Software Construction and User Interface (SE/ComS 319)

Ali Jannesari

Department of Computer Science

Iowa State University, Fall 2018

EVENT-DRIVEN PROGRAMMING

Outline

- Event-Driven Programming (EDD):
  - Concepts
    - Event handling
    - Event-driven architecture
    - Asynchronous programming, etc.
  - Web UI and EDD with JavaScript (Node.js)
  - GUI and EDD with JavaFX

Event-Driven programming (1)

- A programming paradigm in which the flow of the program is determined by **events** such as:
  - User actions (mouse clicks, key presses)
  - Sensor outputs (mostly in embedded systems)
  - Messages from other programs/threads (device drivers)

Event-Driven programming (2)

- Event-driven programming
  - … is the dominant paradigm used in **graphical user interfaces** and other applications
    - e.g. JavaScript web applications: performing actions in response to user input.
  - … is used in **Human-computer interaction (HCI)**

Human-computer interaction (HCI)

- HCI: Interactive computing systems for human use
    - CLI: command line interface (with keyboard)
    - **GUI: graphical user interface (mouse)**
    - NUI: natural user interface with Audio/Video (Kinect)
- A main HCI Component: **Interaction**
    - User interaction
    - Event
    - Event Handling
    - Output
- A **GOOD GUI** allows users to perform interactive tasks easily:
    - What you see is what you get

Event-Driven programming (2)

- Application waits (idles) after initialization until the user generates an event trough an input device (keyboard, mouse, …).
- The OS dispatches the event to the application who owns the active window.
- The corresponding event handler(s) of the application is invoked to process the event.

Event-Driven programming (2)

Listener: Event Loop

Event Mapping & Even Dispatching

Event

Event Handler

```
char c;
boolean done = false;
while(!done) {
        system.out (“Please make your selection, q to end:”)
        system.in (c);
        switch(c) {
        case “+”:
                add( );
                break;
        case “-”:
                sub( );
                break;
        case “q”:
                done = true;
                break;
        }
}
```

Event-Driven programming (4)

1. Event generators: GUI components (e.g. buttons, menus, …)
2. Events/Messages: e.g. MouseClick, …
3. Event loop (Listener) : an infinite loop constantly waits for events.
4. Event mapping / Event registration: inform event dispatcher which event an event hander is for.
5. Event dispatcher: dispatch events to the corresponding event handlers.
6. Event handlers: methods for processing events. E.g. OnMouseClick(), …

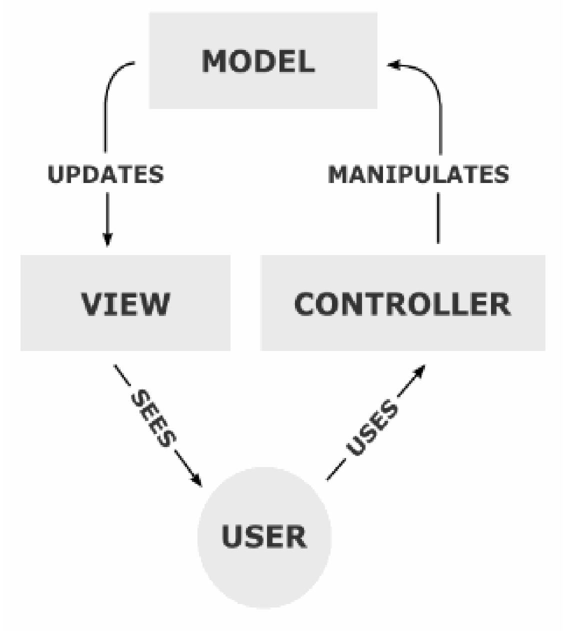
Event-driven programming (5)

- Concepts
- Event-driven programming with
  - JavaScript (Node.js)
  - JavaFX (Java)

EVENT-DRIVEN PROGRAMMING WEB USER INTERFACES

Event-Driven Programming – Web UI

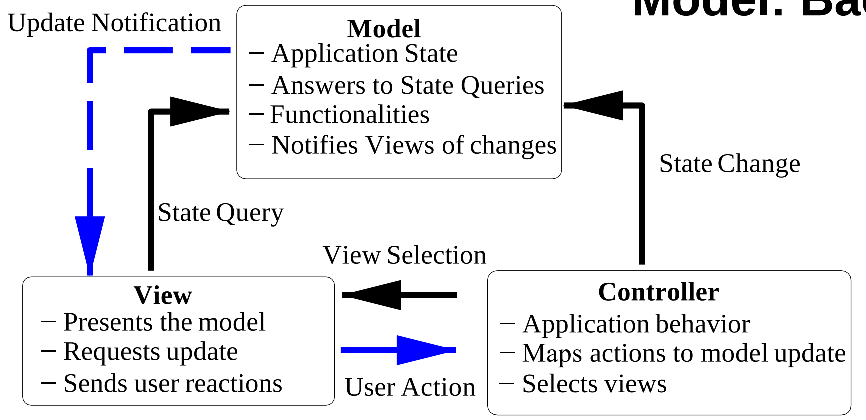
- **MVC (Model – View – Controller) in Web UI:**
  - **View**: Browser presentation (HTML)
  - **Model**: Data (Backend Database or (simple) embedded)
  - **Controller**:
    - Client scripts/programs, e.g. JavaScript
    - Server scripts/programs, e.g. Node.js



MVC

- Model-View-Controller architecture:

Model: Backend Database



View: HTML

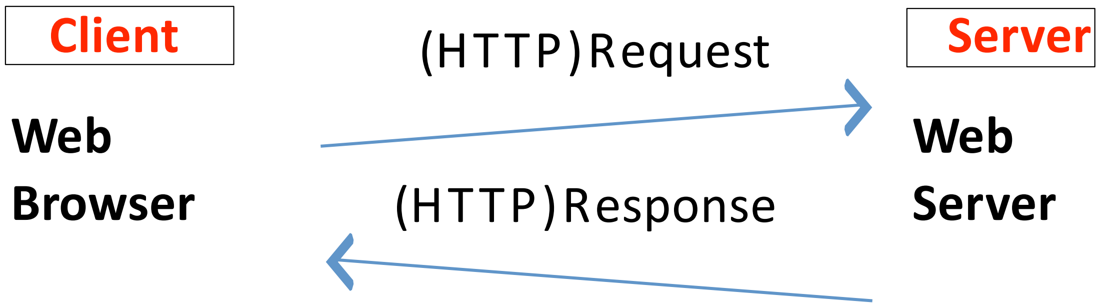
Controller: JavaScript, Node.js code

- Advantages:
  - Loosely coupled, modular
  - Model with different views
  - Controller decides when/how to update the model and/or the view
  - Model can change the view

