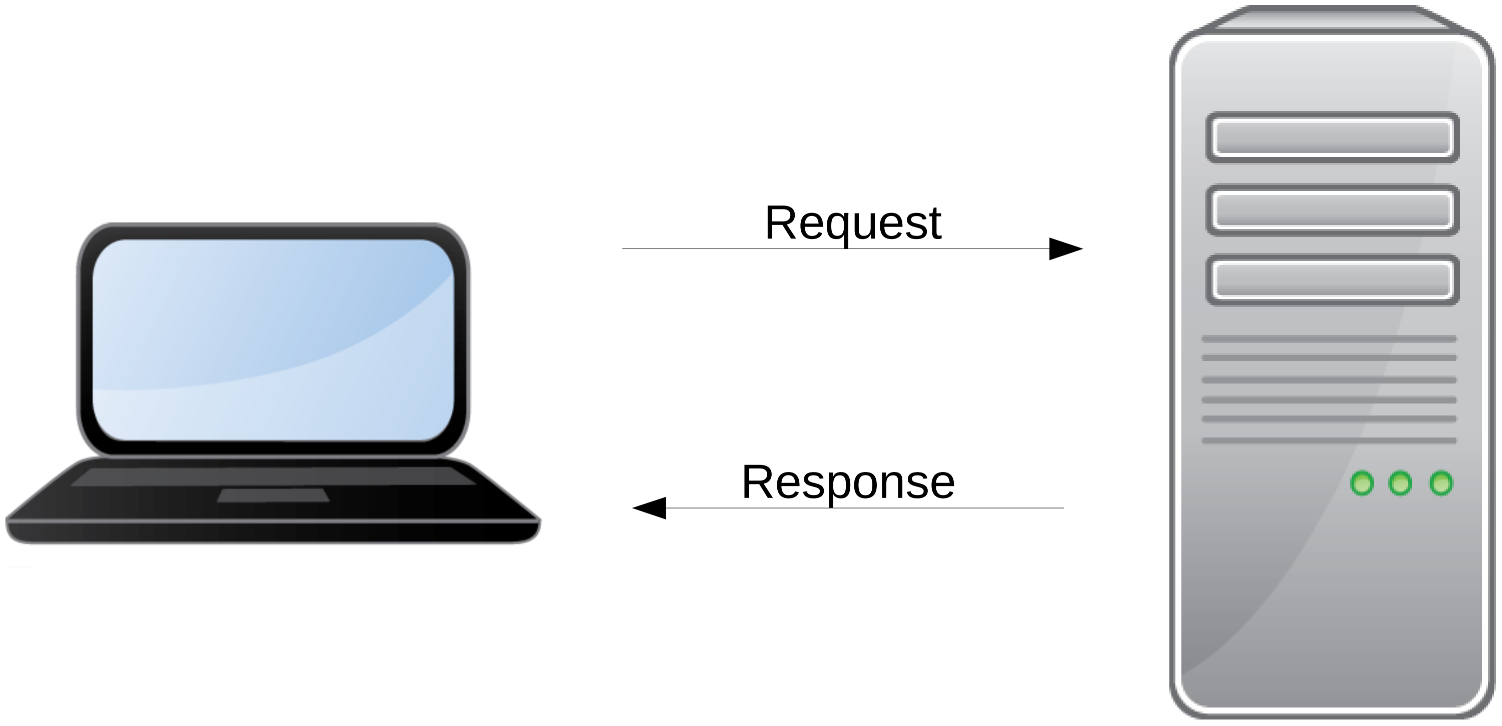
Web Technologies

Lab session 11

Real time web applications

- HTTP protocol follows a client-server model
  - The client **always initiates** the request
  - The server responds



Real time web applications

- What if the server wants to initiate conversation? Use cases:
  - Loading data in the browser as it is created on the server
  - Sending chat messages to clients as they arrive to the server
- In traditional applications we have to refresh the page to see changes

