Web Application Development Lab

Task7: Analyse various HTTP requests and identify problems if any.

Aim: To analyse HTTP requests and identify problems of HTTP in web environment.

Description: HTTP is a transfer protocol used by the World Wide Web distributed hypermedia system to retrieve distributed objects. HTTP uses TCP as a transport layer.

Requests: The most commonly used method is "GET", which ask the server to send a copy of the object to the client. The client can also send a series of optional headers; these headers are in Request For Comment- 822 (RFC) format.

Example

GET /index.html HTTP/1.0

Accept: text/plain

Accept: text/html

Accept: */*

User-Agent: Yet another User Agent

Responses: Responses start with a status line indicating which version of HTTP the server is running, together with a result code and an optional message. The server now sends any requested data. After the data have been sent, the server drops the connection.

Example

HTTP/1.0 200 OK

Server: MDMA/0.1

MIME-version: 1.0

Content-type: text/html

Last-Modified: Thu Jul 7 00:25:33 1994

Content-Length: 2003

Access Patterns: A client requests a hypertext page, then issues a sequence of requests to retrieve any icons referenced in the first document. Once the client has retrieved the icons, the user will typically select a hypertext link to follow.

HTTP Illustration: The problems with HTTP can best be understood by looking at the network traffic generated by a typical HTTP transaction. The headers used in the request shown were captured from requesting object.

Stage 1: Time = 0

The trace begins with the client sending a connection request to the http port on the server

```
.00000 unc.32840 > ncsa.80: S 2785173504:2785173504(0) win 8760 <mss 1460> (DF)
```

Stage 2: Time = 0.077

The server responds to the connect request with a connect response. The client acknowledges the connect response, and send the first 536 bytes of the request.

```
.07769 ncsa.80 > unc.32840: S 530752000:530752000(0) ack 2785173505 win 8192
.07784 unc.32840 > ncsa.80: . ack 1 win 9112 (DF)
.07989 unc.32840 > ncsa.80: P 1:537(536) ack 1 win 9112 (DF)
```

Stage 3: Time = 0.35

The server acknowledges the first part of the request. The client then sends the second part, and without waiting for a response, follows up with the third and final part of the request.

```
.35079 ncsa.80 > unc.32840: . ack 537 win 8192
.35092 unc.32840 > ncsa.80: . 537:1073(536) ack 1 win 9112 (DF)
.35104 unc.32840 > ncsa.80: P 1073:1147(74) ack 1 win 9112 (DF)
```

Stage 4: Time = 0.45

The server sends a packet acknowledging the second and third parts of the request, and containing the first 512 bytes of the response. It follows this with another packet containing the second 512 bytes. The client then sends a message acknowledging the first two response packets.

```
.45116 ncsa.80 > unc.32840: . 1:513(512) ack 1147 win 8190
.45473 ncsa.80 > unc.32840: . 513:1025(512) ack 1147 win 8190
.45492 unc.32840 > ncsa.80: . ack 1025 win 9112 (DF)
```

Stage 5: Time = 0.53

The server sends the third and fourth response packet. The fourth packet also contains a flag indicating that the connection is being closed. The client acknowledges the data, and then sends a message announcing that it too is closing the connection. From the point of view of the client program, the transaction is now over.

```
.52521 ncsa.80 > unc.32840:  .1025:1537(512) ack 1147 win 8190
```

```
.52746 ncsa.80 > unc.32840: FP 1537:1853(316) ack 1147 win 8190
.52755 unc.32840 > ncsa.80: .ack 1854 win 9112 (DF)
.52876 unc.32840 > ncsa.80: F 1147:1147(0) ack 1854 win 9112 (DF)
```

Stage 6: Time = 0.60

The server acknowledges the close.

```
.59904 ncsa.80 > unc.32840:  .ack 1148 win 8189
```

Connection Establishment: TCP establishes connections via a three-way handshake. The client sends a connection request, the server responds, and the client acknowledges the response. The client can send data along with the acknowledgement. Since the client must wait for the server to send its connection response.

Data transfer: Segments: When TCP transfers a stream data; it breaks up the stream into small segments. The size of each segment can vary up to a maximum segment size (MSS). For remote connections, the MSS defaults to 536 bytes.

Data transfer: Windows and Slow Start: Instead of having to wait for each packet to be acknowledged, TCP allows the sender to send out new segments even though it may not have received acknowledgements for previous segments. To prevent the sender from overflowing the receiver's buffers, in each segment the receiver tells the sender how much data it is prepared to accept without acknowledgments. This value is known as the window size. This approach is ideal for normal connections; these connections tend to last a relatively long time, and the effects of slow start are negligible. However, for short lived connections like those used in HTTP, the effect of slow start is devastating.

Affects HTTP: HTTP is struck out by slow start on both the client and server sides. Because the HTTP headers are longer than the MSS, the client TCP needs to use two segments (Stage2). Because the congestion window is initialized to one, we need to wait for the first segment to be acknowledged before we can send the second and third (Stage3). This adds an extra Round Trip Time (RTT) onto the minimum time for the transaction.

When the server is ready to send the response, it starts with a congestion window set to 2. This is because the acknowledgement it sent in Stage 3 counts is counted as a successful transmission, allowing the congestion window to open up a notch before it comes time to send data.

Latency and Bandwidth: Latency and Bandwidth are the two keys to protocol performance. Latency, as measured by the RTT, is a measure of the fixed costs of a transaction and does not change as documents get bigger or smaller. Bandwidth, as we discussed earlier, is a measure of how long it takes to send data.

Effects of opening new connection per transaction: The cost of opening a new connection for each transaction can be clearly seen by estimating the transaction time as it would have been if we had been reusing an existing connection. In the above example, the total transaction time was 530 ms.

Effects of Requesting: A Single Object per Transaction: Because HTTP has no way to ask for multiple objects with a single request, each fetch requires a single transaction. With the current protocol, this would require a new connection to be set up; however, even if the connection were to be kept open, each request/response pair would incur a separate round trip delay.

Other Problems: One scalability problem caused by the single request per connection paradigm occurs due to TCP's TIME_WAIT state. When a server closes a TCP, connection, it is required to keep information about that connection for a period of time, in case a delayed packet turns up and sabotages a new incarnation of the connection.

The recommended time to keep this information is 240 seconds. Because of this, a server will leave some resources allocated for every single connection closed in the past four minutes. On a busy server, this can add up to thousands of control blocks.

Conclusion: HTTP/1.0 interacts badly with TCP. It incurs frequent round-trip delays due to connection establishment, performs slow start in both directions for short duration connections, and incurs heavy latency penalties due to the mismatch of the typical access profiles with the single request per transaction model. HTTP/1.0 also requires busy servers to dedicate resources to maintaining TIME_WAIT information for large numbers of closed connections.

Result: The various HTTP requests and problems were analyzed.