Client/Server programming

- Use **client-side** programming for
  - Validating user input
  - Prompting users for confirmation, presenting quick information
  - Calculations on the client side
  - Preparing user-oriented presentation
  - Any function that does not require server-side information
- Use **server-side** programming for
  - Maintaining data across sessions, clients, applications

Web software: Client/Server (1)



HTML
CSS
JavaFX
Plugins
(Java)Script
…

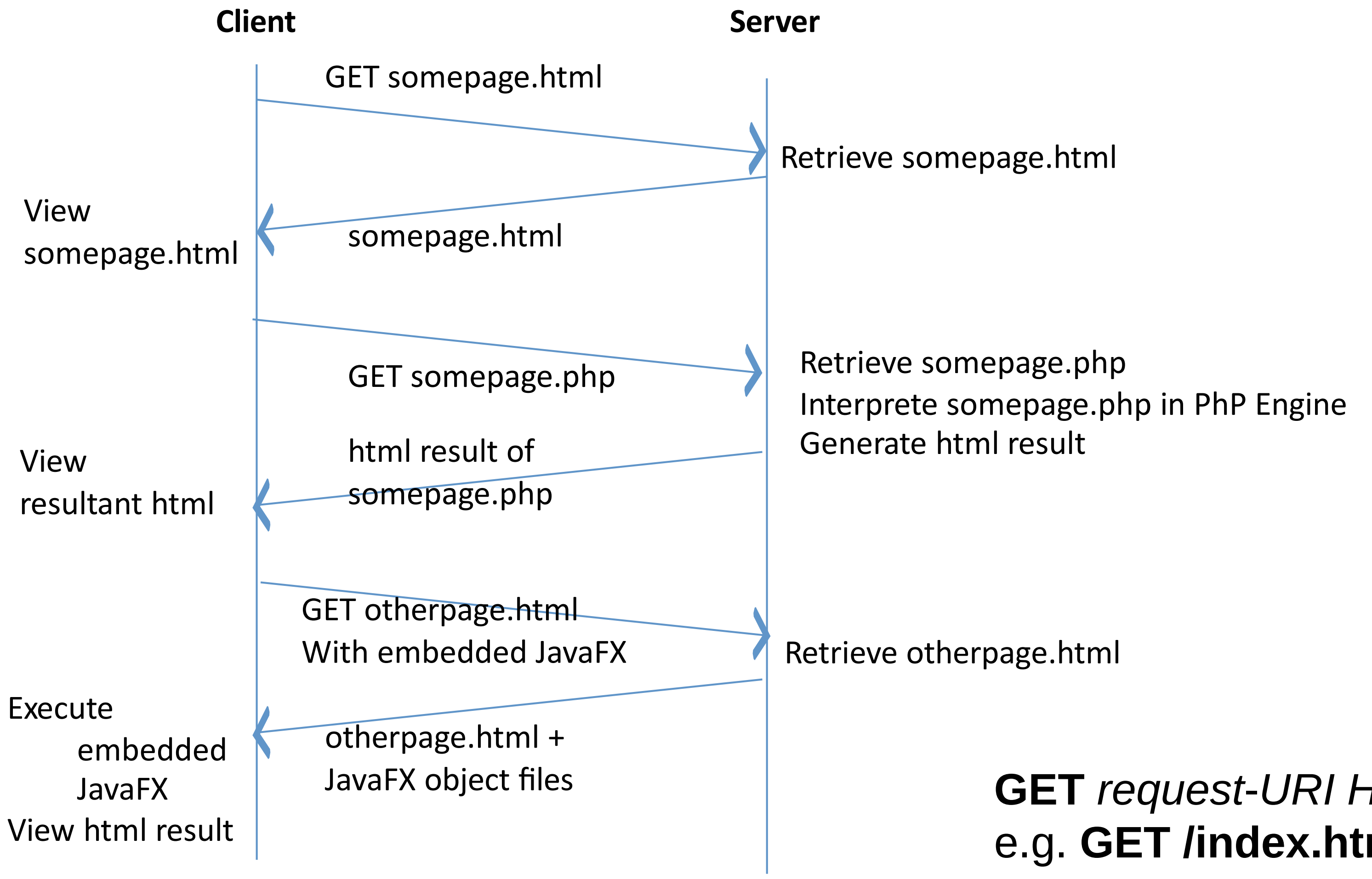
CGI
Node.js
ASP/JSP
PhP
…

HTTP (Hypertext Transfer Protocol): HTTP is a client-server application-level protocol. It typically runs over a TCP/IP connection.

Web software: Client/Server (2)

- Web-client and Web-server communicates using HTTP protocol
  - Client can send a HTTP request: method “`get`" or “`post`"
  - Server can read a HTTP request and produce HTTP response
- Server side programs should be capable of reading HTTP request and producing HTTP response

Web software: Client/Server (3)



GET *request-URI HTTP-version*
e.g. **GET /index.html HTTP/1.0**

Common Gateway Interface (CGI) – Classic method

- Standard for the server to communicate with external applications
- Server receives a client (Http) request to access a CGI program
- Server creates a new process to execute the program
- Server passes client request data to the program
- Program executes, terminates, produces data (HTML page)
- Server sends back (Http response) the HTML page with result to the client

HTML – Example

```
<html>
<head></head>
<body>
<form action="<some-server side cgi program>" method="post">
First Name: <input type="text" name="fname"/>
Last Name: <input type="text" name="lname"/>
<input type="submit" value="Submit"/>
</form>
</body>
</html>
```

- Once the user clicks the submit button, the data provided in the form fields are “submitted" to the server where it is processed by a CGI program!

