JavaScript servers

Concurrency

Two reasons for wanting concurrency are ***responsiveness*** and ***fairness***

Responsiveness means that different parts of a program (e.g. the GUI in an application, or a request handler in a server) need to be active at the same time

Fairness means that the different parts (e.g. different server requests) all make reasonable progress

Neither of these require ***parallelism*** such as multiple cores, it is enough to avoid idleness in a single processor, sharing its power among the different parts

Approaches to concurrency

One approach is shared data (e.g. Java) with problems: races, deadlock, livelock, starvation, non-determinism, unrepeatability, undebuggability

Another is channels (e.g. Go) which makes it easier to reason about concurrency, but it has ***all the same problems***

The third is event loops (JavaScript) which have far fewer problems, but are deliberately non-parallel, i.e. programs can't take advantage of multi-core processors

e.com/datacenters

Servers

Node is used, more than anything else, to write servers

It uses the event-loop approach to concurrency, where
(a) interactions are asynchronous
(b) the programmer splits the code into small transactions, mostly short callback functions
(c) outstanding transactions are treated as a queue executed in an 'event loop'

How do you make use of multiple cores? You just run multiple identical servers as separate processes

Security

A networked server is a program running with ***your*** permissions which can be used by ***other*** people

So you must think about security ***before*** you write any server code

One security precaution lies in using a port number above 1024, which avoids standard services and is stopped by firewalls (and means you don't need root/administrator privileges)

Another is to limit what the server can possibly do (e.g. only access files inside one directory/folder)

More security

More security can be added by using encryption, which prevents hackers from tampering with legitimate connections

Even more can be added by requiring clients to authenticate, to prevent hackers from connecting directly

For unencrypted networking, see the net module, and for encrypted and optionally authenticated networking see the TLS/SSL module

Resources

Don't run a server on a shared computer like snowy because (a) it unfairly uses resources which are meant for other things and (b) port numbers would clash

It is OK to run a server on a workstation, but don't leave it running when you logout

If you run a server on your own computer, take great care over security, because you may not be protected by the University firewall

A simple echo server

```
// Create an echo server on a fixed port.
// Security: (a) run inside firewall (b)
// don't extend (c) don't leave running.
"use strict";
var net = require('net');
var port = 8888;

// Create the server object,
// which waits for client connections.
var server = net.createServer(connect);
server.listen(port);

// Called when a connect request comes in
// The argument is a socket.
function connect(client) {
  client.on('data', get);
  function get(t) { echo(client, t); }
}
```

Net module and port

```
"use strict";
var net = require('net');
var port = 8888;
```

The `require` function is used to import the `net` library module

A port number is chosen, from 1024 to 65535 so that
(a) it doesn't require root/administrator access to use it
(b) it doesn't clash with any standard services
(c) the server is protected by normal firewalls

Start

```
var server = net.createServer(connect);
server.listen(port);
```

The `createServer` and `listen` calls return straight away

Whenever an incoming connection request arrives from a client on the given port, the `connect` callback function will be called

Connect

```
function connect(client) {
  client.on('data', get);
  function get(t) { echo(client, t); }
}
```

When a client sends a connect request, this is called

From then on, whenever the same client sends data, the `get` method is called

Because `get` is inside `connect`, it has access to the `client` local variable via a closure

In order to make `echo` a separate function, `get` is a simple one-liner which passes on the `client` object

Interact

```
function echo(client, text) {
  client.write(text);
}
```

The echo function represents the interaction with the client

Whenever the client sends data, presumably a request, the server responds by sending the same text back to the client

Clients

The `client` variable in the server code is really a socket object, i.e. "the current state of the connection with the client"

To find out what you can do with it, go to the Node.js web site, then follow the API DOCS link, then find the `net` module, and then the `socket` section

Security and flexibility

The echo server's security precautions are the University firewall or other suitable firewall, and the fact that its actions are harmless

