Transcript for the Multiplayer Pong Lecture by Daniel Chcouri

Legend: [Slide title | point 1 / point 2...]

[Multiplayer Pong | Rapid Web App Development]

Hello everyone.
My name is Daniel Chcouri. I am very happy to talk to you today.

I'll present to you a little multiplayer game, which I've developed as a showcase for rapid web app development, with a very cool stack that includes Node.js, Express and Socket.IO.

[Game Terminology]

Let's begin with a short explanation about the game and its terminology.

[Field]

First we have the pong field, to which we will connect with our mobile phones, using a QR code.

The QR code contains a URL to the remote controller web-app, that is linked to the loaded field.

[Controllers]

The controllers will have:
* Up and down buttons
* Middle bar that, once the game begins, will show you which team you're on
* And a bottom bar, that will show the connection status of the remote controller to the server

Once the game begins, all the connected players will be split randomly into two teams: one team will control the left paddle and the other the right one.

The movement of each paddle will be decided by the majority of its team's controllers.

The players of the winning team will be splitted again for another round until a single winner remains.

[The Agent | a snapshot of the agent]

The communication between the controllers and the field will be initialized and managed by the agent server.

I'll go now to explain how the agent was implemented, by going over the stack of technologies it was built upon, and how the controllers and the field use it to communicate.

Then, we will all have a pong tournament to find out which one among us is the "Pong Master".

[Layer 1: Node.js | Server side JS / Chrome's JavaScript interpreter: V8]

The first layer of the stack is a Node.js server.

Node.js is a JavaScript interpreter, that is based on Chrome's excellent open-sourced JavaScript interpreter – the V8, with tools for development of servers, such as HTTP server, TCP server, and so on.

Today developers use Node even to develop programs that aren't limited to servers like the LessCSS compiler, Grunt, and many more.

[I/O operations: Traditional Approach – Blocking I/O | Default approach of most of the languages used for web development / an image that illustrates threads]

When it comes to scaling web-apps to handle multiple concurrent requests, the main issue that web servers have to deal with is how to best utilize the CPU power it has to quickly serve as many connections as possible.

In most programming languages used for web development – such as Python, Ruby, Java or PHP – when we do input/output operations, like reading a file from the disk, requesting a web page, fetching data from the database, and so on, the default behavior is to wait for these operations to return before we continue to do anything else.

Therefor we call them blocking input output operations or simply

blocking IO. While waiting for the data transmission to complete, which can take hundreds of milliseconds, the CPU remains idle, and no other connections can be served by it.

That, in general, is the main obstacle for the full utilization of the CPU, which affects our ability to scale our app.

The traditional approach web servers take to deal with this issue is to fork multiple threads for your app -- one for each open connection -- so when one thread is waiting for an input/output operation, the others can still handle requests.

The traditional approach has CPU overhead, because of the wasted CPU cycles needed for switching between the threads, and a memory overhead for the memory each thread needs for its operations. Also sharing memory between the threads is a complex task.

In general the overhead increases more as the number of connections increase.

Because the cpu can actually work on a single thread at a time it needs to constantly switch between these threads to give some processing time to each one of them. At best this switching scales linearly (O(n)) over the number of threads present, but in practice it's most likely worse.

[Nonblocking IO: Dealing with the C10k problem | Node embraces the Non blocking IO (asynchronous IO) paradigm /
No threads - one single process / A diagram that compares sync to async]

Unlike other languages used for web development, the **default** way to perform operations in Node is non-blocking.

That means that whenever we do an operation that involves input/output, we don't wait for the data transmission from that operation to complete. Instead we tell Node what to do when data is ready. This lets code flow to continue without waiting and let the same app process to serve more connections in the time that in the traditional blocking approach was dead. Node embraces the non-blocking I/O paradigm and encourages developers to code this way. That lets node to run the server in one single process and avoid the multi-threading switching overhead.

The ability to handle more than 10,000 concurrent connections with a single web-server, which is also known as the C10K problem, has been demonstrated on Node with its non-blocking I/O approach since its early beginning.

[All In One | Node serves as both the parser and the server -- our

web-app needs no HTTP server]

Another side effect of this non-blocking approach is the fact that unlike other languages used for web development, Node needs no agent server such as Apache or Nginx. It serves as both the server and the parser.

[Non-blocking I/O | code example for async file reading / JS idioms - code example for JS setTimeout]

The above code demonstrates how a non-blocking file read is being made in Node. fs.readFile gets a file path as a parameter and another parameter which is a callback that will be called once the file read finishes. That callback will receive two attributes: an error object that will be null if there were none, and the file's data.

```
  callback(err, data)
```

Being a hybrid functional programming language, JavaScript fits perfectly with the non-blocking model which JS programmers are familiar with from internal JS functions such as setTimeout(), and from popular tools like jQuery, that implement lots of their functionality in an asynchronous fashion.

