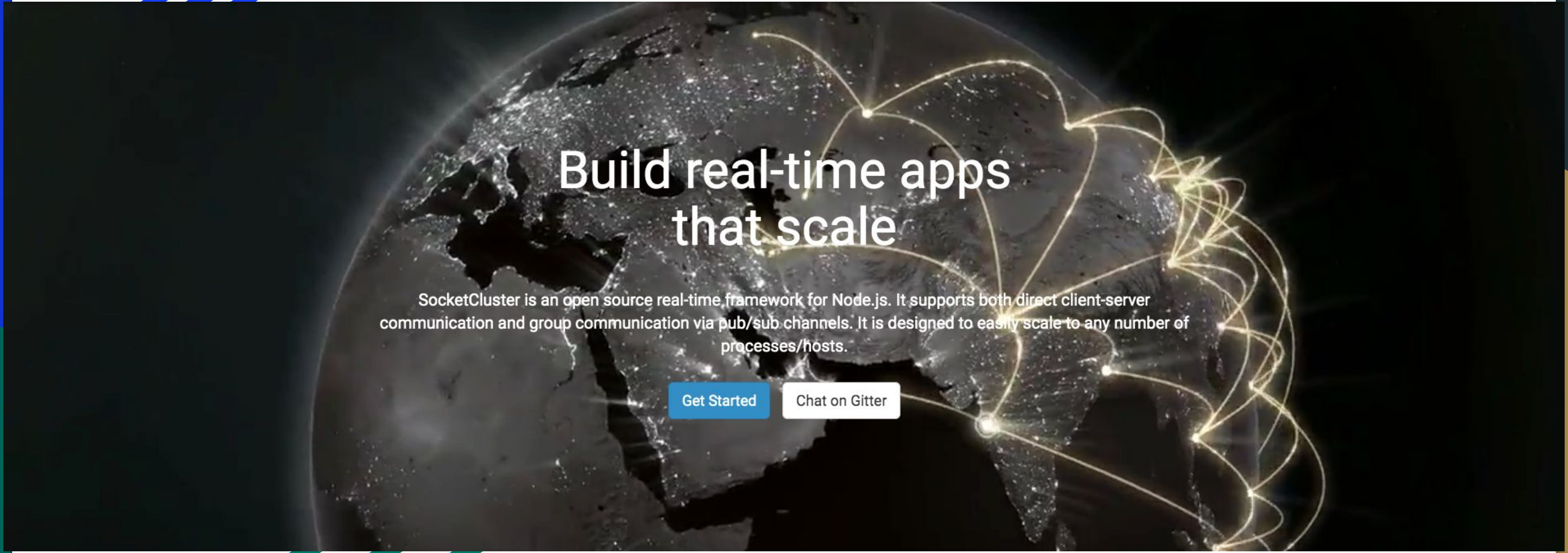
Socket Cluster



Build real-time apps that scale

SocketCluster is an open source real-time framework for Node.js. It supports both direct client-server communication and group communication via pub/sub channels. It is designed to easily scale to any number of processes/hosts.

Introduction

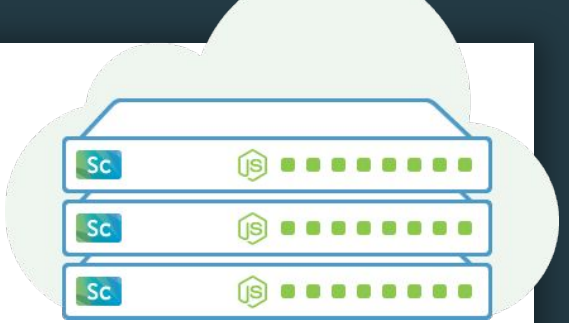
SocketCluster is a fast, highly scalable HTTP + WebSockets server environment which lets you build multi-process real-time systems that make use of all CPU cores on a machine/instance. It removes the limitations of having to run your Node server as a single process and makes your backend resilient by automatically recovering from worker crashes.

SocketCluster works like a pub/sub system (which extends all the way to the browser/IoT device) - It only delivers particular events to clients who actually need them.

You can use SocketCluster as a standalone framework (to act as both a HTTP and WebSocket server) or use it only as a real-time engine and serve the client script separately.

- SC is ideal for building highly flexible, resilient and scalable chat systems.
- SC is ideal for building multi-player online games.

