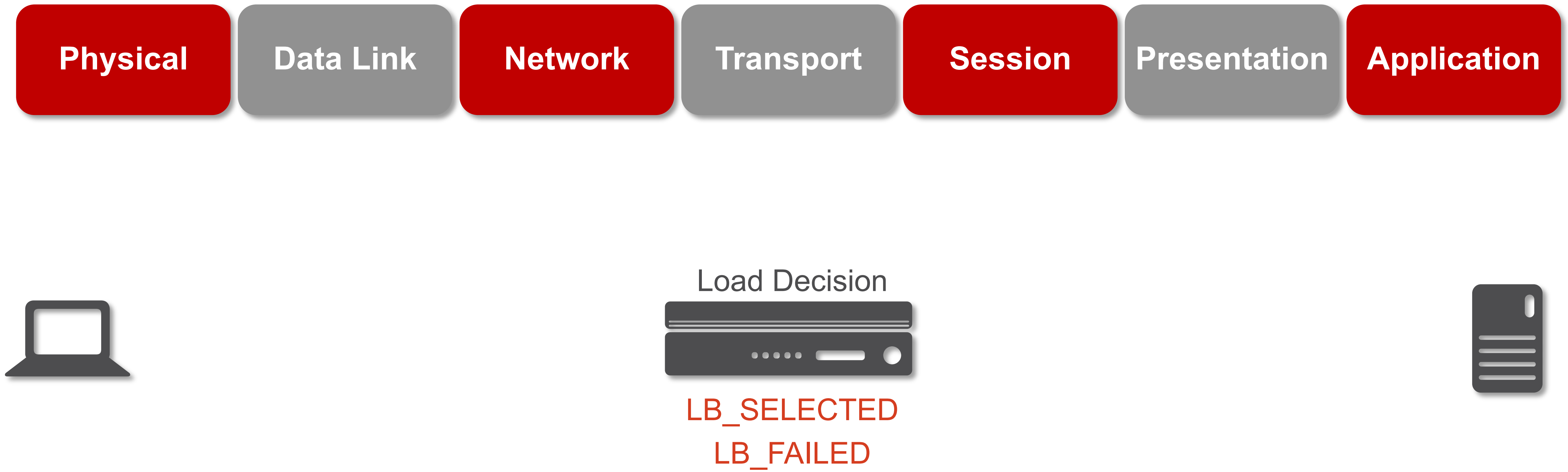
Example: the event life cycle of an HTTP dialog



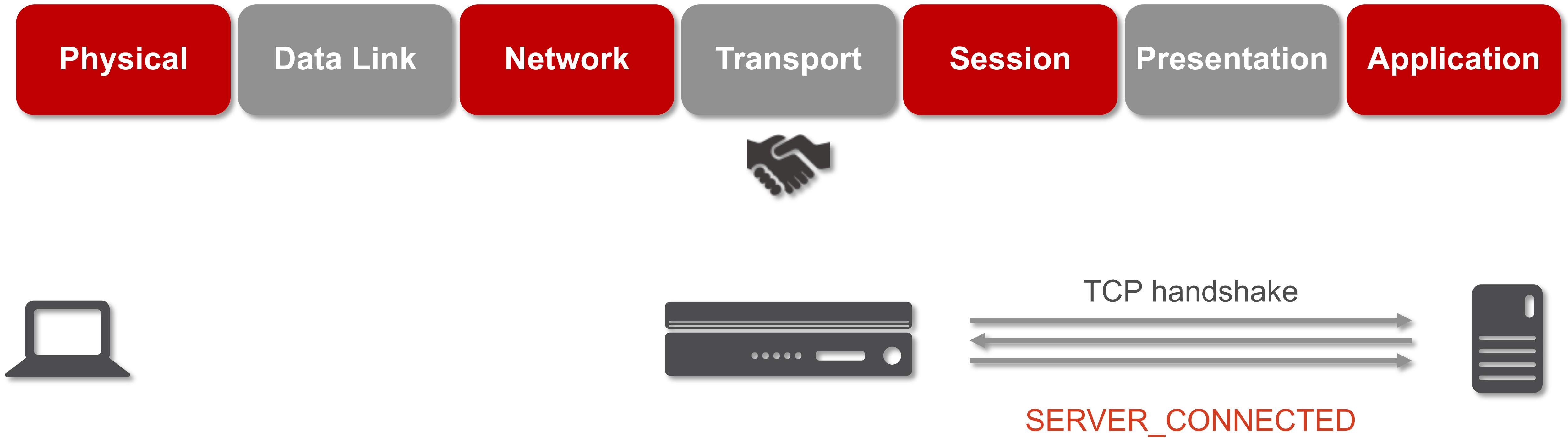
Example: the event life cycle of an HTTP dialog



