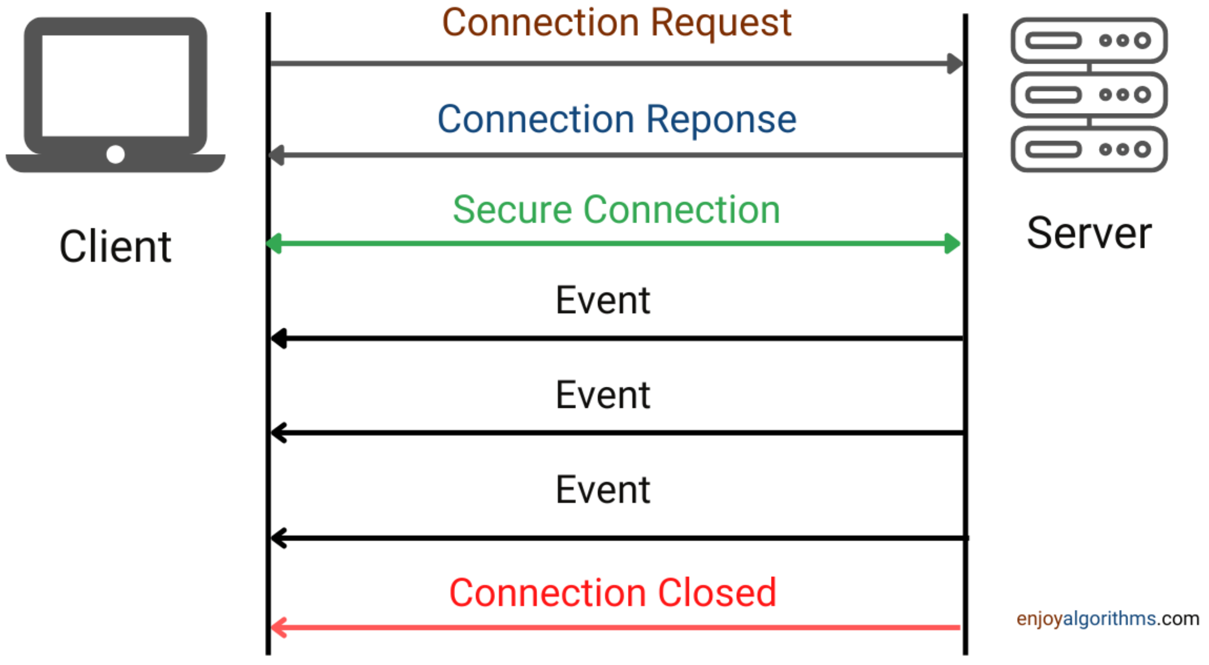
Server Sent Events: System Design Concept

What is Server-Sent Events?

There are two main approaches for delivering data from servers to clients: client pull and server push. In a client pull scenario, the client (such as a web browser) requests data from the server. In a server push scenario, the server continuously sends updates to the client without an initial request.

Server-Sent Events (SSE) is a type of server push technology that establishes a long-lasting connection between the client and the server. It allows the server to automatically send updates to the client via an HTTP connection, using a single directional channel for data delivery. To receive these updates, the client uses a JavaScript API called EventSource. SSE is designed to improve cross-browser streaming capabilities by establishing a unidirectional connection to send continuous data streams and updates to the client.



Before diving into the specifics of how Server-Sent Events work, it is important to understand the basics of how communication occurs over the internet using HTTP requests. HTTP is a common protocol used for data transfer in client-server architectures.

In the context of Server-Sent Events, if the client loses the connection to the event source, it will try to reconnect by sending the ID of the last event received to the server via a new HTTP request, using the "Last-Event-ID" header. The server will then listen for this request and start sending events that have occurred since the supplied ID. This allows the client to maintain a continuous stream of updates from the server, even if the connection is temporarily lost.

Let’s look into the internal architecture of Server-Sent Events.

