# C

# HTTP Primer

This appendix introduces the Hyper Text Transfer Protocol (HTTP) [12]. HTTP is the fundamental interaction protocol powering the World Wide Web (WWW). As simple as it is elegant, HTTP is a flexible request-response protocol that provides machine- and location-independent access to remote resources. HTTP appears in modern speech processing systems in several places. Within the MRCP protocol, HTTP provides a mechanism for retrieving data such as speech grammars, audio files and speaker voiceprints, and pushing data such as audio recordings to remote Web servers. HTTP is also fundamental in enabling the VoiceXML architectural model by allowing a clean separation of the application logic from the implementation platform. Last but not least, the structure of the HTTP protocol has significantly influenced the design of both the MRCP protocol and the Session Initiation Protocol (SIP) - a protocol that is in turn used by the MRCP framework for session management. An understanding of HTTP is therefore not only valuable in itself but also helpful for newcomers seeking to understand the MRCP and SIP protocols.

## C.1   Background

HTTP 0.9 dates back to 1991 when it was created by Tim Berners-Lee (accredited with inventing the Web). The original protocol was very basic and involved setting up a TCP connection to the server and issuing a single verb called GET (all encoded in ASCII text) followed by the name of the resource. The response was a HTML document. HTTP has since then evolved significantly to include header fields to parameterise the request, content independence (no restriction to HTML), additional verbs that allow data to also be pushed, and caching and connection persistency to improve performance. The current version of HTTP is 1.1 and is documented in RFC 2616 [12].

## C.2   Basic Concepts

HTTP is a text-based, request-response protocol running over TCP/IP[1] and employing an exceptionally simple message exchange pattern. All messages are initiated from the client (called a user agent) to the Web server. The Web server acts upon this request and issues a single response. Common examples of user agents include desktop Web browsers, VoiceXML interpreters, and MRCP media resources (which must fetch objects such as audio files, grammar files, voiceprints, etc.). Web servers can vary from the very basic, which serve static files from the local disk over HTTP to the more advanced, which include the ability to execute server-side scripts or programs to dynamically generate HTTP responses. Often this latter capability is separated out into a separate entity called an application server. The application server[2] provides a managed container for server-side programs (e.g. Java servlets). A common deployment architecture uses a dedicated, optimised Web server for static content and an application server for dynamic content (the Web server is able to forward requests to the application server as required enabling a single point of entry to the server-side infrastructure).

There are a total of eight methods in HTTP but here we only concentrate on the two most commonly applied: GET and POST. Both methods are used to retrieve a resource with the latter enabling data to be efficiently pushed to the server as part of the request.

### C.2.1   `GET` method

The HTTP request message consists of a request line, several header fields, and a message body in the case of POST. The request line and the header fields are terminated with a carriage return line feed (CRLF). The header fields (or headers for short) employ the format of header-name: header-value CRLF. An additional CRLF delimits the header portion of the message from its optional body. An example of a GET request is illustrated below:

```
GET /index.html HTTP/1.1
Host: www.example.com
User-Agent: Simple-Browser/1.0
Accept: text/html
```

The request line starts with the method name, the Request-URI and the HTTP version number. Since the Request-URI is a relative URI, it is combined with the Host header field value to identify the exact resource (in this case http://www.example.com/index.html). If the Request-URI is an absolute URI, the Host header field is ignored. The User-Agent header field identifies the name of the agent acting on behalf of the user. The Accept header field indicates the MIME types accepted by the user agent - in this case the user agent indicates that it accepts HTML document formats.

The response to the request follows a similar format consisting of status line, followed by header fields and an optional message body:

---

[1]The default port for HTTP is 80.

[2]We often use the term Web server loosely to mean also the application server.

```
HTTP/1.1 200 OK
Server: Basic-Server/1.0
Date: Fri, 04 Aug 2006 10:46:24 GMT
Last-Modified: Mon, 25 May 2005 04:32:10 GMT
Set-Cookie2: jsessionid=12345;Version="1.0";Path="/"
Content-Type: text/html
Content-Length: 69

<html>
    <body>
        <p>Hello World!</p>
    </body>
</html>
```

The status line consists of the protocol version, the status code (200), and a status reason (OK). The first digit of the status code indicates the family: 1xx messages are informational, 2xx messages indicate success, 3xx message indicate redirection, 4xx messages indicate client failure, and 5xx messages indicate server failure. Table C.1 summarises common status codes and their meanings. The `Server` header field is analogous to the `User-Agent` header field in that it identifies the name of the Web server. The `Date` header field indicates the time at which the response message was originated, and the `Last-Modified` header field indicates the time at which the Web server believes the content was last modified. The `Set-Cookie2` header field is discussed in Section C.4. The requested resource is contained in the message body and is identified by the `Content-Type` header field. The length of the message body is indicated by the `Content-Length` (in bytes).