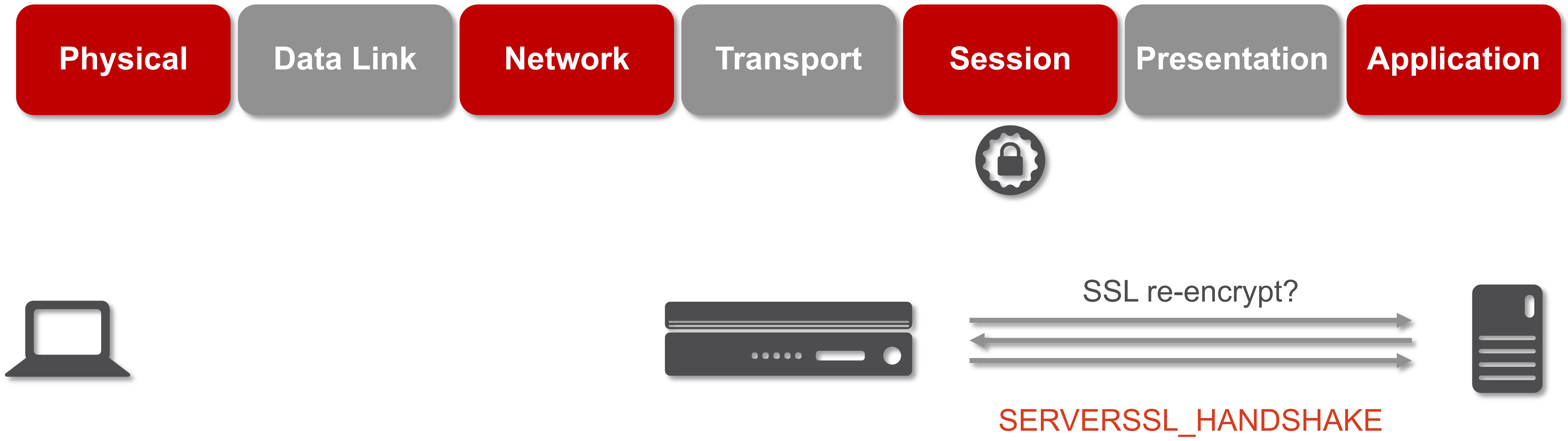
Example: the event life cycle of an HTTP dialog



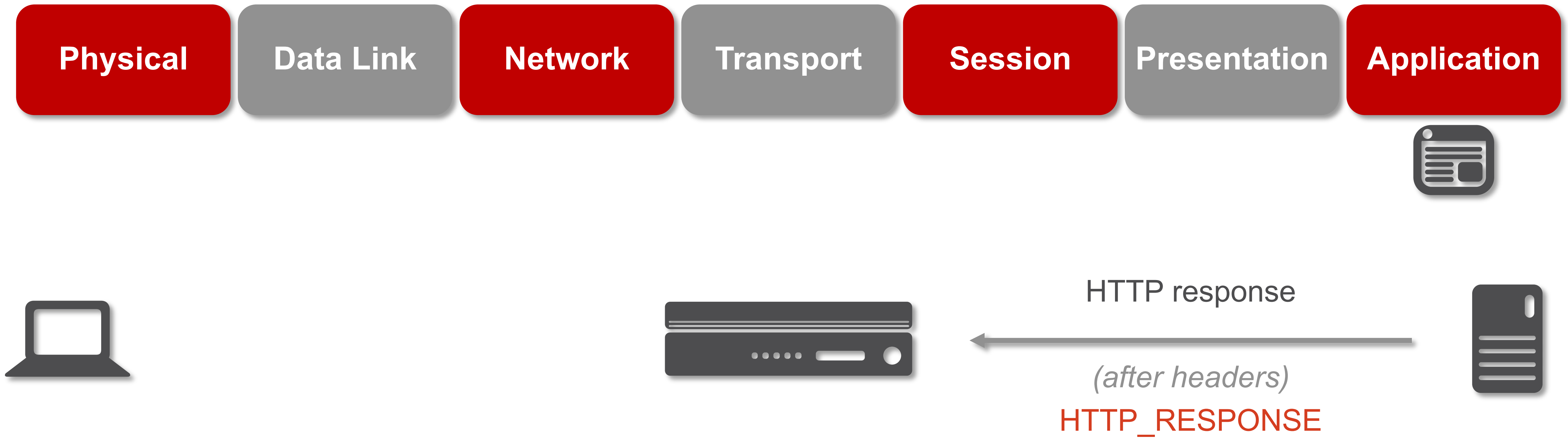
Example: the event life cycle of an HTTP dialog



Example: the event life cycle of an HTTP dialog



Example: the event life cycle of an HTTP dialog



The HyperText Transfer Protocol: cookies

- A cookie is a piece of text data sent by a server to a client to associate information with that client. When the server sets the cookie in a response, it specifies a name/value pair, and optionally a set of ***scope*** parameters:

```
Set-Cookie: myck=1; path=/; domain=y.com; secure; httponly
Set-Cookie: myck=1; expires=Thu, 01 Jan 1970 00:00:00 GMT
```

- Once set by the server, and until it either expires or is removed, the client will send the cookie back to the server on ***every request***:

```
Cookie: myck=1
```

