

# Web 2.0

## Lecture 8: Protocols for the Realtime Web

**doc. Ing. Tomáš Vitvar, Ph.D.**

tomas@vitvar.com • @TomasVitvar • <http://vitvar.com>



Czech Technical University in Prague

Faculty of Information Technologies • Software and Web Engineering • <http://vitvar.com/courses/w20>



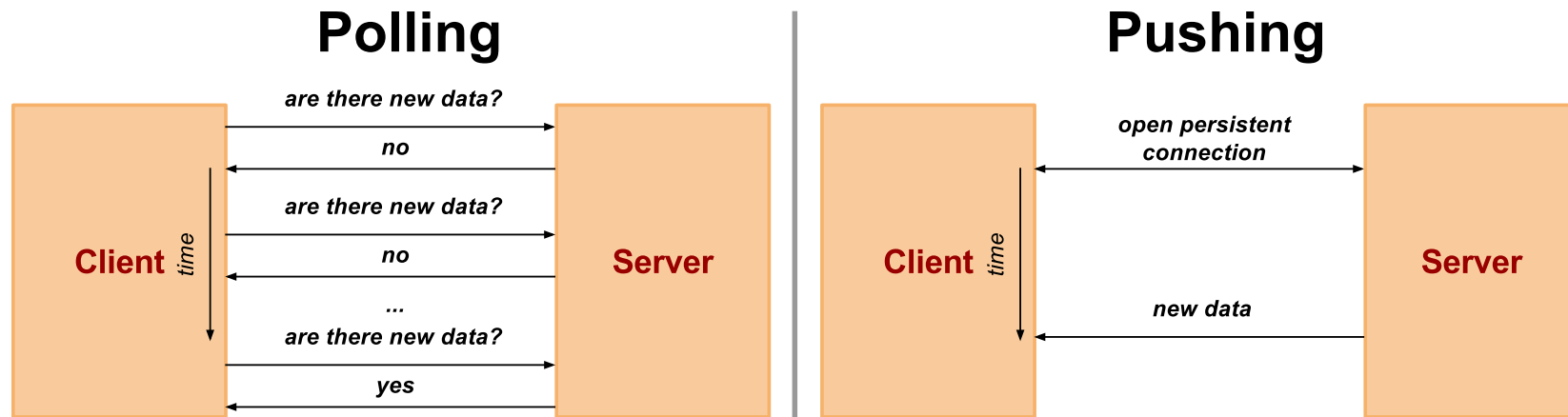
Evropský sociální fond  
Praha & EU: Investujeme do vaší budoucnosti