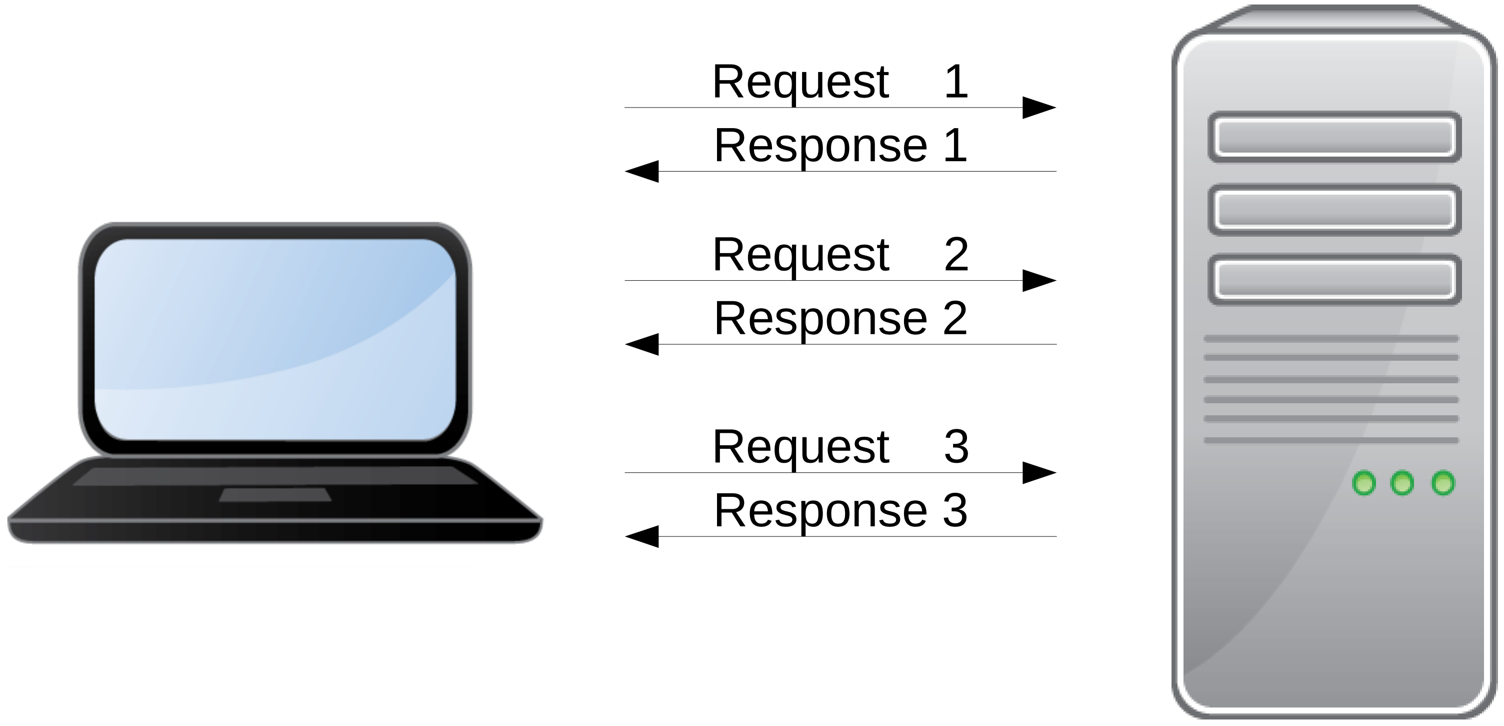
Real time web applications

- Several approaches possible
  - AJAX polling
  - Long polling
  - HTML5 Server-Sent Events
  - Web sockets
- Each has pros and cons

AJAX polling

- Idea
  - Have a JavaScript code that constantly polls the web server for new data
  - *Example chat app*
- Cons
  - Overhead: when there is no data, we are wasting bandwidth

