6

Performing a Load Balancing Act

In this chapter, we will cover the following recipes:

* Doing Load Balancing with Nginx Server
* Using Node.JS Cluster
* Using Redis to Pass Events Between Nodes
* Using Memcached to Manage Multiple Nodes
* Using RabbitMQ to Message Events Across Nodes

# Introduction

As the audience of an application grows, it is important to make sure that the application is scalable. On the server-side, this means that we might want to distribute our application across multiple threads or node instances.

The issue with distributing your application across nodes is that when we emit a message, it will only be received by one of the distributed servers. Sockets that are not connected to the same server as the one that receives the message will not be able to receive it without some additional handling. Luckily, there are some great ways to pass session data between servers using a caching system such as Redis, Memcache or RabbitMQ. By using adapters for one of these distributed caching mechanisms, we can easily scale our servers without compromising our Socket.IO connection.

# Doing Load Balancing with Nginx Server

Nginx is a free, open-sourced, high performance HTTP server and reversed proxy. Unlike traditional servers, Nginx doesn't rely on threads to handle requests. Instead, it uses a much more scalable asynchronous architecture. This architecture uses small and predictable amounts of memory under load.

We can use Nginx to load balance our Node servers and if it is configured correctly, we won’t have to worry about requests being lost between the original handshake and the callback when events are received.

## Getting Ready...

Before we can do effective load balancing with the Nginx server, we will need to install it. Nginx can be installed with Homebrew:

brew install nginx

Once Nginx is installed, you can start it by running:

sudo nginx

You can also stop it by running:

sudo nginx –s stop

## How To Do It...

To load balance a Socket.IO app using Nginx, follow these steps:

1. Open your Nginx config file. This will most likely be located at /usr/local/etc/nginx/nginx.conf
2. Find the http section of the nginx.conf file and add your list of upstream nodes:

upstream io\_nodes {

ip\_hash;

server 127.0.0.1:5000;

server 127.0.0.1:5001;

server 127.0.0.1:5002;

server 127.0.0.1:5003;

}

1. In server / location, configure your proxy to pass all of the headers from the original request:

location / {

proxy\_set\_header Upgrade $http\_upgrade;

proxy\_set\_header Connection "upgrade";

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header Host $host;

proxy\_http\_version 1.1;

proxy\_pass http://io\_nodes;

}

1. Start your nginx server using sudo nginx. By default, your server will listen on port 8080. You can switch it to a different port if you need to in the nginx.conf file.

## How It Works...

The Nginx server will proxy through to your Node server or servers. It will dynamically decide which server to hit by looking at the worker\_processes and the worker\_connections, which is inside the events:

events {

worker\_connections 1024;

}

The worker\_processes indicates the number of workers that Nginx should user. By default, it is set to 1, so it should be bumped to allow multiple workers. You can optimize both of these as needed.

# Using Node.js Cluster

Node.js comes with a “cluster” package that can be used to run Node on multiple threads as opposed to the single thread that it runs on normally. The child processes that cluster creates are able to all run on the same port, which means that you can effectively load balance without running your server on multiple ports.

Unfortunately, there is some boilerplate needed to determine the number of available CPUs to run Node processes in and forking the original node. For that, we can use a module called “Sticky Session”, which is a load balancer that automatically spawns and manages multiple node sessions using the cluster module.

## Getting Ready...

For this recipe, we will be using the Sticky Session npm module. It can be installed by running npm install sticky-session.

## How To Do It...

To create a Node server using Sticky Session, follow these steps:

1. Require your dependencies. This will include the sticky session module as well as cluster, which installs with Node. We will be using cluster to determine if our server is the parent or a child.

var sticky = require('sticky-session'),

http = require('http'),

express = require('express'),

socketIO = require('socket.io'),

cluster = require('cluster');

1. Now, call the sticky() function and create your app inside the function that you pass in as your first argument. Anything that is passed into the sticky() function will only be executed by children.

var server = sticky(function() {

var app = express(), io;

server = http.Server(app);

io = socketIO(server);

// Add your socket.IO connection logic here

return server;

});

1. Now, we can start the server that we return from the sticky() function. When we start the server up, we should see a master server and multiple child servers logging their greeting to the console.

server.listen(5000, function() {

if (cluster.isMaster) {

console.log('Master server started on port 5000');

} else {

console.log('- Child server started on port 5000');

}

});

## How It Works...

By default, Socket.IO performs multiple requests to create a handshake and create a connection to the client. In a distributed environment, each request has the potential to land on a different worker than the previous request. This will break the handshake protocol and Socket.IO will not work.

