第二次作业

a

1

In HTTP 1.0, the default connection model uses non-persistent connections. For fetching any object, the following sequence occurs:

- A new TCP connection is opened, requiring 1 RTT.
- Request-response for an object happens, requiring 1 more RTT.
- TCP connection gets closed.

So, in this case, the following will happen:

- The HTML is fetched, requiring 2 RTTs.
- Then, each of the 5 referenced objects is fetched independently, each requiring 2 RTTs.

In total, 6 objects are fetched in succession. So the total RTTs required:

[2 \times 6 = 12 \text{ RTTs}]

2

The concepts used are the same as in 1).

The following sequence occurs:

- 5 TCP connections are established in parallel, requiring 1 RTT.
- The HTML is fetched on the first connection, requiring 1 more RTT. This connection then gets closed.

So, total RTTs required:

$$ 2 + 2 = 4 \text{ RTTs} $$

3

The concepts used are the same as in 1).

The following sequence occurs:

- 3 TCP connections are established in parallel, requiring 1 RTT.
- The HTML is fetched on the first connection, requiring 1 more RTT. This connection then gets closed, leaving us with 2 open connections.
- 2 referenced objects are fetched on the 2 open connections in parallel, requiring 1 more RTT.
- For the remaining 3 referenced objects, new connections need to be opened, requiring 2 RTTs each.

So, total RTTs required:

$$ 2 + 2 + 2 = 6 \text{ RTTs} $$

4

In HTTP 1.1, the default connection model uses persistent connections.

Here, the sequencing is:

- A TCP connection is opened, requiring 1 RTT.
- The HTML is fetched, requiring 1 RTT.
- The requests for the 6 referenced objects are sent together (pipelining), and responses are received for all, assuming zero failures, in 1 RTT.
- The TCP connection is closed.

So, total RTTs required:

[3 \text{ RTTs}]

5

Using the same basic concepts as in 4), the sequencing is:

- A TCP connection is established, requiring 1 RTT.
- The HTML is fetched, requiring 1 RTT.
- Each of the 5 referenced objects is fetched one after another, requiring 1 RTT each.
- The TCP connection is closed.

So, total RTTs required:

[1 + 1 + 5 = 7 \text{ RTTs}]

6

Using the same basic concepts as in 4) and 5), the sequencing is:

- 5 TCP connections are opened, requiring 1 RTT.
- The HTML is fetched, requiring 1 more RTT. At this time, 5 open connections remain.
- 5 of the referenced objects are fetched on the 5 open connections in parallel, requiring 1 RTT. One open connection remains.
- The remaining referenced object is fetched on the remaining connection, requiring 1 RTT.
- All connections are closed.

So, total RTTs required:

[2 + 1 + 1 = 4 \text{ RTTs}]

7

Similar to 6), total RTTs required:

[2 + 1 + \lceil \frac{5-1}{3} \rceil = 5 \text{ RTTs}]

b

Discuss why pipelined protocols outperform stop-and-wait protocols in throughput

1. Pipelined protocols surpass stop-and-wait protocols by effectively utilizing network bandwidth, whereas the latter inefficiently squanders network resources.
2. When multiple packets need to be sent, a pipelined protocol significantly reduces the time required for transmitting the last packet. Conversely, in a stop-and-wait protocol, the time until the last packet is sent becomes considerably longer due to waiting for acknowledgments.
3. Within a pipelined protocol, the sender dispatches multiple frames to the receiver, allowing for re-transmission of damaged or suspicious frames. Conversely, the stop-and-wait protocol only sends one frame at a time and waits for acknowledgment from the receiver.
4. The efficiency formula for pipelined protocols is given as (\frac{N}{1+2a}), while for the stop-and-wait protocol, it is (\frac{1}{1+2a}).

c

1. **Head-of-Line Blocking**: The most significant drawback of HTTP/1.1 pipelining is head-of-line blocking. If one request stalls or takes longer to process, subsequent requests in the pipeline cannot be completed until the earlier request finishes, even if they are ready to be sent.

2. **Complexity in Error Handling**: When a pipelined request fails, determining the state of subsequent requests becomes complicated. Should they be retried? Were they processed? The uncertainty complicates error recovery.
3. **Out-of-Order Responses**: While servers are expected to respond in the order requests are received, network issues or server-side processing might cause out-of-order responses, complicating client-side handling.
4. **Non-Idempotent Requests**: Pipelining non-idempotent requests (like HTTP POST) can be risky. If a connection drops after some requests are processed but before responses are received, a client cannot safely retry without risking unintended side effects.
5. **Server and Proxy Support**: Not all servers and intermediaries support pipelining, leading to compatibility issues. Some might process requests sequentially, negating the benefits, or worse, mishandle pipelined requests.