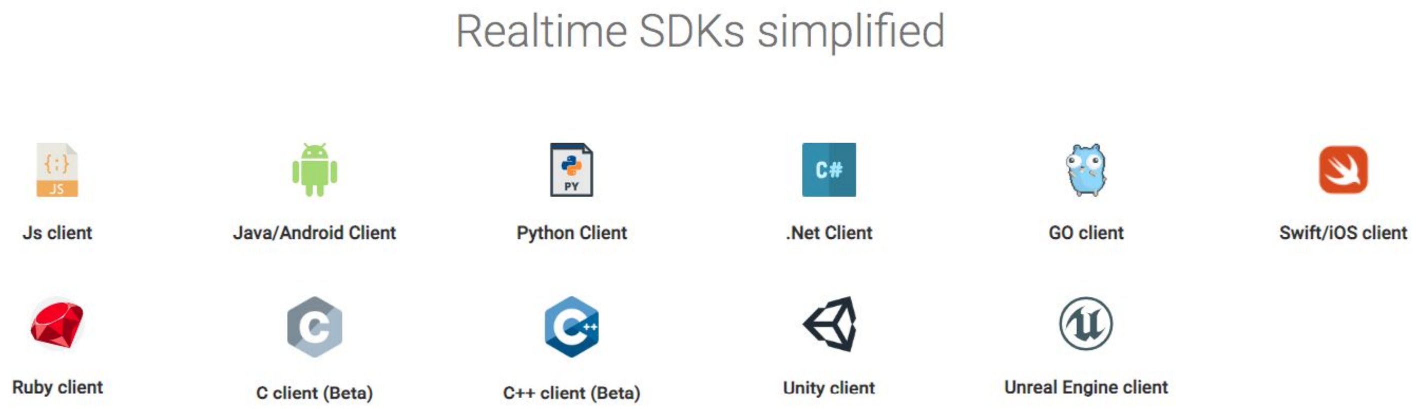
Technical Features



1. Scales linearly as you add more CPU cores and workers.
2. Scales horizontally across multiple machines.
3. Resilient on both the client and backend - Process crashes, lost connections and other failures are handled seamlessly.
4. Supports custom codecs for compressing messages during transmission.
5. Supports both pub/sub channels and direct client-server communication (for RPC).
6. Authentication engine compliant with JSON Web Token (JWT).
7. Authorization via middleware functions (access control for emit, publish in, publish out, subscribe and handshake).
8. Client sockets automatically reconnect (by default) if they lose the connection.
9. SocketCluster is a pure WebSocket (no polling hack) solution.
10. Designed to work alongside any database/datastore.
11. Open source alternative to pub/sub 'as-a-service'.
12. Optimized to run and autoscale on Kubernetes.

What is it good for?

- Single-page apps which need to render live data.
- Finance, cryptocurrencies and other blockchain applications.
- Chatbots and other chat-related applications.
- IoT devices.
- Mobile apps built with web technologies such as React Native or Ionic/Cordova.
- Multiplayer Online Games.
- Any kind of real-time app or service which needs to scale to millions of users.



Realtime SDKs simplified

Js client
Java/Android Client
Python Client
.Net Client
GO client
Swift/iOS client
Ruby client
C client (Beta)
C++ client (Beta)
Unity client
Unreal Engine client

Getting started

To get started with SocketCluster, you need to have Node.js installed.

Once you have Node installed, you can install SocketCluster.
There are two ways to install SocketCluster -

1. You can install the client and server separately (this may be better if you have more specific requirements).
   To install SocketCluster as a standalone server and client, follow the instructions from these 2 repositories.
   https://github.com/SocketCluster/socketcluster-server
   https://github.com/SocketCluster/socketcluster-client

2. You can install it as a framework (this is the simplest way)
   To install it as a framework (recommended):

```
npm install -g socketcluster
```

Once installed, the socketcluster create command will create a fresh SocketCluster installation. For example, socketcluster create myApp will create a directory inside your current working directory called myApp

```
socketcluster create myApp
```

Serving SocketCluster

- When this is done, you can navigate to `myApp` and run your server immediately using `node server.js`