[Hello World | code example / shell command that runs the code example]

Here's Node's Hello World server example.

In the first line we load the http module, and then create a server that listens on port 80.

Each time a new request is received, the callback that was passed to createServer will be called, with the request data and a response object, that will be used to produce a response, which in this case is a plain text -- "Hello World".

[Node Ecosystem - NPM | Integral part of Node / Package manager / Dependencies manager/ $ npm publish]

Before I'll continue with the next layer of the game's stack, I'd like to give few words about node's open-source philosophy.

Node follows the unix Rule of Modularity, and encourages its developers to structure their code functionality into separate standalone packages with a well defined interface. While this modularity helps us to maintain complex projects, in a way that I think is superior to the object-oriented paradigm, it also makes it really easy to use these packages as building blocks for other projects.

To help its developers to share their packages Node has an integral package manager called NPM. Adding a package to this package manager is very easy and can be done with a single command to the NPM: $ npm publish.

[Taking Over The World | Snapshot with GitHub stats]

The effectiveness of this approach can be seen with explosion of Node projects in GitHub, in which almost a quarter of the code is in JS.

[Community in Exponential Growth | Diagram]

And with the exponential growth rate of NPM packages, within two years NPM managed to exceed the amount of packages Python and Perl accumulated in more than 10 years.

[Express.js]

Let's return to the game's stack.

The next layer is the node package Express.js, which in the Pong demonstration was used only to serve the static HTML pages of the field and the remote controllers. But is actually a very powerful micro-framework that allows us to develop web servers quickly.

Here you can see an example for an Express based Hello World server. The first two lines load Express, and then we define our Hello World response under the root route for GET requests.

See how express simplifies the plain node's Hello World server we've seen earlier.

[Middlewares | Example for static files middleware]

Aside for its HTTP routing, Express also has a convenient middlewares model. In the following example you can see a use of its static directory middleware that maps the static files of the directory routes under "/shared_dir".

This middleware was used to serve the static files of the Pong app.

Express has some more built-in middlewares for cookies handling, redirecting, formatting, security, and so on. It is very easy to write your own express middlewares.

[Socket.IO | Bi-directional I/O / Pure JS / Both server and client /Event driven model / JSON / Fallbacks]

The last layer is the Socket.IO package. An implementation of a bi-directional I/O sockets written in pure JS, that gives us a complete solution for continuous interactions between the server and its clients.

Data in Socket.IO is being passed in an event driven model as JSONs.

Socket.IO has fallbacks for browsers that don't support web-sockets.

[Socket.IO Example | Server side code / Client side code]

The following example shows how the clients and the server interacts with each other with Socket.IO.

Let's begin with the server code: the code begins with the initialization of a Socket.IO server that listens on port 80.

Then we bind a callback to the "connection" event, that Socket.IO emits every time a new client connects. This event passes to the callback the socket object that represents the socket connection with the client.

We then emit on the socket an event named "news" with a JSON data object. The event name is arbitrary and we can pick whatever name we want.

We also tell the socket to bind to the "my other event" event and to log the data it receives from it.

On the client side we begin by loading the Socket.IO library, then we initialize the connection and bind to the news event.

When a news event received from the server we log the data received by it (which will be the hello world object we've passed) and emit the "my other event" event with data that will be received by the event listener we defined by the server.

[Determining Paddles Movement / Pic of the game terminology]

I'll go now into the app's actual code, to demonstrate how the correct movement of each paddle is determined from the controllers, and how it is being passed to the field by the agent.

We will have one socket connected to the field – the field socket. And one socket connected to each controller.

[Controller | The controller code]

On the controller we bind to the touchstart and touchend events of the up and down buttons. Once touchstart fires we emit a "direction" event to the socket with the: field ID, controller ID, controller side and the direction: 1 for up -1 for down.
For the touchend event we emit the direction event with the same parameters as before but with the direction set to 0.

[Agent | The agent code]

And here's the binding to the direction event on the agent Socket.IO server:

As you can see the callback we define for this event receives the field ID, controller ID, side, and direction as its attributes.

We determine the field socket for the received field_id and update the controller direction in an array that holds the status of all the controllers sockets of that field.

We sum all the controllers direction for the given side.

And then we emit to the field_socket the direction event with the side and direction: 1 if the sum was bigger than 0, -1 otherwise.

[Field | The field code]

On the field we simply move the paddle of the received side in the specified direction, every time the direction event received from the agent.

END