This server can *already* handle multiple simultaneous clients

Testing server with telnet

```
js> node p10.js
```

```
js> telnet localhost 8888
Trying 127.0.0.1...
Connected to localhost... (127.0.0.1).
Escape character is '^]'.
Line one
Line one
Line two
Line two
^]
telnet> quit
js>
```

Testing

The test uses two terminal windows – the server can write debug messages into its own window

It is not recommended to run the server in the background (e.g. using &) because then it would keep running when you logout, which would tie up the port

`telnet` is an old command to connect to a server and type at it (like `ssh` now)

For non-telnet testing, let's write a JavaScript client

A test client

```
// Interactive test client.
"use strict";
var net = require('net');

// Connect to the server
var host = 'localhost', port = 8888;
var options = { host: host, port: port };
var server = net.connect(options, go);

// When data comes from the keyboard, call obey
// When data comes from the server, call receive
function go() {
  process.stdin.setEncoding('utf8');
  process.stdin.on('data', obey);
  server.setEncoding('utf8');
  server.on('data', receive);
}
```

The test client

`localhost` means the server is running on the same computer as the client, otherwise use a domain name or ip address

`setEncoding` is used on both the server socket and `stdin` to specify text, not binary

This is not a web or http or https or websocket server/client - none of those are needed in this course

It is a TCP socket server/client - streams of bytes go in each direction, and newlines can be used to separate requests/responses

Java test client

```
import java.net.*;
import java.io.*;
import java.util.*;

class Client
{
  private String host = "localhost";
  private int port = 8888;

  public static void main(String[] args)
  throws Exception {
    new Client().run();
  }

  void run() throws Exception {
    Socket server = new Socket(host, port);
    PrintWriter to = new PrintWriter
      (server.getOutputStream(), true);
```

Protocols

The TCP protocol is a low level one which sends ***streams of bytes*** over the net

Usually, they are thought of as requests and responses, so how is a stream divided into a series of these?

The client and server need an agreement (higher level protocol)

Line protocol

Here's a simple protocol

Every request consists of one line, terminated by a newline

Every response consists of one line, terminated by a newline

To make it work:

- agree what a newline is (beware OS differences)
- make sure writing a newline empties the buffer
- beware large responses coming in several packets

Multiple clients

Suppose you want to write a coordinating server

It has to take in a request from one client, and respond by sending messages to other clients

So how can the server keep track of all the currently connected clients?

Let's try an experiment to see what happens when two clients connect to our test server

Changing the server

Let's change the server to print out the computer and the port number of the client:

```
...
function connect(client) {
  console.log(client.remoteAddress, client.remotePort);
  client.on('data', get);
  function get(t) { echo(client, t); }
}
...
```

Connect twice

```
js> node server.js
::ffff:127.0.0.1 50851
::ffff:127.0.0.1 50853
```

```
js> node client.js
```

```
js> node client.js
```

A client sends data to the server at port 8888, but it also chooses an unused port number for itself, so the server knows how to send data back

Address and port

The string `::ffff:127.0.0.1` is the address of the client (it is the IP address in version 6 of the Internet Protocol)

This particular address is the numerical equivalent of `localhost` which means "this computer" - the one the server is running on

The address plus the port number forms a unique id for the client, e.g.:

```
id = client.remoteAddress + "@" + client.remotePort
```

Keeping a table

Now the server can keep a table of clients, indexed by id

```
var clients = {};
...
// on connect
var id = client.remoteAddress + "@" + ...;
clients[id] = client;
...
// broadcast
for (var id in clients) send(clients[id]);
```

Typically, when you have info to send, you don't broadcast it, you send it to a specific set of clients

Event emitters

Instead of writing a loop which runs through all clients and decides which ones to inform of something, an *event emitter* can be used to keep track of "who needs to be told about what" (observer pattern again)

```
"use strict";
var events = require("events");
var informer = new events.EventEmitter();
informer.on('meet', greet);
informer.emit('meet', "hi");
function greet(text) {
  console.log("Someone says:", text);
}
```