Modified: Thu Mar 16 2017, 00:37:56  
Humla v0.3

# Overview

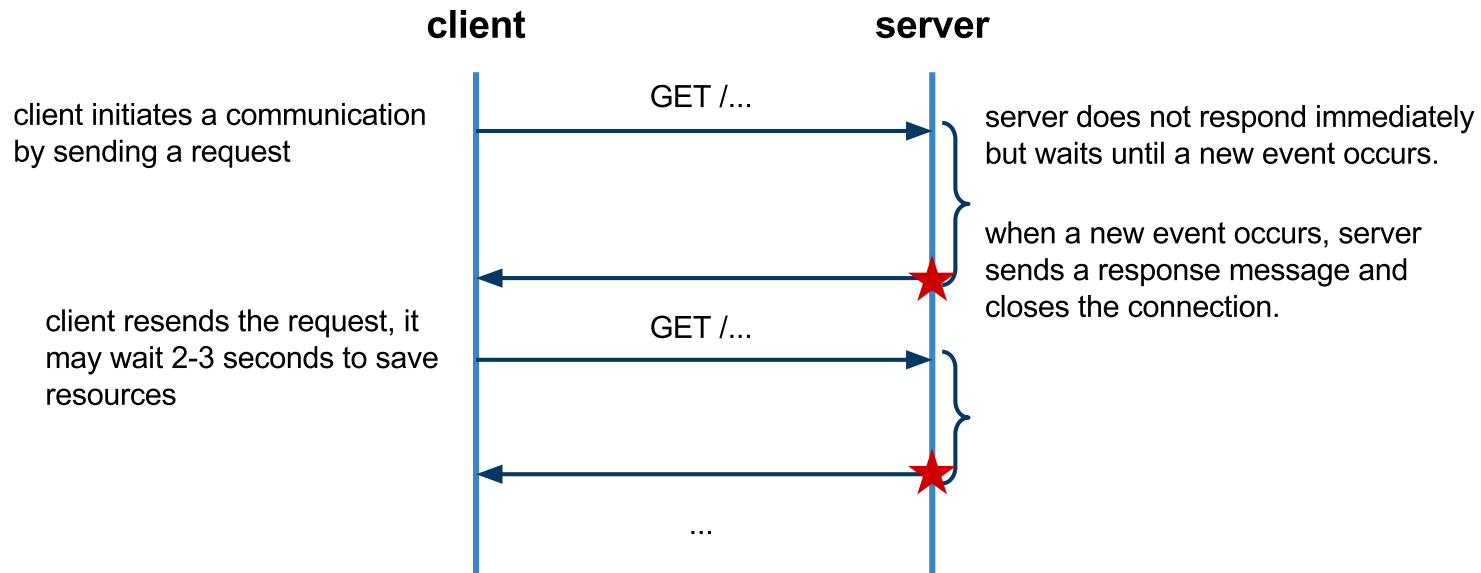
- Long-polling and Streaming
- WebSocket Protocol
- New I/O Model

# Pushing and Polling



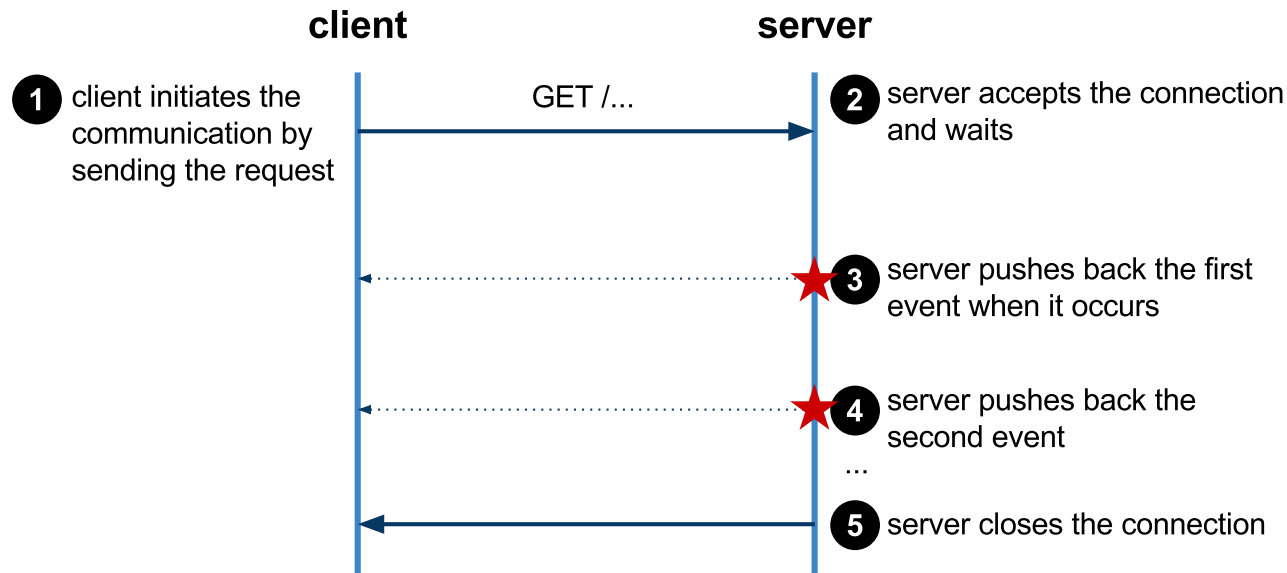
- Conceptual basis in messaging architectures
  - *event-driven architectures (EDA)*
- **HTTP is a request-response protocol**
  - *response cannot be sent without request*
  - *server cannot initiate the communication*
- **Polling** – client periodically checks for updates on the server
- **Pushing** – updates from the server (also called COMET)
  - = **long polling** – server holds the request for some time
  - = **streaming** – server sends updates without closing the socket