The Sticky Sessions module is able to get around this issue by always routing the client to the same worker based on the client’s IP Address. This guarantees that each new request will land on the same worker and everything will work as expected.

# Using Redis to Pass Events Between Nodes

Now that we are able to run multiple Nodes simultaneously with Socket.IO and not loose our socket connection between events, we also need a way to ensure that when an event is emitted on one node, it is also emitted across all of our other nodes.

Socket.IO has an answer for this. Socket.IO uses an interface called an adapter to route messages. Socket.IO allows us to use something other than the default memory-based adapter, so we can use our own instead. For a distributed system, we will need to use an adapter that lives outside of our server nodes.

Redis is a perfect solution for this problem. Redis is key-value store and cache lives outside of our web servers. This means that we can spin server instances up and down and the data that is stored in Redis will not be lost. By plugging Redis into our Socket.IO adapter, we can propagate events across our Nodes rather painlessly.

## Getting Ready...

First, we will need to have an instance of Redis running. Redis can be downloaded from their website at http://redis.io.

There is a very capable NPM package called “socket.io-redis” that we can use to pass in as our adapter. It can be installed from NPM:

npm install socket.io-redis –save

We will also need the Socket.IO emitter to help propagate our events across servers:

npm install socket.io-emitter --save

## How To Do It...

Follow these steps to pass events between nodes using Redis:

1. First, we will need to require the socket.io-redis NPM package on our server and pass in our configuration to it.

var redisConfig = {

host: 'localhost',

port: 6379

}, server, io;

var express = require('express'),

http = require('http'),

socketIO = require('socket.io'),

redis = require('socket.io-redis'),

emitter = require('socket.io-emitter')(redisConfig),

app = express();

1. To test that this works, we want to be able to run our server on multiple ports. This will allow us to emit a message on one port and check to make sure it has successfully propagated to every other port that is listening for the same event. This can be accomplished by using an environmental variable to dynamically set the port number that we will listen to. The environmental variables in node can be accessed using the process.env object.

if (!process.env.PORT) {

throw new Error('Please specify a PORT number, ie: PORT=5000 node server');

}

app.get('/', function (req, res) {

res.sendFile(\_\_dirname + '/index.html');

});

server = http.Server(app);

server.listen(process.env.PORT);

1. Now, we will listen for a message.sent event from the server and use the socket.io-emitter NPM package to emit the event to every socket on any server we have listening to our Redis instance.

io = socketIO(server);

io.adapter(redis(redisConfig));

io.on('connection', function (socket) {

socket.on('message.sent', function (port) {

emitter.emit('message.received', port);

});

});

1. Now, we need to add a template on the client-side. This template will be responsible for emitting messages to the server with the port number the message originates from. The template will also add an item to the list of messages any time a message comes from the server. We will print the port number in the list to prove that we are affectively communicating with Socket.IO across multiple server instances.

<div id="messages"></div>

<button id="broadcast">Broadcast</button>

<script src="/socket.io/socket.io.js"></script>

<script>

// The port number needs to be dynamic so we can

// Use this page on any port

var port = window.location.port,

socket = io.connect('http://localhost:' + port);

// Add new messages to the list

socket.on('message.received', function (port) {

var message = document.createElement('div');

message.innerHTML = `Received message from port ${port}`;

document.getElementById('messages').appendChild(message);

});

// When the "broadcast" button is clicked,

// We will send a message to the server to render the message

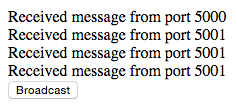
document.getElementById('broadcast').addEventListener('click', function () {

socket.emit('message.sent', port);

});

</script>

1. To test that everything worked as expected, we need to start at least two servers listening on completely separate ports. This can be achieved by passing the environmental variables in before the command run Node. In one terminal window, run: PORT=5000 node server and in a separate terminal window, run PORT=5001 node server.
2. Navigate to each of your two servers and click the broadcast button to send messages. You should see the messages for each port on both servers even though they are completely separate processes. The output will look something like this:



Insert Image B04893\_06\_01.png

## How It Works...

In our application, when we receive an event on the server-side, we are emitting an event to the emitter instead of the io.sockets. The emitter is responsible for taking our event and setting it in Redis. The Redis adapter picks up from there and reacts to changes by doing a io.sockets.emit() on each of our servers behind the scenes.

Redis is essentially its own server, which multiple other servers can access to set and get data from the exposed key-value storage. Typically, Redis is used for caching and persisting data that may not need to be in the persistent database. This is often limited to session information, but in our case, we are using it to set the information that the sockets are emitting from the client.

Fortunately, the Redis adapter for Socket.IO already handles all of the complicated internals of reacting to newly available data in the Redis data store and propagating it across multiple server instances.