AJAX polling

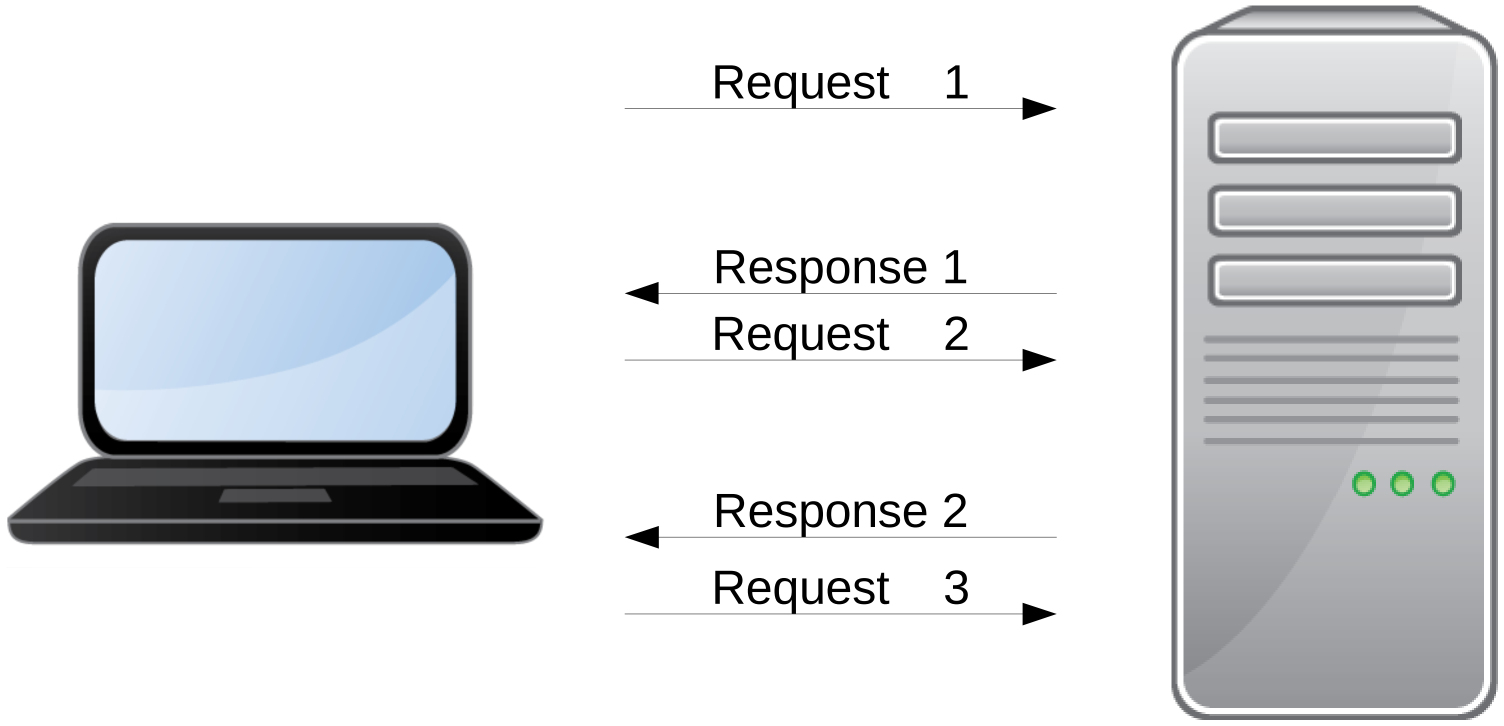


Long polling

- Idea
  - The client sends the initial request, but then ...
  - the **server does not respond immediately: it waits** until new data is available and then returns the response
  - Then the client sends another request and waits ...
- Also called **hanging GET**

Long polling

- Compared to AJAX polling, no empty responses (when there is no new data)
- Cons: still have to send an HTTP request for every new data