# HTTP Long Polling



- Server holds long-poll requests
  - *server responds when an event or a timeout occurs*
  - *saves computing resources at the server as well as network resources*
  - *can be applied over HTTP persistent and non-persistent communication*
- Issues:
  - *maximum time of the request processing at the server*
  - *concurrent requests processing at the server*

# HTTP Streaming



- server defers the response until an event or timeout is available
- when an event is available, server sends it back to client as part of the response; this does not terminate the connection
- server is able to send pieces of response w/o terminating the conn.
  - using **transfer-encoding** header in HTTP 1.1
  - using *End of File* in HTTP 1.0
  - (server omits **content-length** in the response)

# Chunked Response

- Transfer encoding **chunked**
  - *It allows to send multiple sets of data over a single connection*
  - *a chunk represents data for the event*

```
1 HTTP/1.1 200 OK
2 Content-Type: text/plain
3 Transfer-Encoding: chunked
4
5 25
6 This is the data in the first chunk
7
8 1C
9 and this is the second one
10
11 0
```

- *Each chunk starts with hexadecimal value for length*
  - *End of response is marked with the chunk length of 0*
- Steps:
  - *server sends HTTP headers and the first chunk (step 3)*
  - *server sends second and subsequent chunk of data (step 4)*
  - *server terminates the connection (step 5)*

# Issues with Chunked Response

- Chunks vs. Events
  - *chunks cannot be considered as app messages (events)*
  - *intermediaries might "re-chunk" the message stream*
    - *e.g., combining different chunks into a longer one*
- Client Buffering
  - *clients may buffer all data chunks before they make the response available to the client application*
- HTTP streaming in browsers
  - *Server-sent events*

# Server-Sent Events

- W3C specification
  - *part of HTML5 specs, see Server-Sent Events* [🔗](#)
  - *API to handle HTTP streaming in browsers by using DOM events*
  - *transparent to underlying HTTP streaming mechanism*
    - *can use both chunked messages and EOF*
  - *same origin policy applies*
- **EventSource** interface
  - *event handlers: **onopen**, **onmessage**, **onerror***
  - *constructor **EventSource(url)** – creates and opens the stream*
  - *method **close()** – closes the connection*
  - *attribute **readyState***
    - **CONNECTING** – *The connection has not yet been established, or it was closed and the user agent is reconnecting.*
    - **OPEN** – *The user agent has an open connection and is dispatching events as it receives them.*
    - **CLOSED** – *The conn. is not open, the user agent is not reconnecting.*



# Example

- Initiating **EventSource**

```
1  if (window.EventSource != null) {  
2      var source = new EventSource('your_event_stream.php');  
3  } else {  
4      // Result to xhr polling :(  
5  }
```

- Defining event handlers

```
1  source.addEventListener('message', function(e) {  
2      // fires when new event occurs, e.data contains the event data  
3  }, false);  
4  
5  source.addEventListener('open', function(e) {  
6      // Connection was opened  
7  }, false);  
8  
9  source.addEventListener('error', function(e) {  
10     if (e.readyState == EventSource.CLOSED) {  
11         // Connection was closed  
12     }  
13 }, false);
```

- *when the conn. is closed, the browser reconnects every ~3 seconds*  
→ *can be changed using **retry** attribute in the message data*

# Event Stream Format

- Format

- *response's **content-type** must be **text/event-stream***
- *every line starts with **data:**, event message terminates with 2 **\n** chars.*
- *every message may have associated **id** (is optional)*

```
1 | id: 12345\n
2 | data: first line\n
3 | data: second line\n\n
```