While there are many ways to use Socket.IO with a multiple load-balanced servers, the Redis adapter definitely makes it the easiest approach for handling multiple server instances.

# Using Memcached to Manage Multiple Nodes

Memcached is an in-memory key-value store, which is designed for handling small chunks of arbitrary data. Typically, Memcached is used for caching server and API responses in memory so that we can render the cached data instead of hitting the database and waiting for a response if the data has already been persisted in the cache.

Like Redis, Memcached is run in a separate server instance outside of the web server. This means that we can use it in the same way that we used Redis to propagate events across multiple server nodes.

There are a couple of projects on Github with the intension of providing the ability to use Memcached with Socket.IO, but at the time of this writing, there were none that had been updated after the 1.0 release of Socket.IO and the implementations all appeared to be either incomplete or buggy. The good news is that the lack of quality Memcached Socket.IO adapters will give us an opportunity to explore how we can use Memcached in our project at a lower level.

## Getting Ready...

For this recipe, we will need to install Memcached on our machine or run it remotely from a service such as Heroku. The latest version of Memcached can be downloaded from their website at http://www.memcached.org/downloads.

We will be using the memjs NPM package to access Memcached in Node. This can be installed by running npm install memjs –save in your terminal.

## How To Do It...

To use Memcached to load-balance Socket.IO processes, follow these steps:

1. We will create an adapter.js file, to abstract some of our internal Memcached work. On our server, we will actually require this file instead of Socket.IO itself. We are basically overwriting a few Socket.IO methods to get it to send data to Memcached instead of client-side sockets. We will then ping Memcached on an interval to see if there is new data and if it is, we will call the default emit() method on Socket.IO which we pass over originally.

var memjs = require('memjs'),

socketIO = require('socket.io'),

\_ = require('lodash');

function Memcached (uri, options) {

// Create a Memcached connection

var client = memjs.Client.create(uri, options);

// This will be the new io method to use instead

// of the old one

return function (server) {

var io = socketIO(server);

// We will be overriding the emit function, so we'll make

// a copy of it for us to use later on

var \_emit = io.emit.bind(io);

// This value represents the last time that the Memcached

// value was emitted. It will be updated each time there

// is new data.

var \_lasttime = new Date().getTime();

function processDataFromCached (err, value, key) {

// If there is no value, or it can't be parsed to a

// string, we need to return it so it doesn't

// break everything

if (!value || !value.toString()) {

return;

}

// Parse the data back from a string into JSON

value = JSON.parse(value && value.toString());

// If the data has not been emitted on this server

if (value.time > \_lasttime) {

// Update the time stamp

\_lasttime = value.time;

// Emit the new data using the real

// socket.io emit function

\_emit(value.topic, value.value);

}

}

function checkDataCache () {

// Get the socket.io key from Memcached

client.get('socket.io', processDataFromCached);

}

// We will intercept the default behavior of the emit

// function. Not to worry, though. We are holding onto the

// real socket.io emit function and calling it \_emit. We

// will call the \_emit function when we check the cache

// and notice a change

io.emit = function (topic, value) {

client.set('socket.io', JSON.stringify({

topic: topic,

value: value,

// We will use the time stamp to compare to the

// last time the cache was updated. If it is newer

// than the value of \_lasttime, we will emit the

// new change

time: new Date().getTime()

}));

};

// Check the cache on an interval to see if there is a

// new message

setInterval(checkDataCache, 500);

// Return our modified io object

return io;

}

}

module.exports = Memcached;

1. The consumer.js file will be responsible for consuming the adapter and using it to emit events across multiple servers. We will export the entire consumer as a function so that we can pass in a port number to start the server on. You may also notice that the MEMCACHED\_URI, MEMCACHED\_USERNAME and MEMCACHED\_PASSWORD are being pulled from our environmental variables, so you will need to create those variables to connect to the correct instance of Memcached.

module.exports = function (port) {

var express = require('express'),

http = require('http'),

socketIO = require('./adapter')

(process.env.MEMCACHED\_URI, {

username: process.env.MEMCACHED\_USERNAME,

password: process.env.MEMCACHED\_PASSWORD

}),

app = express();

console.log('Starting server on port ' + port);

app.use(express.static(\_\_dirname));

var server = http.Server(app);

server.listen(port);

var io = socketIO(server);

io.on('connection', function (socket) {

socket.on('message.sent', function (port) {

io.emit('message.received', port);

});

});

};

1. The server.js will be extremely simple. It will just instantiate a couple of the consumers on different ports.

var consumer = require('./consumer');

consumer(5555);

consumer(5556);