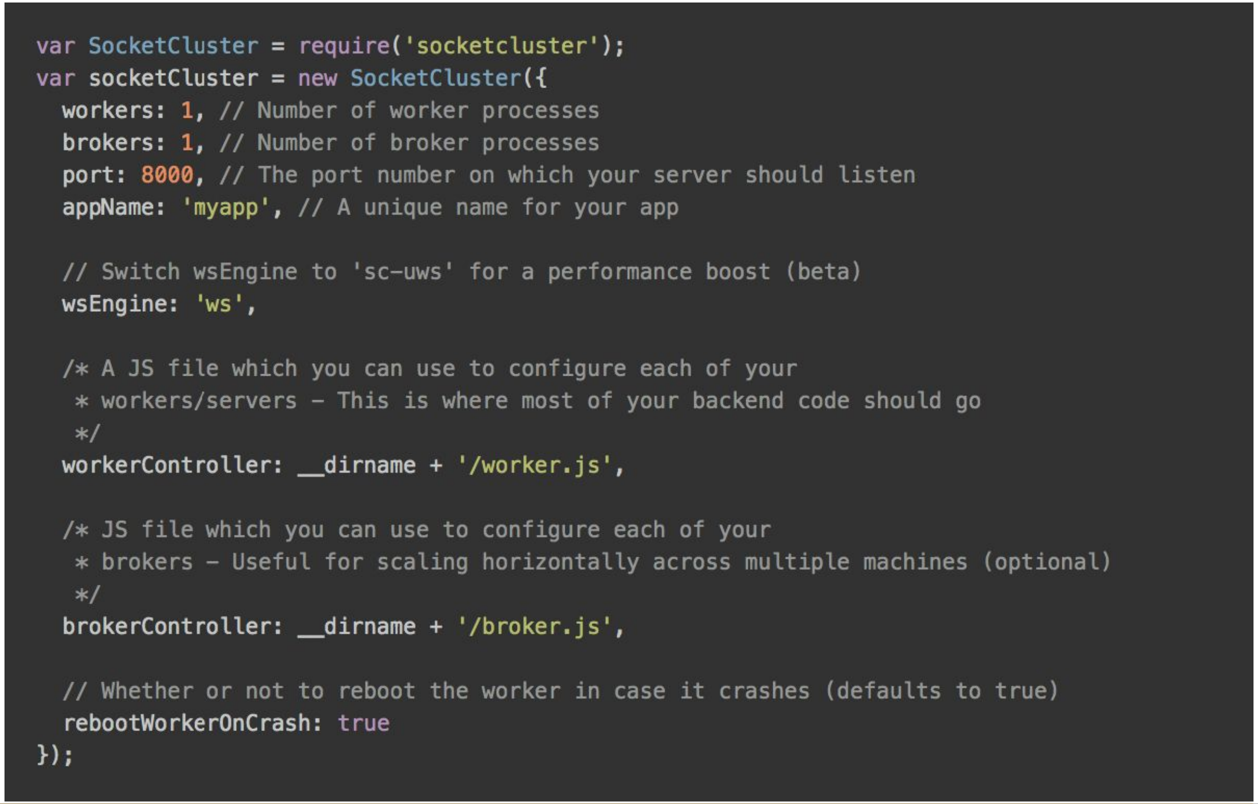
  ```
  node server.js
  ```

- You can connect to your server by navigating to http://localhost:8000/ in your browser.

  To test SocketCluster's real-time features, you can open your browser's developer console and enter this:

  ```
  // Client side
  // Use socketCluster.connect() if socketcluster-client < v10.0.0
  var socket = socketCluster.create();
  socket.emit('sampleClientEvent', {message: 'This is an object with a message property'});
  ```

Here is a sample (basic) `server.js` file (note that the default one which comes with the framework might be more detailed - See here).

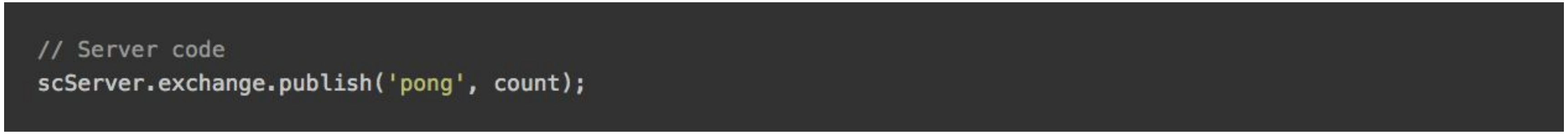


```
var SocketCluster = require('socketcluster');
var socketCluster = new SocketCluster({
  workers: 1, // Number of worker processes
  brokers: 1, // Number of broker processes
  port: 8000, // The port number on which your server should listen
  appName: 'myapp', // A unique name for your app

  // Switch wsEngine to 'sc-uws' for a performance boost (beta)
  wsEngine: 'ws',

  /* A JS file which you can use to configure each of your
   * workers/servers - This is where most of your backend code should go
   */
  workerController: __dirname + '/worker.js',

  /* JS file which you can use to configure each of your
   * brokers - Useful for scaling horizontally across multiple machines (optional)
   */
  brokerController: __dirname + '/broker.js',

  // Whether or not to reboot the worker in case it crashes (defaults to true)
  rebootWorkerOnCrash: true
});
```