Server Sent Events are real-time events that are emitted by the server and received by the browser. To establish communication between the client and server, the client creates a new JavaScript EventSource object and passes the URL of the endpoint that is expected to return a stream of events. The EventSource interface connects to the server over HTTP and receives events in an event-stream or text format.

The client sends a regular HTTP request to the server and expects to receive a series of event messages in return. Whenever the server writes an event to the HTTP response, the client receives it and processes it in a listener callback function. The HTTP response connection remains open until it is considered stale or until the client closes it.

Server-Sent Events are formatted according to a specific standard. Each event consists of key/value pairs separated by a colon, with each pair terminated by a newline. Two newlines mark the end of the events.

Let’s look at how Client and Server are implemented in SSE.

Client-Side Implementation

As previously mentioned, the client creates a new EventSource object to receive events from the server. The EventSource object takes a URL as an argument, which is the location from where the events will be drawn in "text/event-stream" format.

The client receives the events and processes them in a listener callback function. Callback functions, also known as event handlers, are registered to handle specific events. The EventSource object has a method called addEventListener, which is used to register these handlers. If the original event message contains multiple data lines, they will be concatenated together by the browser to form a single string before the callback functions are called.

It is worth noting that there is a limit to the number of Server-Sent Events connections that can be active at any given time. Most browsers are limited to six SSE connections.

Server-Side Implementation

The server receives an HTTP request from the client and responds with valid Server-Sent Event messages. It instructs the client about the content type and tells the client to keep the connection alive in order to facilitate the transmission of events over the same connection.

The server can only accept EventSource requests and must maintain a list of all connected users in order to emit new stream events. It also needs to keep a history of messages in case a client misses any, and should be able to remove dropped connections from the list of connected users.

It's worth noting that the server-side implementation of Server-Sent Events can be coded in any programming language, such as Java, C, Python, or Go, while the client-side requires the use of JavaScript.

Stopping an Event Stream

Once the required data has been received, it is necessary to close the connection and stop the event stream. There are two ways to do this, depending on the client and server:

Client-side: The client has the option to stop the events using the .close() method of the EventSource object. When this method is called, the server detects it and stops sending events to the client by closing the corresponding HTTP response.

Server-side: The server can also stop the event stream by sending a final event with a unique ID that corresponds to the "end of stream" event. Alternatively, the server can stop the event stream by closing the HTTP response connection with the client.

In this way, both the client and server have the ability to close the connection and stop the event stream.

Connection failure

In the real world, it is not possible to maintain a fully persistent connection. In Server-Sent Events, the connection is established via HTTP and may be dropped due to network issues, which can disrupt the transfer of events and lead to incomplete event messages. To address this issue, the client can try to reconnect to the event source by sending the ID of the last event as an HTTP header called "Last-Event-ID" to the server via a new HTTP request. The server listens for this request and begins sending events that have occurred since the supplied ID. This allows the client to catch up on any missed events and helps to ensure the reliability of the event stream.

Application of Server Sent Events

Server-Sent Events are widely used in the development of real-time web applications and are also used to build real-time notification services, which are common in many applications for notifying users or administrators. Companies like Uber have also relied on Server-Sent Events for updating trip information for both drivers and customers.

For example, consider a scenario in which a rider requests a ride and a driver is available to provide the service. Uber's matching technology finds a match on the backend and sends a trip offer to the driver. All parties involved (rider, driver, and backend) should now be aware of each other's intentions. To keep updated on the progress of the trip, the driver app can poll the server every few seconds to check for new offers, while the rider app can also poll the server every few seconds to check if a driver has been assigned. This continuous communication is facilitated by Server-Sent Events.

Conclusion

Server-Sent Events are useful for delivering fast updates and have a low overhead in their implementation. They are well-suited for systems that require real-time, one-way data flow and are commonly used in the news feeds of social media platforms like Twitter, Instagram, and Facebook. They are also useful for updating stock price charts and providing live sports updates.

Thanks to Suyash for his contribution in creating the first version of this content. If you have any queries/doubts/feedback, please write us at contact@enjoyalgorithms.com. Enjoy learning, Enjoy system design, Enjoy algorithms!