Web applications often need to provide a block of data such as the result of a form to the Web server. It is possible to supply a limited amount of data with the `GET` method by employing a query string. The query string is separated from the rest of the URI by a '?' character. By convention, data is encoded using a format of name=value and multiple name-value pairs are separated by ampersands, e.g.:

```
http://example.com/servlet?name1=value1&name2=my%20value2
```

Within the query string, certain characters are reserved and must be escaped if they are to be part of the name or value. In the above example, `param2` evaluates to `my value2` and the space is escaped as `\%20`. Characters are escaped using a character triplet consisting of a % followed by two hexadecimal values which encode the ASCII character. The URI syntax is described in RFC 2396 [22].

## C.2.2   `POST` method

While the `GET` method can be used to submit data to a Web server, it is not ideal for this purpose. For one, the user agent, Web server, or intermediate proxy cache may limit the total length of the URI. For another, it is not appropriate to place sensitive information in the URI

Table C.1   Common HTTP response status codes.

| Status code | Reason phrase | Meaning |
|---|---|---|
| 100 | Trying | The initial part of the request has been received and the client should continue to send the remainder of the request. |
| 200 | OK | Action received, understood, and accepted (the requested resource is in the response). |
| 302 | Found | The requested resource resides temporarily under a different URI. The URI for the resource is placed in the `Location` header field in the response. |
| 304 | Not Modified | The resource has not modified since the last request (implying a cached version can be used). |
| 401 | Not Authorized | The request requires authentication. |
| 404 | Not Found | The identified resource was not found. |
| 500 | Internal Server Error | The server encountered an unexpected error condition which prevented it from fulfilling the request. |

as it may appear in the user agent or Web server logs. Finally, encoding of binary data is inefficient (requiring three bytes for each byte of data if we attempt to use the %xx escaping method described above).

The POST method is designed specifically for sending data to the Web server. The data in question is supplied in the message body of the request. The `Content-Type` header field identifies the type of request data and the `Content-Length` header field identifies the length of the data. Placing the data in the message body allows arbitrary large content to be supplied, more efficient encoding of the data to be used (see below), and allows the data to be conveniently encrypted (e.g. if the HTTP protocol runs over a secure transport layer then the data is encrypted for free).

The two most common `Content-Type` encodings used in conjunction with the POST method are the `application/x-www-form-urlencoded` and `multipart/form-data` types. The `application/x-www-form-urlencoded` encoding is used for submitting textual information such as that obtained from a form. The encoding format is the same as described for the query string in the previous section. An example of a POST request using such an encoding is given below:

```
POST /submitform.cgi HTTP/1.1
Host: www.example.com
User-Agent: Simple-Browser/1.0
Accept: text/html
```

```
Content-Type: application/x-www-form-urlencoded
Content-Length: 30

name1=value1&name2=my%20value2
```

Usually the target of a POST request will be a script or server-side program designed to accept the data and operate on it (e.g. decode it and insert into a database). The response to a POST request is the same as for a GET request, e.g.:

```
HTTP/1.1 200 OK
Server: Basic-Server/1.0
Content-Type: text/html
Content-Length: 87

<html>
    <body>
        <p>Thank you for submitting data!</p>
    </body>
</html>
```

The multipart/form-data encoding is ideal for sending binary data to a Web server and is specified in RFC 2388 [78]. This encoding is commonly used in the WWW for uploading a file from a form and is used in VoiceXML to submit recorded media to a Web server, for example. The multipart encoding allows the message body to comprise of several parts delimited by an arbitrary boundary token (specified by the boundary attribute in the Content-Type value). The example below shows a binary file called test.bin being POSTed to the Web server. The four bytes of the binary file (shown here as the hexadecimal values for each byte) are encoded with a Content-Type of application/octet-stream (a more accurate Content-Type value is allowed here, for example, an audio file might be identified as audio/x-wav). For more information on multipart encoding, see Section 7.1.4.1.

```
POST /submitform.cgi HTTP/1.1
Host: www.example.com
User-Agent: Simple-Browser/1.0
Accept: text/html
Content-Type: multipart/form-data;boundary=1a2b3c
Content-Length: 141

--1a2b3c
Content-Disposition: form-data;name="myform";filename="test.bin"
Content-Type: application/octet-stream
```

```
0a 2b e2 f3
--1a2b3c--
```

## C.3   Caching

HTTP provides a set of rules to allow responses to be cached. Caching can lead to significant increases in performance, reduced load on Web servers (and corresponding application servers and databases), and allows network administrators to conserve bandwidth requirements and therefore save on IP connectivity costs. User agents usually implement a local cache either in memory, on disk, or both. Intermediate caches deployed in the network (commonly called proxy caches) provide network level caching. If one user agent has previously requested a document and a new user agent requests the same document, the response may be obtained from the proxy cache rather than contacting the Web server directly.

The specific purpose of HTTP caching is twofold:

1. to avoid making requests to the Web server when possible in most cases, and

2. to avoid the need to send full responses to requests in many cases.