HTTP Request/Response Message

- Message Header
  - Who is the requester/responder
  - Time of request/response
  - Protocol used …
- Message Body
  - Actual message being exchanged

HTTP Request

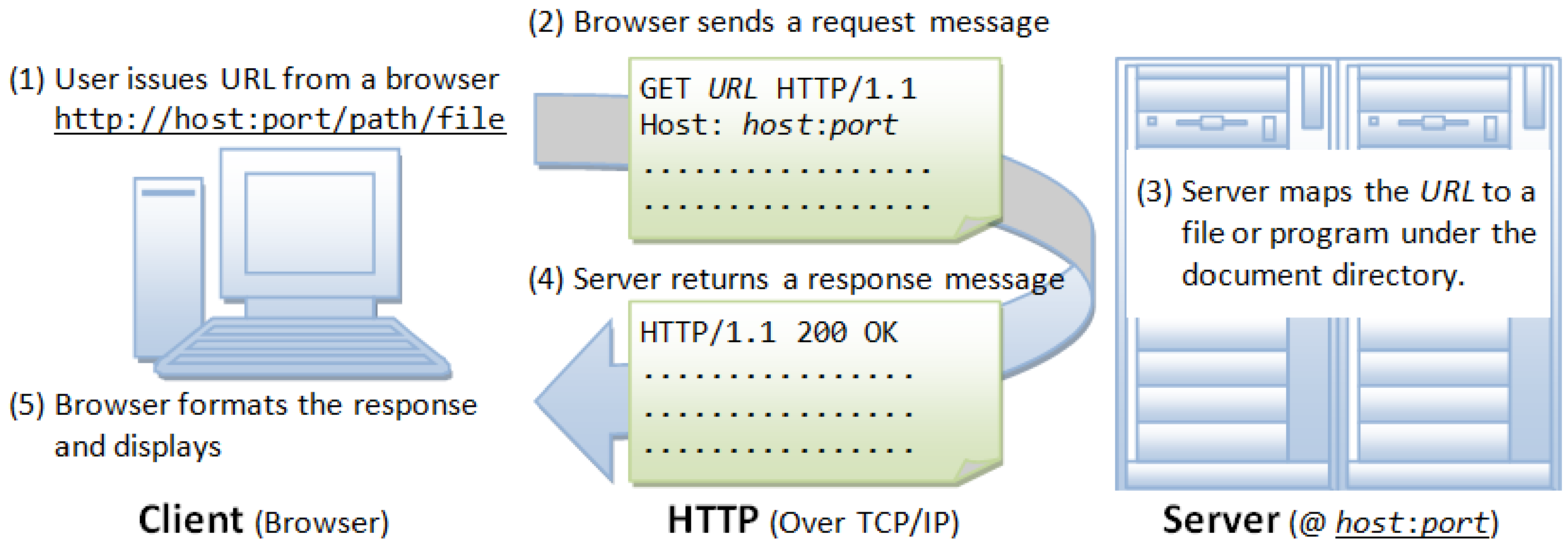
```
GET /index.html HTTP/1.1
Host: http://www.se.iastate.edu
Accept-Language: en
User-Agent: Mozilla/8.0
Query-String: ...
```

HTTP Response

```
HTTP/1.1 200 OK
Date: Sat, 27 Oct 2007 16:00:00 GMT
Server: Apache
Content-Type: text/html
```

- Response Codes:
  - 200s: good request/response
  - 300s: redirection as the requested resource is not available
  - 400s: bad request leading to failure to respond
  - 500s: server failure

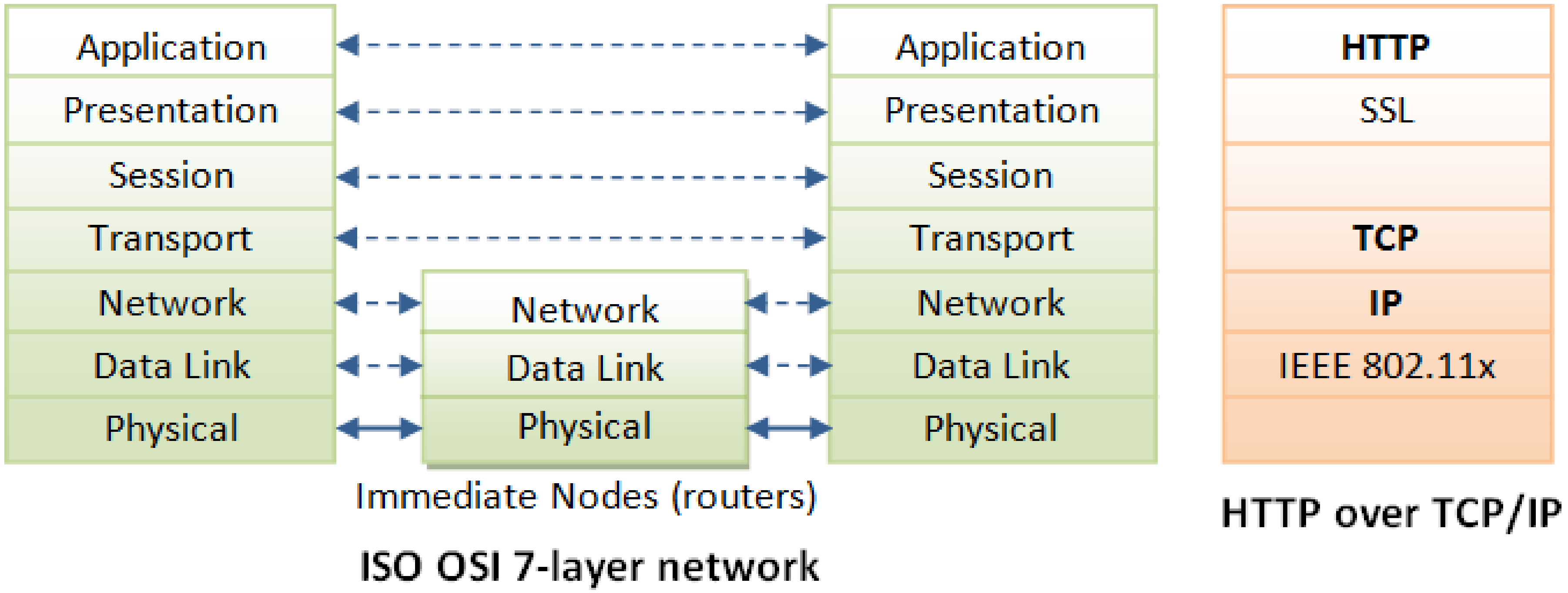
Web software: Client/Server


(1) User issues URL from a browser
http://host:port/path/file
(2) Browser sends a request message
GET URL HTTP/1.1
Host: host:port
(3) Server maps the URL to a file or program under the document directory.
(4) Server returns a response message
HTTP/1.1 200 OK
(5) Browser formats the response and displays
Client (Browser)
HTTP (Over TCP/IP)
Server (@ host:port)