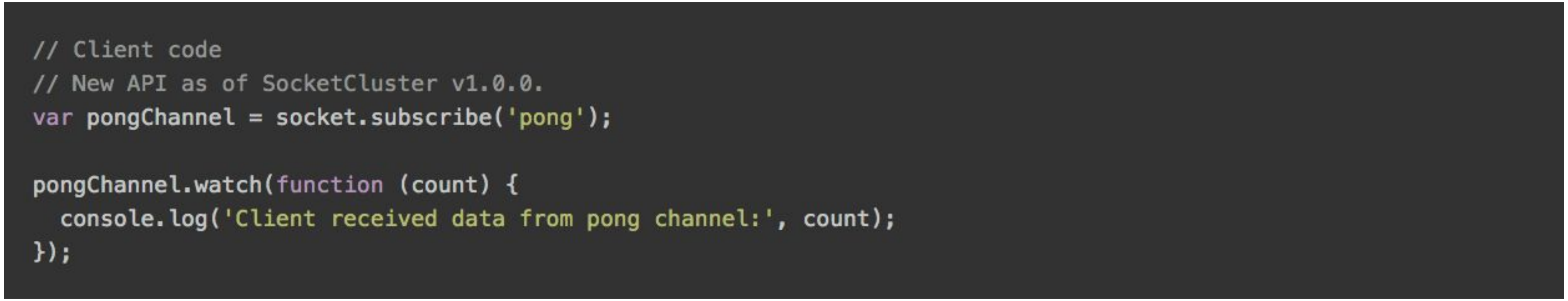
```
// Server code
app.use(serveStatic(__dirname + '/public'));

httpServer.on('req', app);
```

```
// Server code
var count = 0;

scServer.on('connection', function (socket) {
  // ...

  socket.on('ping', function (data) {
    count++;
    console.log('PING', data);
    scServer.exchange.publish('pong', count);
  });
});
```

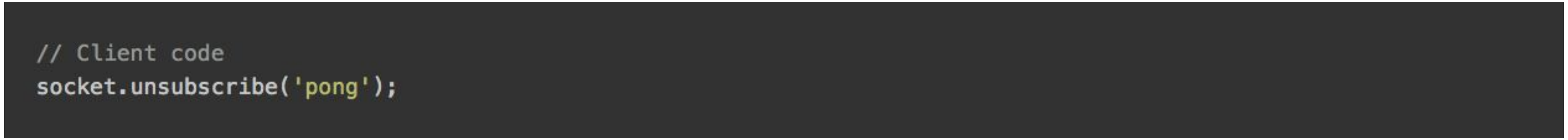
```
// Client code
socket.emit('ping', 'This is a PING message')
```



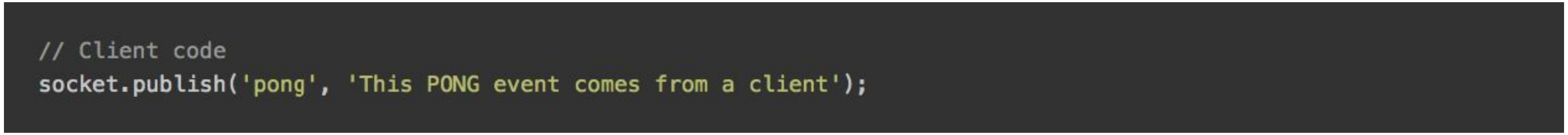
```
// Server code
scServer.exchange.publish('pong', count);
```