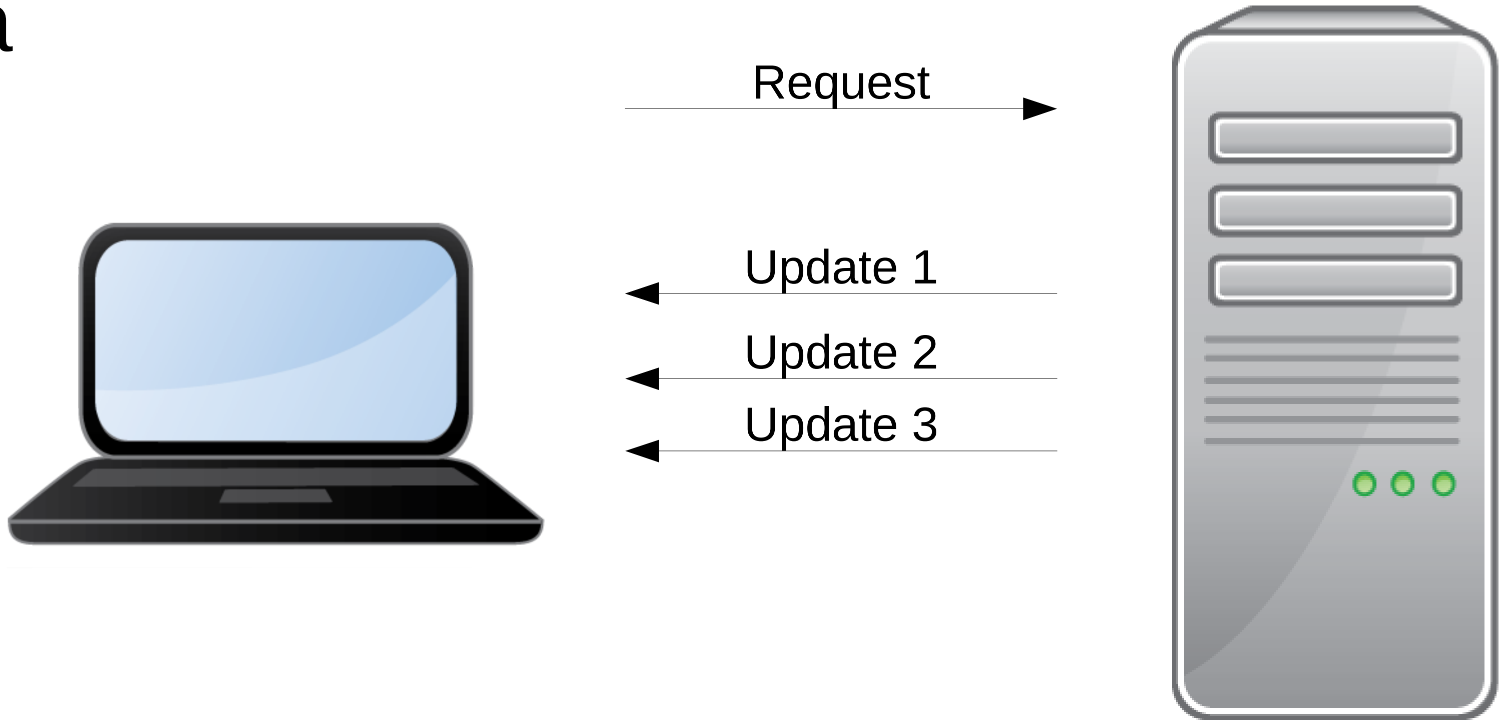


Server-Sent Events (SSE)

- Idea
  - Client sends a request, the server sends back the response header, but the response body is never complete
  - When new data is created, the server simply writes it to the response body, and the client immediately receives it

Server-Sent Events (SSE)

- **Pro.** A single request is sent to the server: server can send fresh updates without client explicitly requesting it
- **Con.** Unidirectional: only the server can send data