Source: https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html

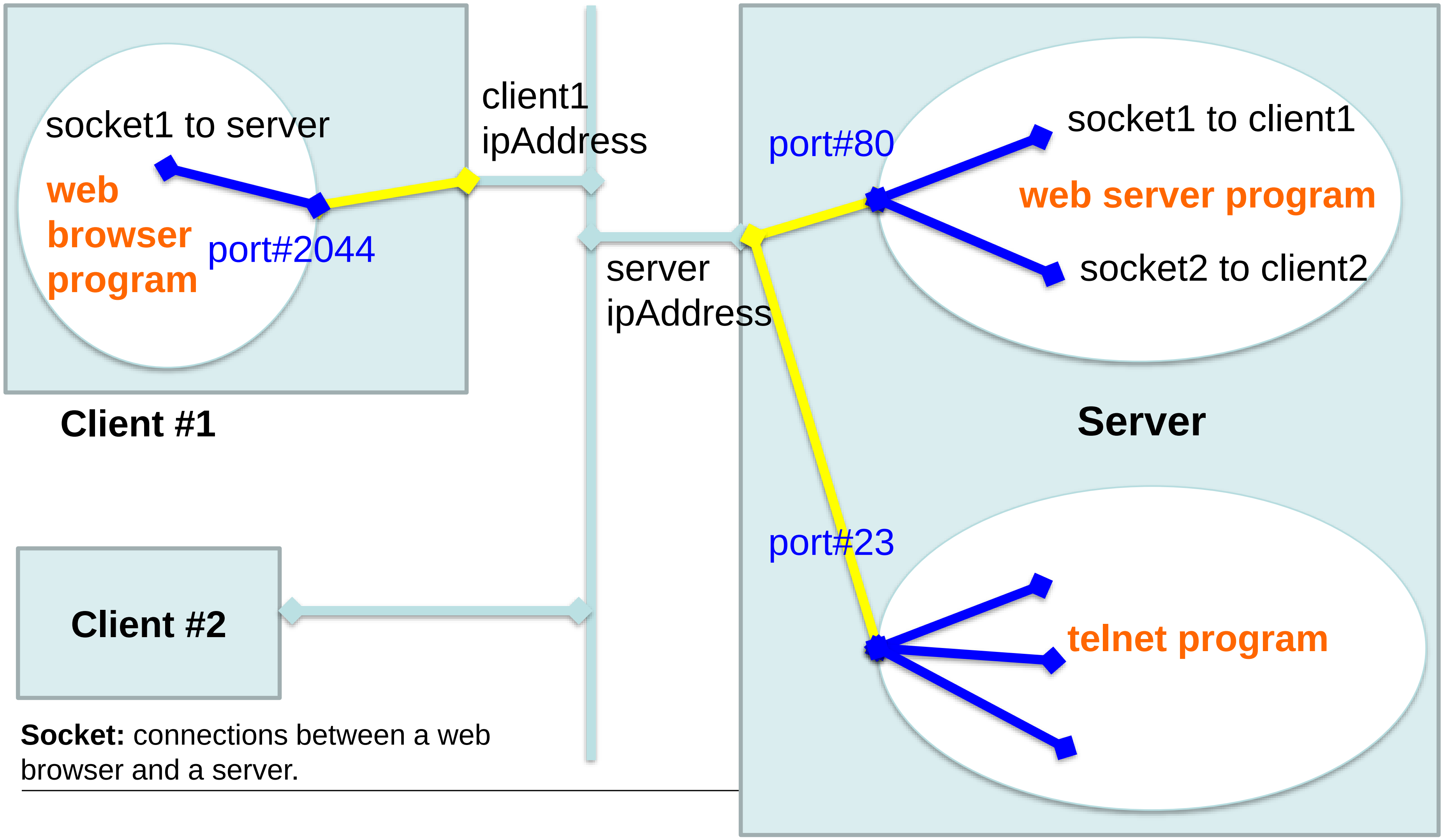
- **GET**: The GET method is used to retrieve information from the given server using a given URI.
  - Requests using GET should only retrieve data and should have no other effect on the data.

Client/Server: HTTP over TCP/IP



Source: https://www.ntu.edu.sg/home/ehchua/programming/webprogramming/HTTP_Basics.html

Web software: Client/Server – Connections



Socket: connections between a web browser and a server.

Client-Side Dynamics (1)

- HTML + Javascript
- Html elements: **forms**
- Html style elements: fonts, headings, breaks
- CSS: uniformly manipulate styles
- JavaScript:
  - manipulate styles (CSS)
  - manipulate html elements
  - validate user data
  - communicate with the server-side programs
- In HTML: `<input id="clkb" type="button" value="Click” onclick="clkF()"/>`
- In Javascript file: `function clkF() { alert("Hello"); }`

Client-Side Dynamics (2)

- Html elements: **View**
- CSS: **Model**
- Javascript: **Controller**
- CSS: A simple mechanism for adding style to Web documents.
  - Look & feel of Webpages
  - Layouts, fonts, text, image size, location
  - Objective: Uniform update
- Javascript as a client side event-driven programming
  - Client-side computations
  - Form validation + warnings
  - Dynamic views

How to add JavaScript to html file?