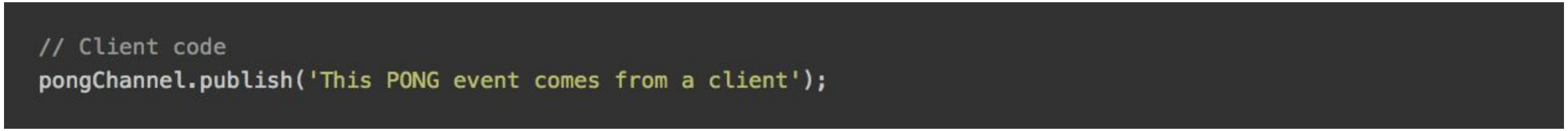
```
// Client code
// New API as of SocketCluster v1.0.0.
var pongChannel = socket.subscribe('pong');

pongChannel.watch(function (count) {
  console.log('Client received data from pong channel:', count);
});
```



```
// Client code
socket.unsubscribe('pong');
```



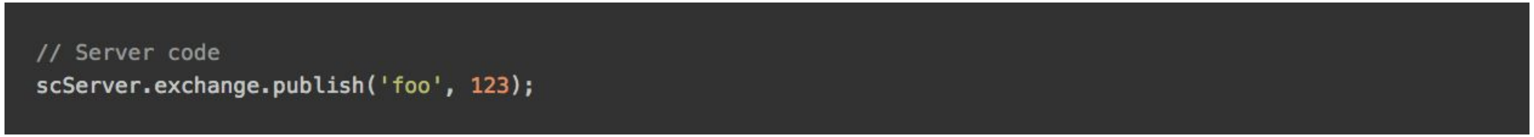
```
// Client code
socket.publish('pong', 'This PONG event comes from a client');
```



```
// Client code
pongChannel.publish('This PONG event comes from a client');
```