- JSON data in multiple lines of the message

```
1 | data: {\n
2 | data: "msg": "hello world",\n
3 | data: "id": 12345\n
4 | data: }\n\n
```

- Changing the reconnection time

- *default is 3 seconds*

```
1 | retry: 10000\n
2 | data: hello world\n\n
```

# Server-side implementation

- Java Servlet
  - *method* `doGet`

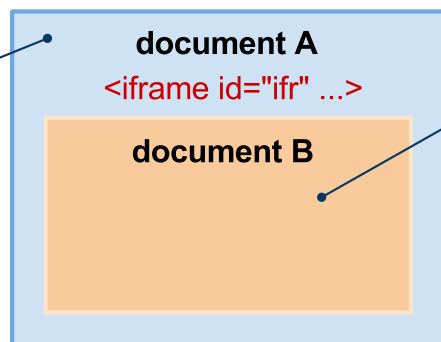
```
1  public void doGet(HttpServletRequest req, HttpServletResponse resp)
2      throws IOException {
3
4      // set http headers
5      resp.setContentType("text/event-stream");
6      resp.setHeader("cache-control", "no-cache");
7
8      // current time in milliseconds
9      long ms = System.currentTimeMillis();
10
11     // push data to the client for 20 seconds
12     // client should reconnect when the connection is closed
13     while (System.currentTimeMillis() - ms < 20000) {
14         resp.getWriter().print("data: servlet runs for " +
15                               (System.currentTimeMillis() - ms)/1000 + " seconds.\n\n");
16         resp.getWriter().flush();
17         try {
18             Thread.sleep(4000);
19         } catch (InterruptedException e) {
20             // do nothing;
21         }
22     }
23 }
```

# Other Technologies

- Cross-document messaging

**script in document A**

```
var o=document.getElementById("ifr");  
o.contentWindow.postMessage("Hello world",  
"http://example.org/")
```



**script in document B**

```
window.addEventListener('message', receiver, false);  
function receiver(e) {  
  if (e.origin == 'http://example.com') {  
    if (e.data == 'Hello world') {  
      e.source.postMessage('Hello', e.origin);  
    } else {  
      alert(e.data);  
    }  
  }  
}
```

- *The use of Cross Document Messaging for streaming*

1. *The client loads a streaming resource in a hidden **iframe***
2. *The server pushes a JavaScript code to the **iframe***
3. *The browser executes the code as it arrives from the server*
4. *The embedded iframe's code posts a message to the upper document*

- Channel API

- *Google Technology for streaming API for AppEngine*
- *not based on HTTP streaming*
- *utilizes XMPP capabilities + hidden iframe at client-side*

# Overview

- Long-polling and Streaming
- **WebSocket Protocol**
- New I/O Model

# WebSocket

- Specifications
  - *IETF defines WebSocket Protocol* [↗](#)
  - *W3C defines WebSocket API* [↗](#)
- Design principles
  - *a new protocol*
    - *browsers, web servers, and proxy servers need to support it*
  - *a layer on top of TCP*
  - *bi-directional communication between client and servers*
    - *low-latency apps without HTTP overhead*
  - *Web origin-based security model for browsers*
    - *same origin policy, cross-origin resource sharing*
  - *support multiple server-side endpoints*
- Two phases
  - *Handshake – as an **upgrade** of a HTTP connection*
  - *data transfer – the protocol-specific on-the-wire data transfer*

# Handshake – Request

- Request

- *client sends a following HTTP request to upgrade the connection to WebSocket*

```
1 GET /chat HTTP/1.1
2 Host: server.example.com
3 Upgrade: websocket
4 Connection: Upgrade
5 Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
6 Sec-WebSocket-Origin: http://example.com
7 Sec-WebSocket-Protocol: chat, superchat
8 Sec-WebSocket-Version: 7
```