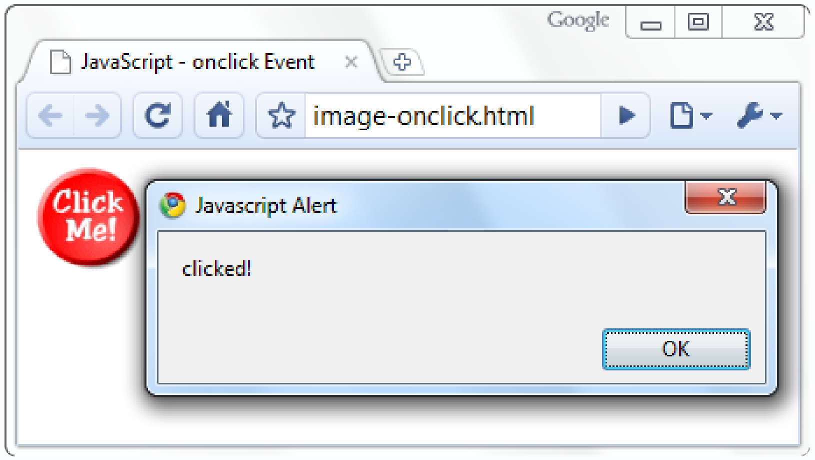
- Include in html file:
  - `<script> your javascript code goes in here </script>`

- Can also include from a separate file:
  - `<script src="./01_example.js"></script>`

- Can include from a remote web site:
  - `<script src="http://…./a.js"></script>`

JavaScript Event Handler – Example

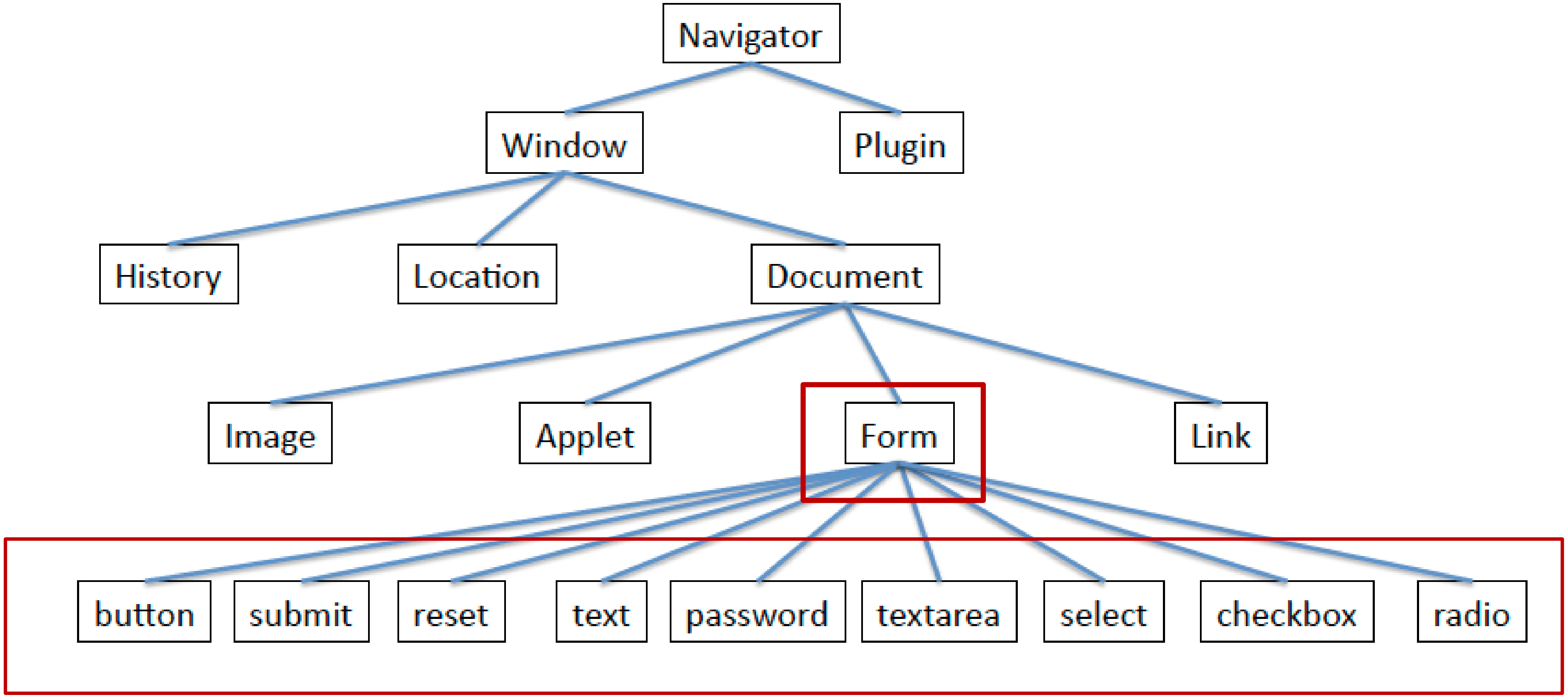
```
<html>
<head>
<script type="text/javascript">
  function test (message) {
    alert(message);
  }
</script>
</head>

<body>
  <img src="logo.gif"
    onclick="test('clicked!')" />
</body>
</html>
```



clicked!

Using **onclick**, we attach **event handlers**.

JavaScript accessibility hierarchy



NODE.JS

Event-driven programming – Node.js

- Open-source, cross-platform **JavaScript** run-time environment that executes JavaScript code **server-side**
  - Historically JavaScript used for client-side programing
- "JavaScript everywhere" paradigm (popular)
  - Unifying web application development
  - Same language for server side and client side scripts.

A JavaScript runtime environment running Google Chrome's V8 engine

Goal is to provide an easy way to build scalable network programs

Why Node.js?

- Non Blocking I/O
- V8 Javascript Engine
  - V8 is Google's open source high-performance JavaScript engine, written in C++ and used in Node.js
- Single Thread with Event Loop
- 40,025 modules: JavaScript libraries you can include in your project
- Different platforms: Windows, Linux, Mac,…
- 1 Language for Frontend and Backend
  - Core in C++ on top of V8
  - Rest of it in javascript
- Active community

Event-driven programming – Node.js

- Event-driven architecture
  - Asynchronous I/O
  - Scalability with many input/output operations
  - Real-time Web applications
    - e.g., real-time communication programs, browser games and data streaming, etc.
- Node.js functions are non-blocking
  - Commands execute concurrently or even in parallel (unlike PHP that commands execute only after previous commands finish)
    - Node.js uses callbacks to signal completion or failure

Asynchronous programming – Node.js

- Node.js uses asynchronous programming (runs single-threaded, **non-blocking**) ➔ very memory efficient
- Handling a file request:
  - In PHP/ASP.net:
    1. Sends the task to the computer's file system.
    2. Waits while the file system opens and reads the file.
    3. Returns the content to the client.
    4. Ready to handle the next request.
  - In Node.js:
    1. Sends the task to the computer's file system.
    2. Ready to handle the next request.
    3. When the file system has opened and read the file, the server returns the content to the client.