1. Now, we will create an index.html file which will load a couple of iframes on the two port numbers we are running our servers on. This way, we can see both servers interacting in real-time. Please note that there are various ways to get around cross-domain issues with iframes these days. Going through WebSockets that communicate over a load-balanced data store is probably not the most ideal way to pull it off, but it is an interesting exercise.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Memcached</title>

<style media="screen">

body {

margin: 0px;

padding: 0px;

}

iframe {

width: 45%;

height: 600px;

border: 0px;

}

</style>

</head>

<body>

<iframe src="http://localhost:5555/iframe.html"></iframe>

<iframe src="http://localhost:5556/iframe.html"></iframe>

</body>

</html>

1. Now, we will need our index.html files to render messages that come in and broadcast messages. This will allow us to prove that we are indeed communicating on multiple Socket.IO instances across multiple servers.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Memcached</title>

</head>

<body>

<h1>This iframe is on port #<span id="port-number"></span></h1>

<div id="messages"></div>

<hr>

<button id="broadcast">Broadcast</button>

<script src="/socket.io/socket.io.js"></script>

<script>

// The port number needs to be dynamic so we can

// Use this page on any port

var port = window.location.port,

socket = io.connect('http://localhost:' + port);

document.getElementById('port-number').innerHTML = port;

// Add new messages to the list

socket.on('message.received', function (port) {

var message = document.createElement('div');

message.innerHTML = `Received message from port ${port}`;

document.getElementById('messages').appendChild(message);

});

// When the "broadcast" button is clicked,

// We will send a message to the server to render the message

document.getElementById('broadcast').addEventListener('click', function () {

socket.emit('message.sent', port);

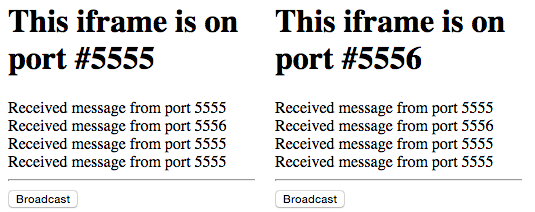
});

</script>

</body>

</html>

1. Start your server by running node server. You can navigated to localhost:5555 in your browser and click the “broadcast” button in either iframe to broadcast a message in both domains. The result should look something like this:



Insert Image B04893\_06\_02.png

## How It Works...

The premise that we are using is actually fairly simple. We are setting data on a server instance outside of our web servers that all of our web server have access to. We can then check the data on each of our web servers and determine whether or not anything has updated since the last time we checked the server for new data. While we are using Memcached for this, it could really be used for any in-memory data store that doesn’t live on our web server.

# Using RabbitMQ to Message Events Across Nodes

RabbitMQ is a message-oriented middleware that implements Advanced Message Queuing Protocol, or AMQP for extremely robust messaging across a distributed system.

In this recipe, we will use RabbitMQ to allow us to use multiple servers and broadcast messages across them. One big advantage that RabbitMQ holds over something like Memcached for this sort of task is that it is actually meant to be used for publish / subscribe style events. That means that we won’t have to ping a server to determine if there are changes, RabbitMQ will emit changes as they happen, which makes RabbitMQ a perfect solution for the existing style of Socket.IO.

At the time of this writing, there were no satisfactory open source RabbitMQ adapters for Socket.IO. This means that we will need to write our own abstraction.

## Getting Ready...

For this recipe, we will need to install RabbitMQ and have it running locally on our machine. It can be installed at <https://www.rabbitmq.com/download.html>.

There are a few officially supported RabbitMQ clients for Java, C# and Erlang, but for Node, we will have to settle for one that wasn’t created by RabbitMQ. One package that works perfectly for this is amqplib. It can be installed from NPM by running npm install amqplib –save.

## How To Do It...

To use RabbitMQ to message events across nodes, follow these steps:

1. First, we will need to create an adapter.js file. This will be an abstraction around Socket.IO. We will override the emit() function, but leave the rest of the Socket.IO behavior untouched. When an event is emitted, we will emit it to RabbitMQ instead of sending it directly to the client. We will be listening for an event from RabbitMQ so that whenever an event is triggered from any server, every other server will receive the event and be able to emit the event using the standard Socket.IO emit() method.

var amqp = require('amqplib'),

socketIO = require('socket.io'),

\_ = require('lodash');

function RabbitMQ (uri, options) {

// This will become a reference to the RabbitMQ connection

var client;

// This will be the new io method to use instead of the

// old one

return function (server) {

var io = socketIO(server);