- **Connection** – *request to upgrade the protocol*
- **Upgrade** – *protocol to upgrade to*
- **Sec-WebSocket-Key** – *a client key for later validation*
- **Sec-WebSocket-Origin** – *origin of the request*
- **Sec-WebSocket-Protocol** – *list of sub-protocols that client supports (proprietary)*

# Handshake – Response

- Response

- *server accepts the request and responds as follows*

```
1 | HTTP/1.1 101 Switching Protocols
2 | Upgrade: websocket
3 | Connection: Upgrade
4 | Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+x0o=
5 | Sec-WebSocket-Protocol: chat
```

- **101 Switching Protocols** – *status code for a successful upgrade*

- **Sec-WebSocket-Protocol** – *a sub-protocol that the server selected from the list of protocols in the request*

- **Sec-WebSocket-Accept** – *a key to prove it has received a client WebSocket handshake request*

- *Formula to compute **Sec-WebSocket-Accept***

```
1 | Sec-WebSocket-Accept = Base64Encode(SHA-1(Sec-WebSocket-Key +
2 | "258EAF5-E914-47DA-95CA-C5AB0DC85B11"))
```

- **SHA-1** – *hashing function*

- **Base64Encode** – *Base64 encoding function*

- **"258EAF5-E914-47DA-95CA-C5AB0DC85B11"** – *magic number*



# Data Transfer

- After successful handshake
  - *socket between the client and the "resource" at the server is established*
  - *client and the server can both read and write from/to the socket*
  - *No HTTP headers overhead*
- Data Framing
  - *defines a format for data transmitted in TCP packets*
  - *payload length, closing frame, ping, pong, type of data (text/binary), etc. and payload (message data)*

# WebSocket API

- Client-side API
  - *clients to utilize WebSocket, supported by Chrome, Safari*
  - *Hides complexity of WebSocket protocol for the developer*
- JavaScript example

```
1  // ws is a new URL schema for WebSocket protocol; 'chat' is a sub-protocol
2  var connection = new WebSocket('ws://server.example.org/chat', 'chat');
3
4  // When the connection is open, send some data to the server
5  connection.onopen = function () {
6      // connection.protocol contains sub-protocol selected by the server
7      console.log('subprotocol is: ' + connection.protocol);
8      connection.send('data');
9  };
10
11 // Log errors
12 connection.onerror = function (error) {
13     console.log('WebSocket Error ' + error);
14 };
15
16 // Log messages from the server
17 connection.onmessage = function (e) {
18     console.log('Server: ' + e.data);
19 };
20
21 ...
22
23 // closes the connection
24 connection.close()
```

# Sockets.IO

- Many options for streaming
  - *long-polling, streaming, iframe, WebSockets*
  - *Not all browsers support WebSockets*
  - *Socket.IO* [↗](#) – *a layer providing a unified API*
- Sockets.IO
  - *API and JavaScript implementation*
  - *checks the availability of WebSocket protocol*
    - *fallback to long-polling or other technologies when not available*

```
1 // creates a new socket
2 var socket = new io.Socket();
3
4 // event handlers
5 socket.on('connect', function(){
6     socket.send('hi!');
7 })
8 socket.on('message', function(data){
9     alert(data);
10 })
11 socket.on('disconnect', function(){}))
```