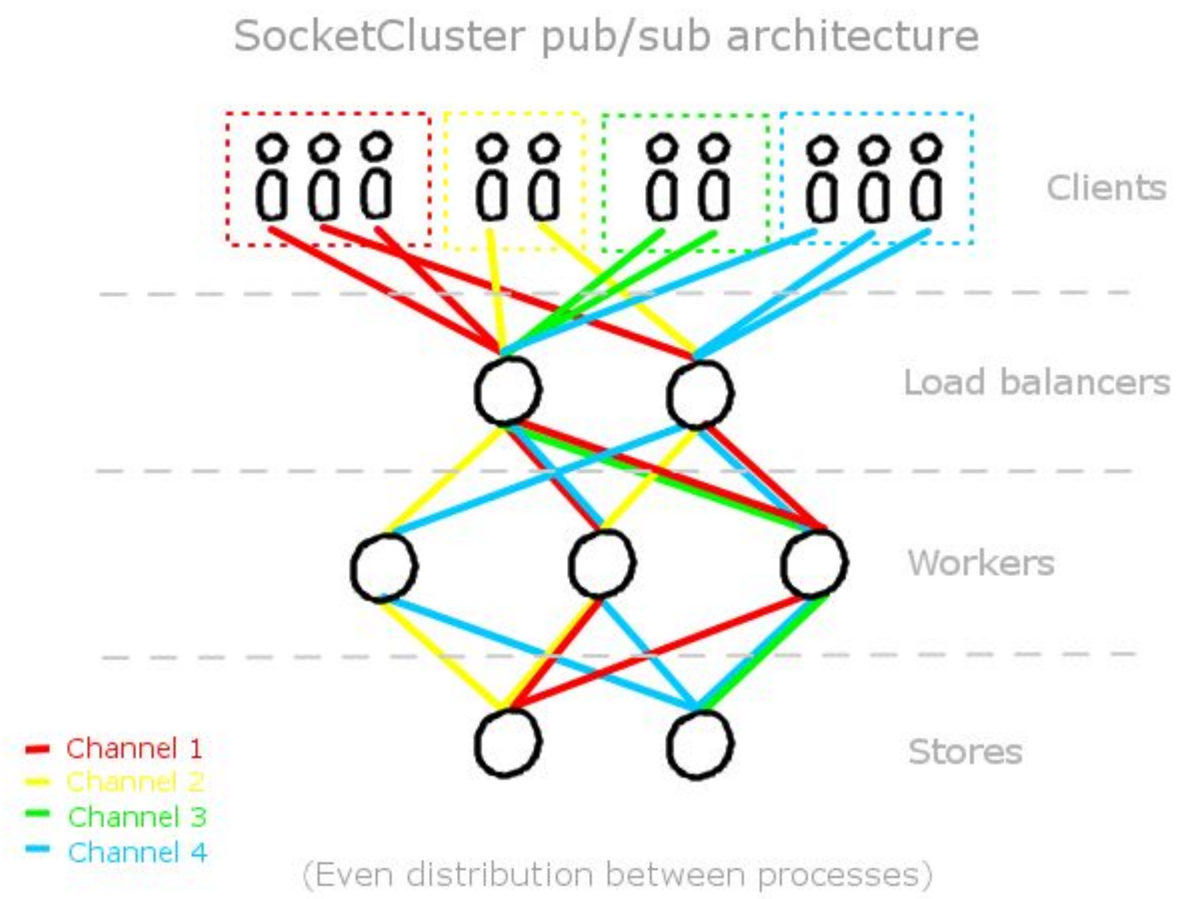
Summary

- The socket.emit(event, data) function allows you to send messages between 1 client socket and 1 matching server socket (1 client socket ⇄ 1 server socket — One to one communication between client and server).
- The socket.publish(event, data) and channel.publish(data) functions allow you to send group messages between multiple client sockets (n client sockets ⇄ n client sockets - Many to many communication directly between clients). As shown earlier, you can also call publish from the server using the exchange object:



```
// Server code
scServer.exchange.publish('foo', 123);
```

You can also use publish for one to one communication between two clients but you should setup some middleware to make sure that only the two authorized clients can share the same channel


SocketCluster pub/sub architecture
Clients
Load balancers
Workers
Stores
Channel 1
Channel 2
Channel 3
Channel 4
(Even distribution between processes)

Component of SocketCluster

The component of SocketCluster, which runs a SocketCluster server, which is generated in the server process.

1. The main process (Server.js) where everything starts, you can set the parameters, and will call Workers and Brokers

2.Workers: In the workerController you can set the HTTP server logic, as well as manage SocketCluster real-time connections and events (sending broadcasts, etc.)

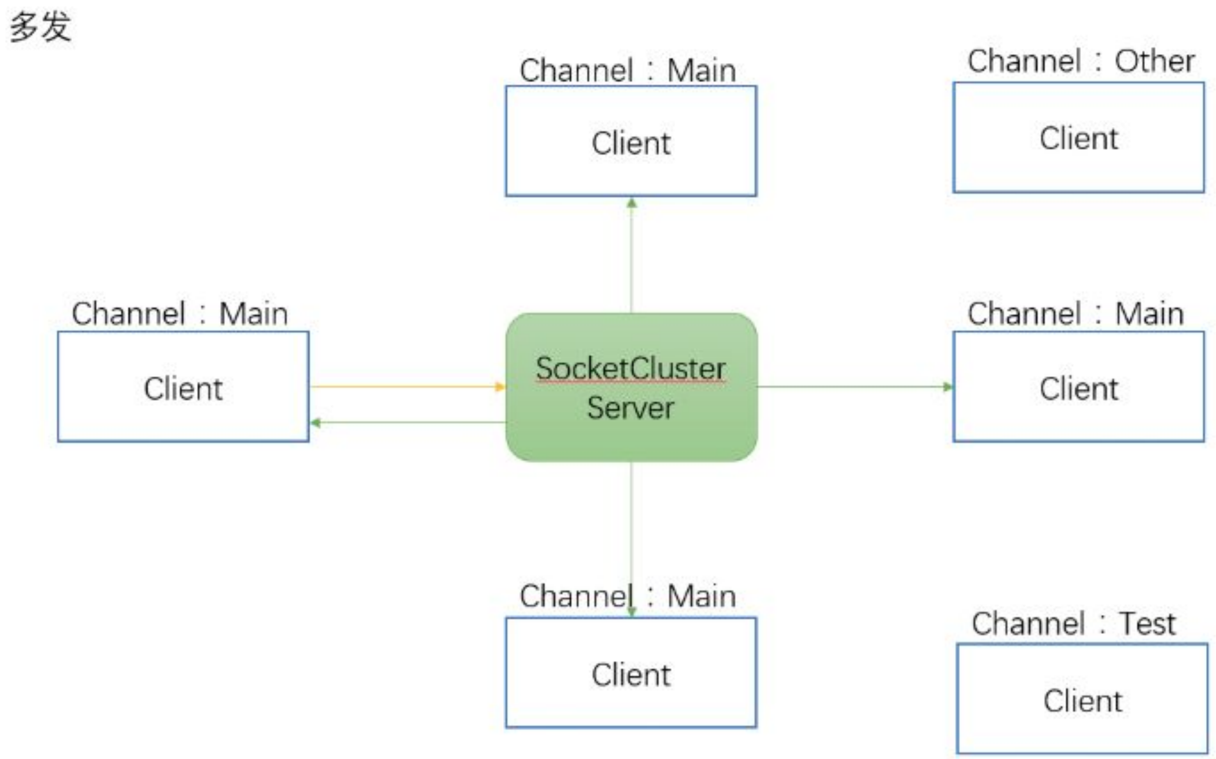
3.Brokers: Mainly used inside SocketCluster, allowing efficient sharing of channel data between different workers, also using its session data and horizontally expanding nodes among multiple servers