The HyperText Transfer Protocol: cookies

- Cookie ***scope*** is defined by a set of optional parameters in the Set-Cookie header. These instruct the client on how and when to return the cookie.

```
path=/foo
domain=f5.com
secure
httponly
expires=Thu, 01 Jan 1970 00:00:00 GMT
```

iRules HTTP commands

- By far, one of the BIG-IP's greatest assets is its extremely rich set of events and tools to handle HTTP traffic. There are 10 events and 26 HTTP-specific commands. Starting with ***request*** events:

```
HTTP_REQUEST – triggered when the system fully parses the
complete client HTTP request headers

HTTP_REQUEST_DATA – triggered when an HTTP::collect command
has collected the specified amount of data

HTTP_REQUEST_SEND – triggered immediately before an HTTP
request is sent to the server side TCP stack
```

iRules HTTP commands

- Then there are the ***response*** events:

HTTP_RESPONSE – triggered when the system parses all of the response status and header lines from the server response

HTTP_RESPONSE_DATA – triggered when an HTTP::collect command has collected the specified amount of data

iRules HTTP commands

- And since there are so many HTTP commands, we’ll just look at a few examples. ***HTTP::header*** can read and write HTTP header data.

```
when HTTP_REQUEST {
    if { [HTTP::header Host] starts_with "foo" } {
        pool foo_pool
    }
}
when HTTP_RESPONSE {
    foreach aHeader { Server X-Powered-By } {
        HTTP::header remove $aHeader
    }
    HTTP::header insert X-Local-Port \
                  [clientside [IP::local_port]]
}
```

iRules HTTP commands

- Be mindful of the context in which a command is used. For example, the ***HTTP::header*** command used in requests and responses will produce different results.

```
when HTTP_REQUEST {
    set sessionid [HTTP::cookie "appid"]
    HTTP::header insert "X-Session-Id" $sessionid
}
when HTTP_RESPONSE {
    if { $sessionid eq "" } {
       if { [HTTP::header exists "X-Session-Id"] } {
           HTTP::cookie insert X-Session-Id \
                        [HTTP::header X-Session-Id]
       }
    }
}
```

iRules HTTP commands

- The ***HTTP::redirect*** command redirects an HTTP request or response to the specified URL.

```
when HTTP_REQUEST {
    if { [HTTP::uri] starts_with "/auth" } {
        HTTP::redirect "https://auth.y.com"
    } elseif { [HTTP::uri] starts_with "/images" } {
        HTTP::uri "/imagesrv"
        pool image_pool
    }
}
```

iRules HTTP commands

- The ***HTTP::cookie*** command can read and write HTTP cookie data.

```
when HTTP_REQUEST {
    if { [HTTP::cookie exists "oldcookie"] } {
        set cookievalue [HTTP::cookie value "oldcookie"]
        HTTP::cookie insert name "newcookie" value $cookievalue
        HTTP::cookie remove "oldcookie"
    }
}
```

iRules HTTP commands: getting in the game

- Because everything else wasn't cool enough, we also have the ***HTTP::respond*** command, which lets you respond to a client request *on behalf* of the server. Some possibilities include:

  `Static HTML`

  `JavaScript/CSS`

  `Dynamic information`

  `More flexible redirects`

  `Adding/removing application functionality`

  `Enhanced and extremely flexible authentication`

  `No need for an actual server!`

iRules HTTP commands: getting in the game

- Let’s look at an example.

```
when HTTP_REQUEST {
    if { [active_members mypool] < 1 } {
        switch [string tolower [HTTP::uri]] {
            "/logo.png" {
                HTTP::respond 200 content [ifile get logo] "Content-Type" "image/png"
            }
            default {
                HTTP::respond 400 content "…maintenance page HTML…"
            }
        }
    }
}
```

$$\texttt{HTTP::respond}\ \underbrace{\texttt{200}}_{\text{code}}\ \underbrace{\texttt{content}}_{\text{keyword}}\ \underbrace{\texttt{\{<html>…</html>\}}}_{\text{content}}\ \underbrace{\texttt{"header name" "header value"}}_{\text{optional headers}}$$

DevCentral is…

Resources

Wiki-based access to F5 product and API documentation.

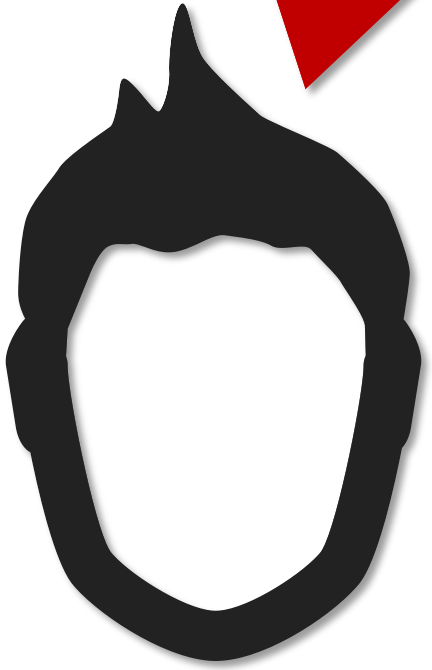
https://clouddocs.f5.com

Forums

A place to ask and answer F5 product and API technology questions.

150,000+

and growing community.



https://devcentral.f5.com