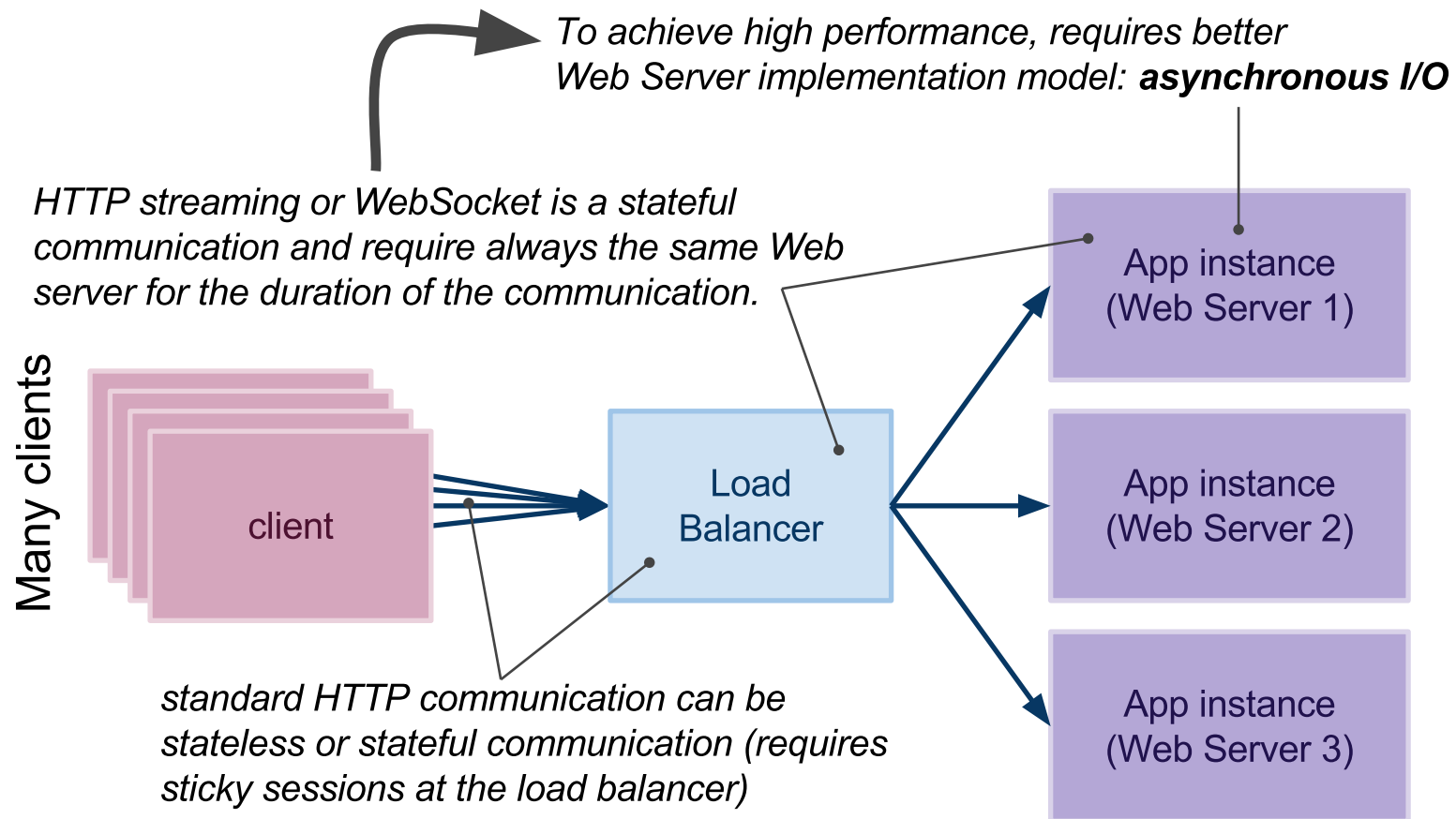
# Overview

- Long-polling and Streaming
- WebSocket Protocol
- New I/O Model

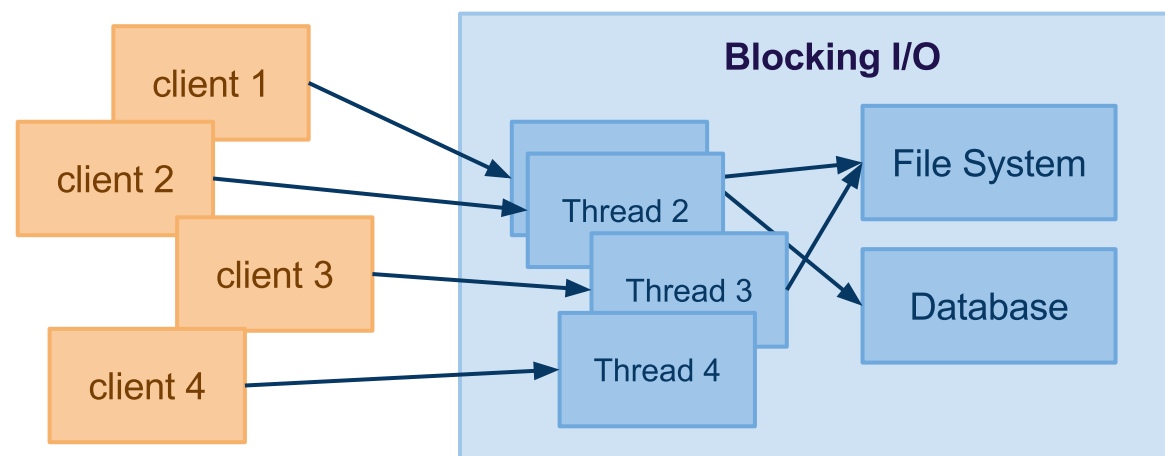
# Highly Scalable Web Servers

- Concurrent connections
  - *servers must serve a huge amount of concurrent connections*
  - *Highly scalable Web apps*
    - *many concurrent requests at the same time*
    - *QPS: 10-100 or more (GAE scales up to 500 QPS)*
  - *more significant with new trends regarding streaming (HTTP and WebSocket)*
- Web server implementation models:  
**Synchronous I/O vs. Asynchronous I/O**
  - *synchronous I/O (aka blocking I/O)*
    - *traditional: server creates a thread for every connection*
  - *asynchronous I/O (aka non-blocking I/O)*