Method of sending a message

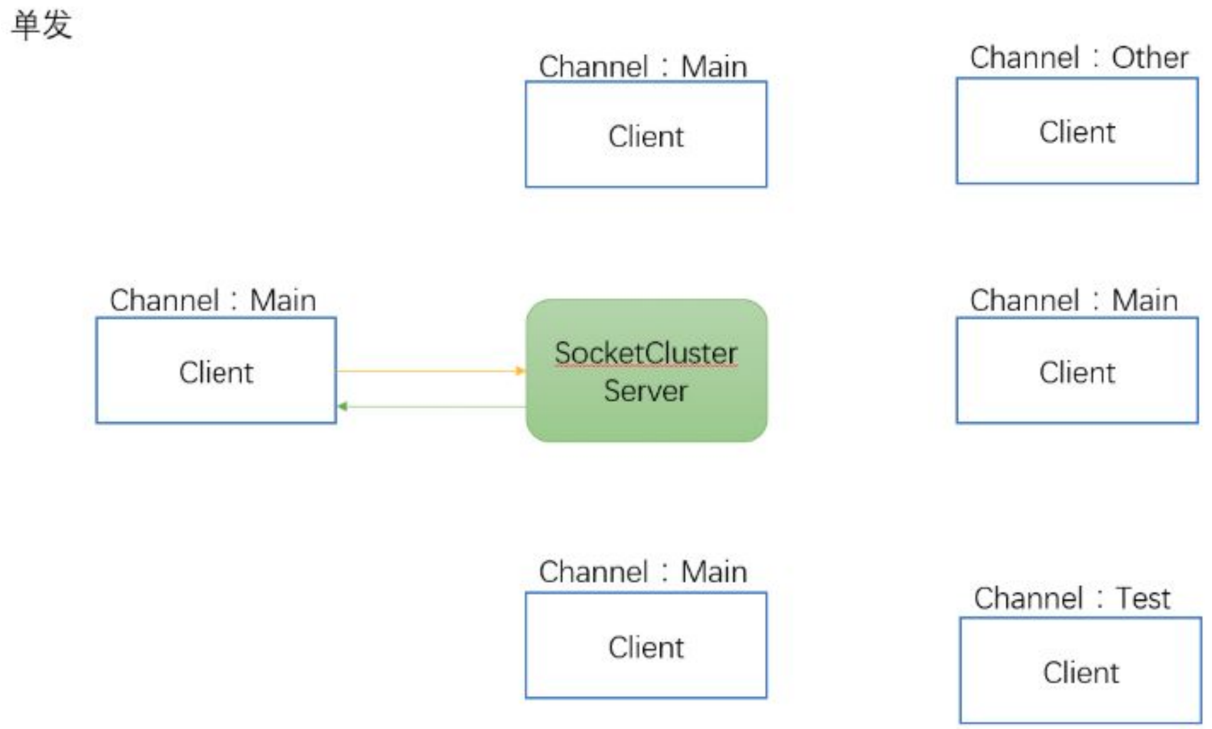
- Publish: socket.publish and channel.publish allow a set of messages to be sent to different clients, ie (n client sockets ⇄ n client sockets - Many to many communication directly between clients)

- Emit: allows messages to be sent on one client and one Socket server (1 client socket ⇄ 1 server socket — One to one communication between client and server)

N client sockets ⇄ N client sockets
Many to many communication directly between clients



1 client socket ⇄ 1 server socket
One to one communication between client and server



THANK YOU.