Blocking vs. non-blocking: PHP vs. Node.js

To select data from a table in MySQL, use the "SELECT" statement

- PHP:

```
<?php
$result = mysql_query('SELECT * FROM ...');
while($r = mysql_fetch_array($result)){
    // Do something
}

// Wait for query processing to finish...
?>
```

- Node.js:

Error handler

Callback!

```
<script type="text/javascript">
mysql.query('SELECT * FROM ...', function (err, result, fields){
        // Do something
    });

// Don't wait, just continue executing
</script>
```

The third parameter of the callback function is an array containing information about each field in the result object

Blocking vs. non-blocking

- **Blocking**:
  - Read data from file
  - Show data
  - Do other tasks

```
var data = fs.readFileSync( “test.txt” );
console.log( data );
console.log( “Do other tasks” );
```

- **Non-blocking:**
  - Read data from file
    - When read data completed, show data!
  - Do other tasks

Callback!

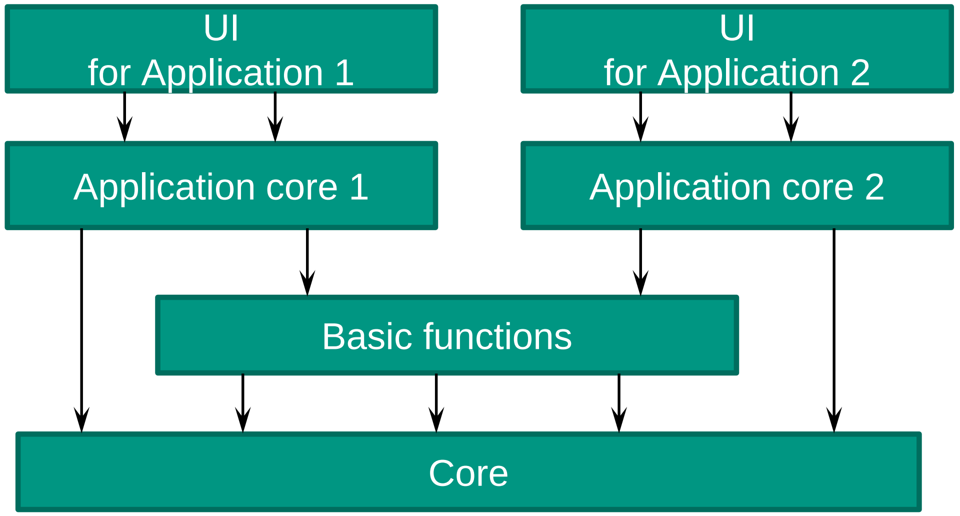
```
fs.readFile( “test.txt”, function( err, data ) {
console.log(data);
});
```

Event-driven programming – When to use Node.js?

- Creation of Web servers and networking tools
  - Ideal for applications that serve a lot of requests but don't use/need lots of computational power per request
- Using JavaScript and a collection of **modules** that handle various core functionality such as:
  - File system I/O, networking (DNS, HTTP, TCP, TLS/SSL, or UDP), binary data (buffers), cryptography functions, data streams, etc.
  - Modules use an API (**interfaces**) designed to **reduce the complexity** of writing server applications

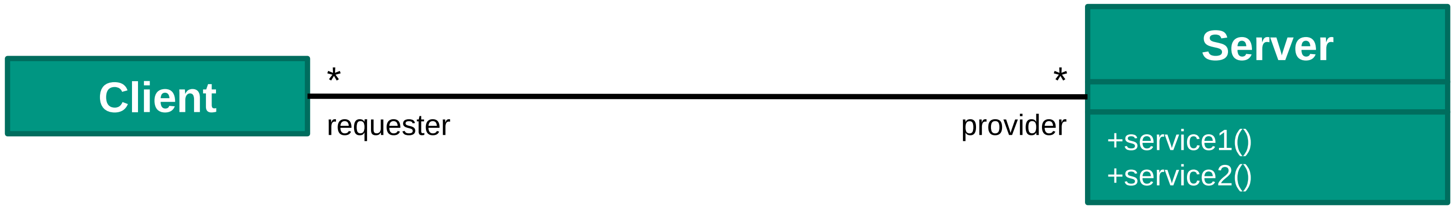
Event-driven architecture (1)

- Architectural patterns (n-tier, client/server, …) may be applied by the design and implementation of applications and systems
  - Transmit events among loosely coupled software components and services
- n-tier architecture (layered architecture, see section Architectural styles)
  - 4-tier:



2-tier architecture – Client/Server

- One or more servers provide services for other subsystems called clients.
- Each client invokes a function of the server which performs the desired service and returns the result.
  - The client must know the interface of the server.
  - Conversely, the server does not need to know the client's interface.
- An example of a 2-tier, distributed architecture:



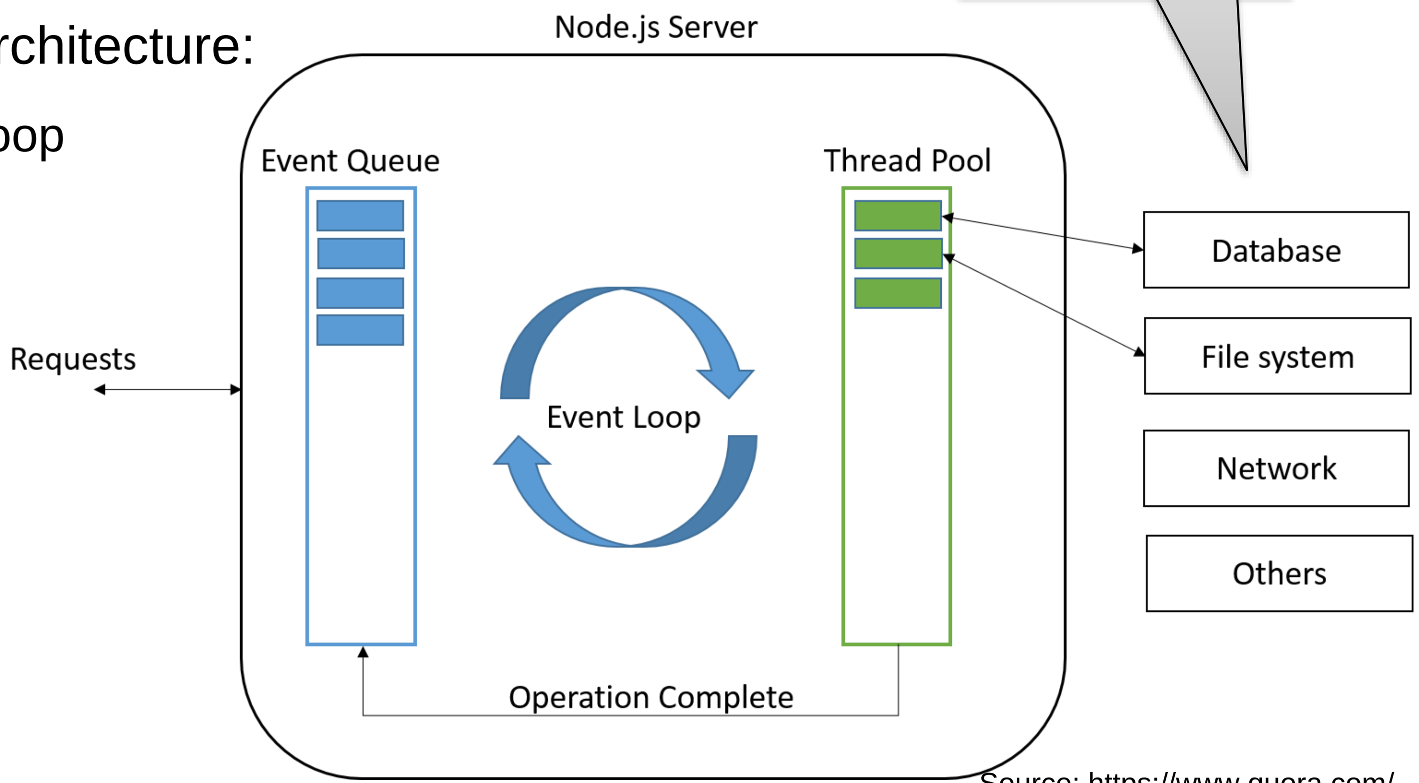
Event-driven architecture (2)

- Event-driven architecture:
  - **Processing loop**
  - **Event queue**
  - **Call-back**

Event-driven architecture – Node.js

non-blocking asynchronous I/O operations

- Event-driven architecture:
  - Processing loop
  - Event queue
  - Call-back



Source: https://www.quora.com/

- Node.js Architecture: The event loop simply iterates over the event queue (a list of events) and callbacks of completed operations.

Event-driven architecture (3)

- Event-driven architecture can complement **service-oriented architecture (SOA)**
- Services can be activated by triggers fired on incoming events.
- SOA is an architecture style that assembles applications from (independent) services (see section Architectural styles)
- Services are considered as central elements of a company (keyword: services)
- Provide encapsulated functionality to other services and applications

Threading – Node.js

- A **single thread**, using non-blocking I/O calls
  - Support tens of thousands of concurrent connections **without** the **cost of thread context switching**
  - Building highly concurrent applications
  - A thread pool handles execution of parallel tasks
- Good for horizontal scaling (lots of request)!
- **Drawback** of the single-threaded approach: **No vertical** scaling by increasing the number of cores (not good for massive parallel computing)
  - Needs additional module: such as cluster, StrongLoop Process Manager, etc.
  - Mitigation: Developers can increase the default number of threads in the thread pool ➔ OS distributes the threads across multiple cores