    - *new one, server handles processing of requests separately from incoming connections*

# Web App Scalability

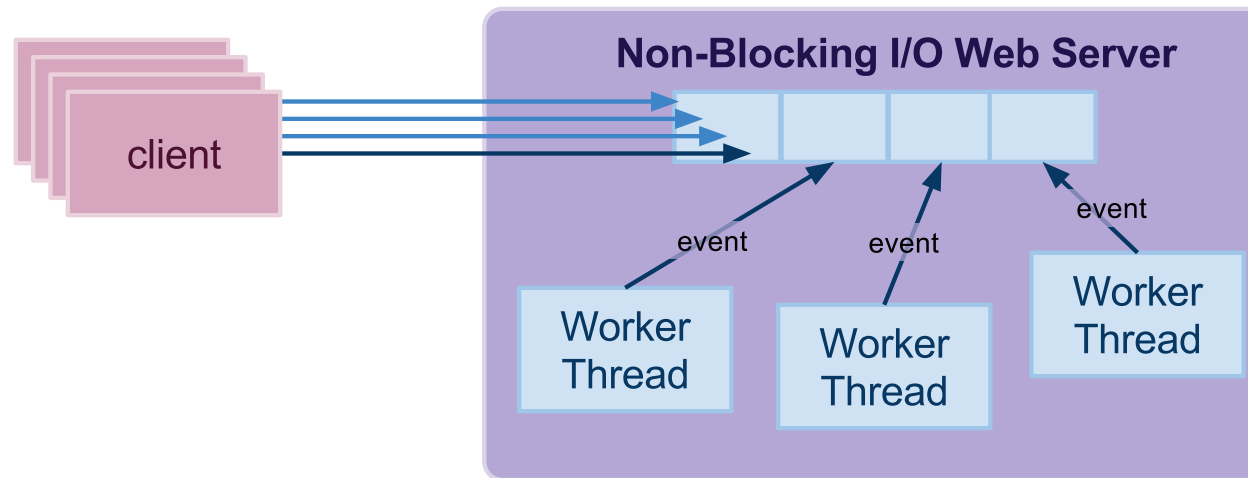


# Synchronous I/O Model



- every request served by a single thread
  - *reserved for the whole processing, the thread is "blocked"*
- when processing of the request is fast, scales well
  - *OS maintains a pool of threads that are reused for new requests*
- when processing of the request requires other interactions with DB/FS or network communication is slow → scaling is bad
  - *more significant with streaming (long polling or HTTP streaming)*
- OS may create couple of hundreds of threads (~1000 is very large)
  - *app may serve over 1K clients easily*