// We will be overriding the emit function, so we'll make

// a copy of it for us to use later on

var \_emit = io.emit.bind(io);

function processDataFromCached (message) {

// If there is no value, or it can't be parsed to a

// string, we need to return it so it doesn't

// break everything

if (!message || !message.content) {

return;

}

// Parse the data back from a string into

// a JSON object

var value = JSON.parse(message.content.toString());

// Emit the new data using the real socket.io

// emit function

\_emit(value.topic, value.value);

// Ack it back to the RabbitMQ que

client.ack(message);

}

// We will intercept the default behavior of the emit

// function. Not to worry, though. We are holding onto the

// real socket.io emit function and calling it \_emit. We

// will call the \_emit function when we check the cache

// and notice a change

io.emit = function (topic, value) {

if (!client) {

return console.warn('The RabbitMQ channel is not available...');

}

// Loop over the ports to emit to

\_.each(options.sendTo, function (portNumber) {

client.sendToQueue('socket.io', new Buffer(JSON.stringify({

topic: topic,

value: value

})));

});

};

// Create a RabbitMQ connection

amqp.connect(uri).then(function(connection) {

// Create a channel from the connection

connection.createChannel().then(function(channel) {

// Store a reference to the channel to

// use on the outside

client = channel;

// Listen for events on the que

client.assertQueue('socket.io', {

durable: false

}).then(function () {

// Consume new events

client.consume('socket.io', processDataFromCached);

});

});

});

// Return our modified io object

return io;

}

}

module.exports = RabbitMQ;

1. The consumer.js file will give us the ability to start a server using a specified port that we can pass in. That way, we can start multiple servers at once from the same process.

module.exports = function (port) {

var express = require('express'),

http = require('http'),

socketIO = require('./adapter')('amqp://localhost', {

sendTo: [5555, 5556]

}),

app = express();

console.log('Starting server on port ' + port);

app.use(express.static(\_\_dirname));

var server = http.Server(app);

server.listen(port);

var io = socketIO(server);

io.on('connection', function (socket) {

socket.on('message.sent', function (port) {

io.emit('message.received', port);

});

});

};

1. The server.js will just be responsible for starting some servers by passing port numbers into the consumer.

var consumer = require('./consumer');

consumer(5555);

consumer(5556);

1. The index.html file will be a wrapper to hold some iframes that will be able to communicate cross-domain and cross-server using RabbitMQ.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Memcached</title>

<style media="screen">

body {

margin: 0px;

padding: 0px;

}

iframe {

width: 45%;

height: 600px;

border: 0px;

}

</style>

</head>

<body>

<iframe src="http://localhost:5555/iframe.html"></iframe>

<iframe src="http://localhost:5556/iframe.html"></iframe>

</body>

</html>

1. Finally, iframe.html file will be in charge of emitting events to the server and displaying messages when events are emitted from the server.

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8">

<title>Memcached</title>

</head>

<body>

<h1>This iframe is on port #<span id="port-number"></span></h1>

<div id="messages"></div>

<hr>

<button id="broadcast">Broadcast</button>

<script src="/socket.io/socket.io.js"></script>

<script>

// The port number needs to be dynamic so we can

// Use this page on any port

var port = window.location.port,

socket = io.connect('http://localhost:' + port);

document.getElementById('port-number').innerHTML = port;

// Add new messages to the list

socket.on('message.received', function (port) {

var message = document.createElement('div');

message.innerHTML = `Received message from port ${port}`;

document.getElementById('messages').appendChild(message);

});

// When the "broadcast" button is clicked,

// We will send a message to the server to render the message

document.getElementById('broadcast').addEventListener('click', function () {

socket.emit('message.sent', port);

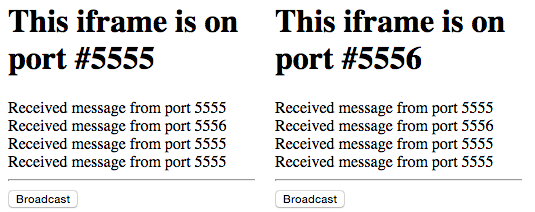
});

</script>

</body>

</html>

1. Start your server by running node server. You can navigated to localhost:5555 in your browser and click the “broadcast” button in either iframe to broadcast a message in both domains. The result should look something like this:



Insert Image B04893\_06\_02.png

## How It Works...

Since RabbitMQ does such a great job of consuming and broadcasting events, this whole process is actually really easy. In our application, Socket.IO will emit a message from the client to the server. The server will take the emitted event and emit it over RabbitMQ. Every server is listening for RabbitMQ to broadcast a socket.io message, so when it does, every server consumes the message and broadcasts it using the default Socket.IO emit() method.