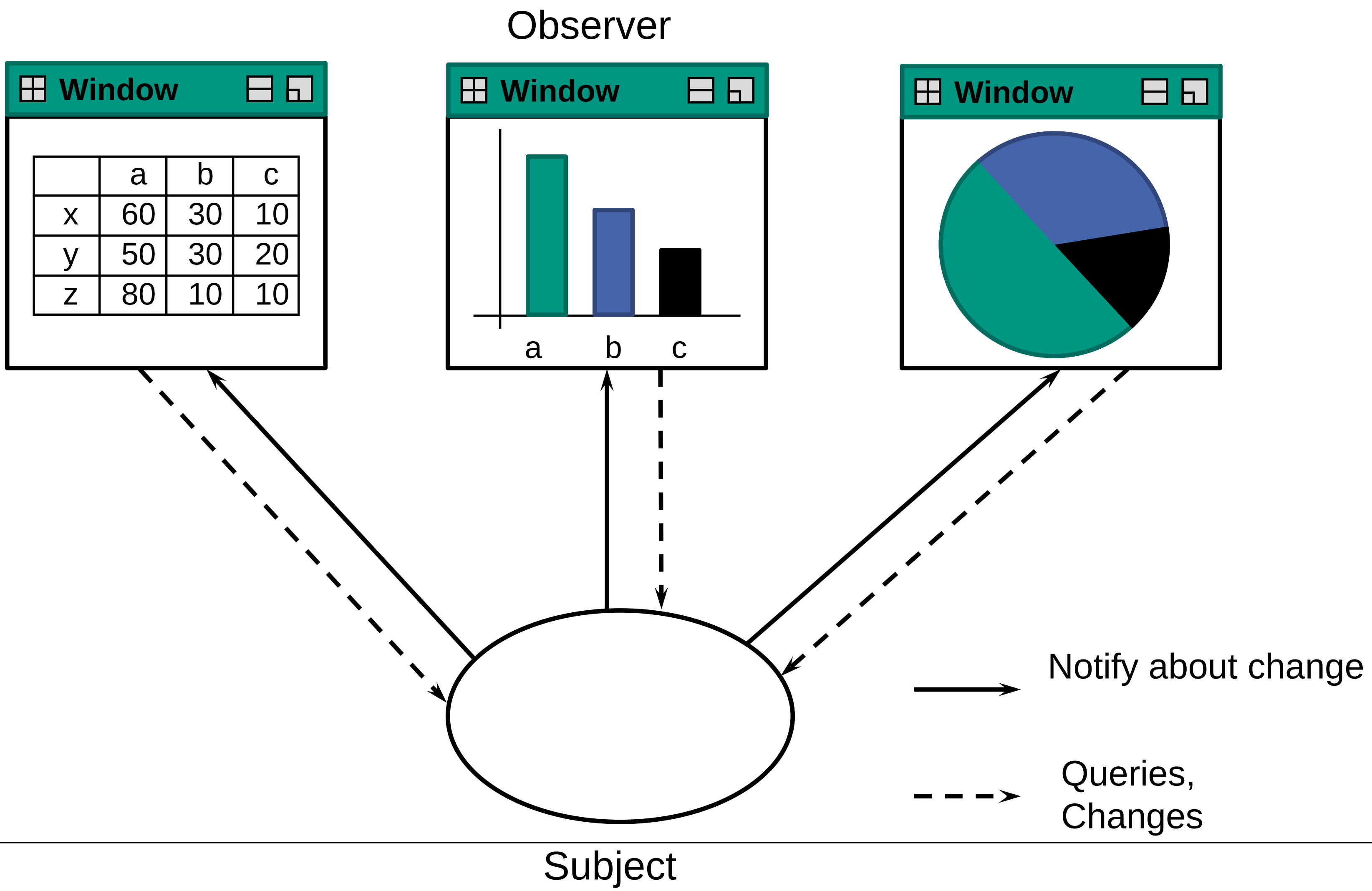
Observer design pattern – Node.js

- A **single thread**, using non-blocking I/O calls

➔ **Observer** design pattern**:** sharing a single thread among all the requests

- Defines a **1-to-n** dependency between objects so that changing a state of an object causes all dependent objects to be **notified** and **updated automatically**.
- One to many relationship
- The many need to know changes in “one” immediately
- Synonyms (aka)
  - Dependence
  - Publisher-subscriber
  - Subject-observer

Observer example – MVC



	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

Thread-based vs. Event-based (Node.js)

Threads	Asynchronous Event-driven
monitor (difficult to program)	event handler (using queue and then processes it)
scheduling	event loop (only one thread, which repeatedly fetches an event)
exported functions	event types accepted by event handler
returning from a procedure (using context switching)	dispatching a reply (no contention and no context switches)
executing a blocking procedure call	dispatching a message, awaiting a reply
waiting on condition variables	awaiting messages

Conclusion:

- Use threads for performance critical applications (kernels)
- Use events for GUI and distributed systems

What can you do with Node.js?

- Node.js file contains tasks and executes them upon set of events
  - Generate dynamic content
  - Create, open and read, or delete files on the server
  - Gather and modify data in the database
  - Collect form data, etc.
- Availability of rich frameworks
  - Angular, Node, Backbone, Ember, etc.
- Ability to keep data in native JSON (JavaScript Object Notation, similar to XML) format in your database
- Very good supportive community
  - Linux Foundation, Google, PayPal, Microsoft, …

Node.js – Example 'Hello World!'

include a module (library), use the require() function with the name of the module

```
var http = require('http');
```

Use the createServer() method to create an HTTP server

Represents the request/response from/to the client

```
//create a server object:
http.createServer(function (req, res) {
  res.write('Hello World!'); //write a response to the client
  res.end(); //end the response
}).listen(8080); //the server object listens on port 8080
```

Writes "Hello World!" if a web browser tries to access your computer on port 8080

Using Existing modules

```
var fs = require('fs');     // here's how
var path = require('path');
// typically an object or a function is returned.

var buf = fs.readdir(process.argv[2],
  function(err, data) {
      for (i = 0; i < data.length; i++) {
          var s = path.extname(data[i]);
          if (s === "." + process.argv[3]) {
              console.log(data[i]);
          }
       } // end of for
  } // end of callback function for readdir
```

Create your own modules – Example

```
exports.myDateTime = function () {
    return Date();
};
var http = require('http');
var dt = require('./myfirstmodule');

http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write("The date and time are currently: " + dt.myDateTime());
    res.end();
}).listen(8080);
```

Save the code above in a file called "myfirstmodule.js"

Include and use the module in any of your Node.js files.

Asynchronous I/O - Example

- **NO WAIT! until read is complete:**

```
var fs = require('fs');
var buf = fs.readFile(process.argv[2],
  function(err, data) { //CALLBACK
    if (err) { return console.log(err); }
    var sArray = data.toString().split("\n");
    console.log(sArray.length-1);
  } );
// NO WAIT! – DO THE NEXT INSTRUCTION RIGHT AWAY
```

Include the File System module:
fs = require('fs‘);
fs.readFile(file, [encoding], [callback]);

fs.readFile() method is used to read files .

Event handling – Event emitter pattern

```
// Instead of only completed event, many events may be fired.
// Handlers can be registered for each event.
var fs = require('fs');
var file = fs.createReadStream('./' + process.argv[2]);

file.on('error', function(err) {
  console.log("Error:" + err);
  throw err;
});

file.on('data', function(data) {
  console.log("Data: " + data);
});
file.on ('end', function() {
  console.log("finished reading all of data");
});
```

readStream object fires events when opening and closing a file

createReadStream fires **error**, **data**, and **end** events

Using **on** function, we attach **event handlers**.

Event emitter API

- **Event types (determined by emitter)**
  - error (special type)
  - data
  - end
- **API**
  - `.on` or `.addListener`
  - `.once` (will be called at most once)
  - `.removeEventListener`
  - `.removeAllEventListeners`

Creating an event emitter – Example

```
// file named myEmitter.js
var util = require('util'); // step 1
var eventEmitter = require('events').EventEmitter; // step 2
var Ticker = function() {
  var self = this;
  setInterval (function() {
       self.emit('tick'); // step 3
    }, 1000) ;
};
util.inherits (Ticker, eventEmitter); // step 4
module.exports = Ticker;
```

Util module provides access to some utility functions.

With ”**events**” you can create-, fire-, and listen for- your own events.

Inherits methods from one function into another