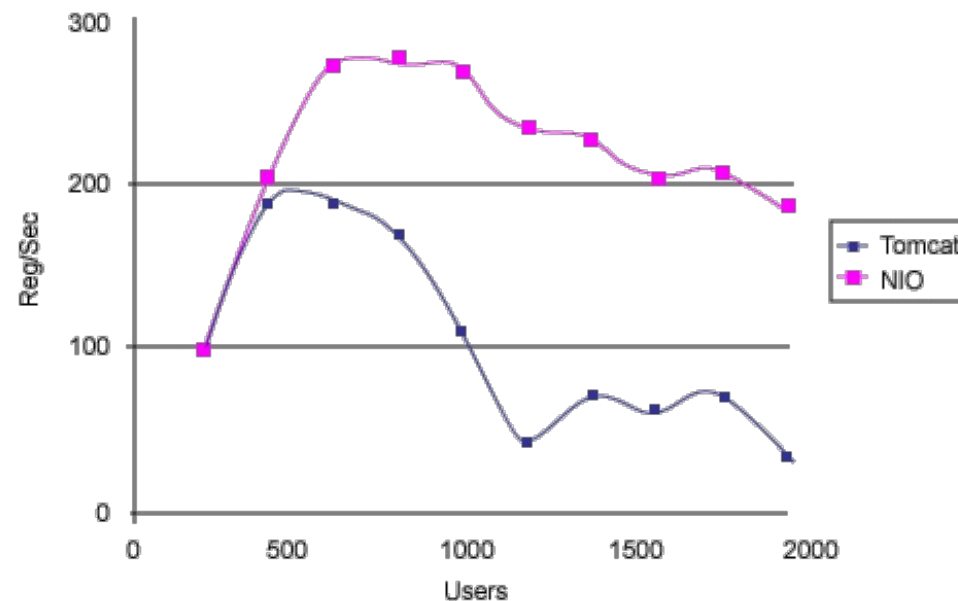
# Asynchronous I/O Model



- requests/connections maintained by the OS
- Web server reacts on the events
  - *such as new socket, read, write*
  - *it may create a working thread to perform required processing*
  - *Web server may control the number of Worker Threads*
- significantly less number of working threads as opposed to blocking I/O



# Performance Experiment



Non-blocking vs. blocking performance (number of requests per second served by the server vs. number of users), source [The Servlet API and NIO: Together at last](#)

- Tomcat – Java-based, uses I/O blocking communication
  - *configured to run up to 2,000 threads*
- NIO – a Web server implemented using Java.NIO (Java New I/O)
  - *only 4 working threads*
- simple HTTP **GET** serving textual content

# Emerging Technologies

- Node.js

- *NodeJS* [🔗](#) – *event-driven I/O framework on JavaScript V8 engine*  
→ *every I/O as event:*

```
1  // pseudo code; ask for the last edited time of a file
2  stat( 'somefile', function( result ) {
3      // use the result here
4  } );
5  ...
6
7  // web server
8  var http = require('http');
9  http.createServer(function (req, res) {
10     res.writeHead(200, {'Content-Type': 'text/plain'});
11     res.end('Hello World\n');
12 }).listen(8080, "127.0.0.1");
13 console.log('Server running at http://127.0.0.1:8080/');
```

- *runs in Linux/Unix/OS X environments*
- *Executes your server-side JavaScript code*
- *Socket.IO as a modul provides a streaming layer*

- Java.NIO

- *.Java New I/O. standard in .Java SE 7*