The two key concepts behind the caching rules of HTTP are *expiration* and *validation*. A cached copy of a document that is not expired may be executed without requiring a costly fetch to the server. An expired document that is subsequently re-validated against the server may not require a re-transmit of the document to the platform. Specifying the expiration times is the responsibility of the application developer. HTTP caching is controlled primarily by setting HTTP header parameters in the Web server for a particular resource. On the Web server, the HTTP headers of `Cache-Control` or `Expires` can be used to specify how long the resource is to be considered fresh. For example, if the following is inserted in the HTTP response from a Web server:

```
Cache-Control: max-age=86400
```

then the resource is considered fresh for 86400 seconds or 24 hours. An alternative approach is to set the `Expires` header for a time 24 hours in the future, for example:

```
Expires: Fri, 09 Aug 2006 10:01:31 GMT
```

When a resource is still fresh, the user agent context may serve that resource directly from its cache. When a resource in the cache is stale and needs to revalidated with the server, the user agent will send a new HTTP request to the Web server. Included in the request may be a HTTP header called `If-Modified-Since` which includes the `Last-Modified` date that was originally returned in the HTTP response for that resource. The `If-Modified-Since` allows the Web server to determine if the resource has been updated since the last request. If it has, the successful HTTP response will specify a status code of `200 OK` and return the new resource in its body. If the resource has not changed, the HTTP response will specify a `302 Not Modified` status code and omit the message body, thus saving on bandwidth.

Newer implementations may use the header combinations of `ETag` / `If-None-Match` instead of the `Last-Modified` / `If-Modified-Since` pair. The `ETag` is an entity tag that provides a unique identifier for the content (e.g. a hash) and is returned in the original HTTP response containing the resource. The `If-None-Match` header specifies the `ETag` of the content. This allows the Web server to determine if the user agent has the latest content for the requested URI (new content will have a different `ETag`) and hence respond appropriately.

Thus far we have discussed caching header fields contained in the HTTP response only. It is also possible for the client to insert caching header fields in the request to indicate to upstream intermediate caches conditions on the response the user agent wishes to impose. The `Cache-Control: max-age` header field may be placed in the HTTP request to enable the user agent to request a response whose age is no greater than the specified number of seconds. For example, pressing the "Refresh" button on a standard desktop Web browser will trigger a HTTP request that specifies:

```
Cache-Control: max-age=0
```

and prevents upstream intermediate caches returning old (cached) versions of the content. The `Cache-Control: max-stale` header field may be used for essentially the opposite effect. This header field can be specified by the user agent in requests to indicate to upstream intermediate caches that it is willing to accept content that has exceeded its expiration time up to a given number of seconds, e.g.:

```
Cache-Control: max-stale=180
```

## C.4   Cookies

The HTTP protocol itself is inherently stateless . HTTP servers respond to each client request without relating that request to previous requests. Cookies are used within HTTP to enable stateful sessions. There are two mechanisms for cookie handling - the older and most widely implemented version is documented in RFC 2109 [51] and defines the `Set-Cookie` and `Cookie` HTTP header fields. The later version is documented in RFC 2965 [52] and defines the `Set-Cookie2` HTTP header field. The later specification contains several minor improvements on the original and uses a different header field name for backward compatibility.

At its simplest, a cookie is an attribute-value pair set by the HTTP server, usually with some associated reserved attributes that provide metadata, e.g. to specify the domain for which a cookie is valid. The HTTP server sets a cookie for the client to store by using the `Set-Cookie` (or `Set-Cookie2`) response header and the HTTP client returns the cookie in subsequent requests using the `Cookie` header. The example in Section C.2.1 illustrates a cookie with name `sessionid` and value `12345`. The `Version` attribute identifies the cookie specification version and the `Path` attribute specifies the subset of URIs to which this cookie applies. Subsequent requests will echo the cookie back to the server via the `Cookie` header field, thus allowing the HTTP server to correlate a request with a previous request and thereby define a stateful session, e.g.:

```
GET /other.html HTTP/1.1
Host: www.example.com
User-Agent: Simple-Browser/1.0
Accept: text/html
Cookie: $Version="1.0";jsessionid=12345;Path="/"
```

Application servers typically use cookies to set a unique session ID. The session ID is usually used in the application server to provide a key to access temporary server-side storage that persists for the length of the session.

The lifetime of a cookie can be set by adding a `Max-Age` attribute to the `Set-Cookie` or `Set-Cookie2` header field value. A user agent may choose to store the cookie for a shorter period of time. For example, an MRCP media resource starts a new empty cookie store with each new session.

### C.4.1   Security

HTTP messages can be sent securely by issuing them over a Secure Socket Layer (SSL) or Transport Layer Security (TLS) transport. An `https://` URI scheme indicates that a resource is available via secure HTTP[3]. Both SSL and TLS provide authentication and privacy by employing cryptographic techniques running beneath HTTP and above the TCP layer.

---

[3]The default port for secure HTTP is 443.