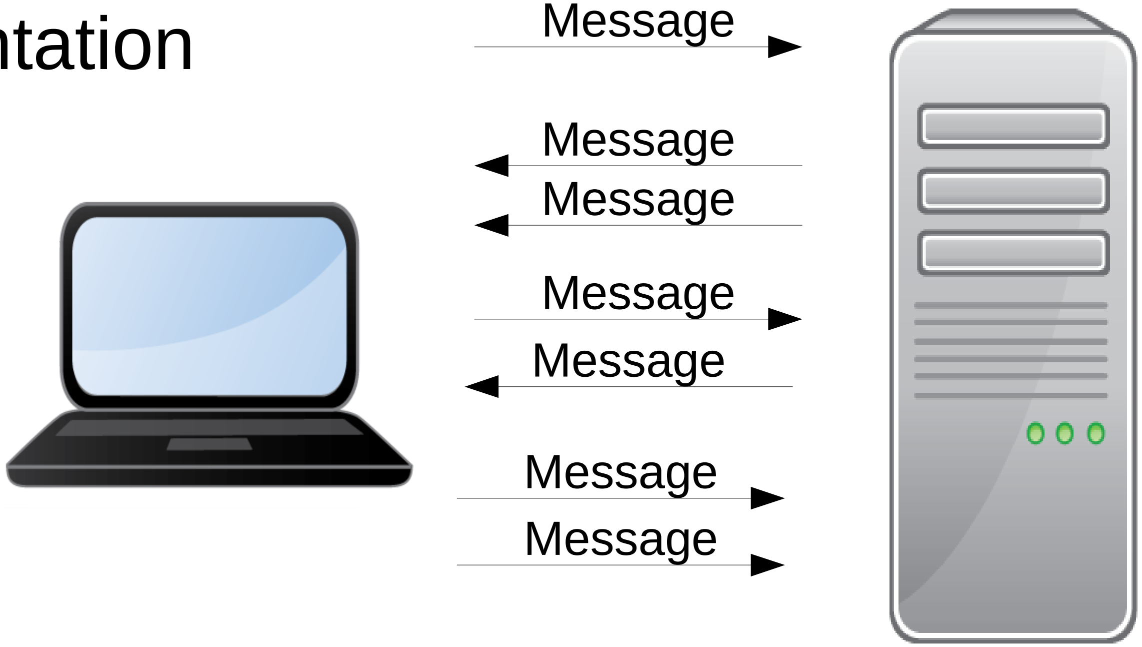


Web socket

- Idea
  - Have a **separate** and an **independent** protocol for bi-directional communication between the client and the server
  - Initial handshake starts with an HTTP request, but then the communication switches to a bidirectional binary protocol not related to HTTP

Web socket

- **Pro.** Richer (bidirectional) and more efficient than SSE
- **Cons**
  - More complex than SSE
  - Requires additional protocol and server implementation



Server-Sent Events: Client

- JavaScript API
- We subscribe to events using **EventSource** object
- If the client gets disconnected, it automatically reconnects

```
const source = new EventSource("/path/to/stream-url");
source.onopen = function () { ... };
source.onerror = function () { ... };
source.addEventListener("event_name", function (event) {
  processFoo(event.data);
});
source.onmessage = function (event) {
  log_message(event.id, event.data);
  if (event.id == "CLOSE") {
    source.close();
  }
}
```

Server-Sent Events: Server

- Set the response headers
  - `Content-Type: text/event-stream`
  - `Cache-Control: no-cache`
- Write the data as a continuous text stream that ends with the empty line
  - `data: Data to be sent\n\n`
- Data can be multiline
  - `data: First line\n`
  - `data: Second line\n\n`

Server-Sent Events: Server

- Each data (event) should have a unique id

  -
    ```
    id: 123\n
    data: Data to be sent\n
    \n
    ```
- This is how client keeps track of what it has seen
- If the connection is dropped, the client sends back a new request where a header name `Last-Event-ID` is set to the largest `id` the client received

Server-Sent Events: Server

- Events can have names

  -
    ```
    id: 1\n
    event: new user\n
    data: Sven\n
    \n
    ```

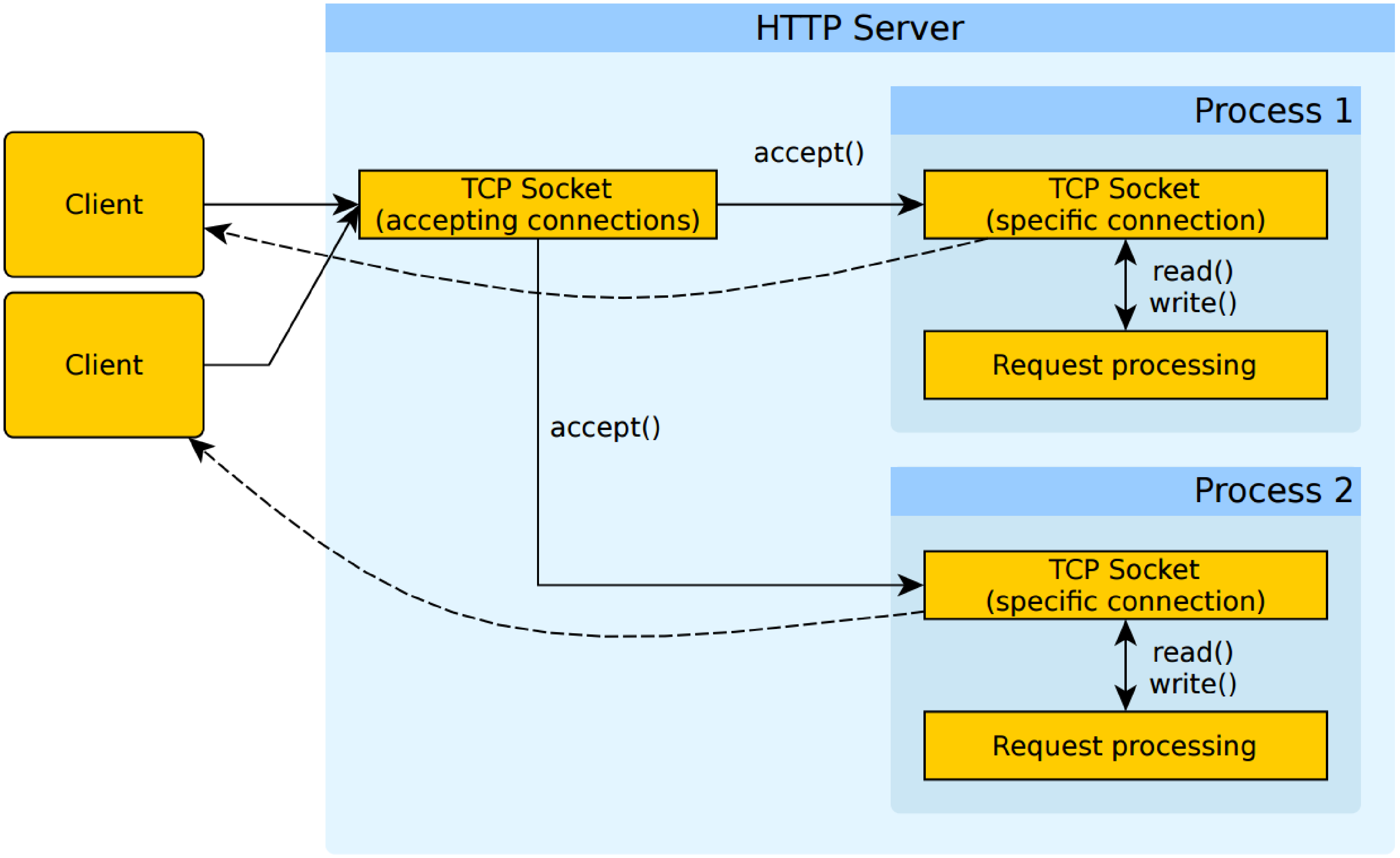
  -
    ```
    id: 2\n
    event: message\n
    data: {user: ‘Sven’, message: ‘Hi!’}\n
    \n
    ```
- Processing incoming data on clients gets easier

SSE: Caveats

- To fully use SSE, our web application should not close the event stream: the web page showing events must never end – it has to run indefinitely
  - Using a `while (true)` concept
- This has serious performance issues with the LAMP stack
  - Recall Apache’s pre-fork model
  - Each request is handled by a dedicated proces
  - **We eventually run out of processes!**
- To fully leverage SSE, our entire stack has to be event based

Parallel requests: pre-fork

- A pool of processes or threads



Resources

- Examples
  - https://www.w3schools.com/html/html5_serversentevents.asp
  - https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events/Using_server-sent_events
  - https://www.html5rocks.com/en/tutorials/eventsource/basics/

- HTTP specs
  - https://html.spec.whatwg.org/multipage/comms.html#server-sent-events
  - https://www.w3.org/TR/2009/WD-eventsource-20091029/