Article
Sean Connelly · May 8, 2018 5m read

Roll your own Node.JS to Caché adaptor

If your looking to develop a Node.JS to Caché library then you might want to consider using a pure TCP connection with a custom message transport protocol. This bypasses the native Caché connector libraries that can get stale with a new release.

Node.JS is very good at non blocking code development, so building a performant solution isn't that complex.

The idea behind this example is that we have a node HTTP server that maps HTTP requests to a Caché TCP connection. If we are implementing JSON-RPC as a message transport protocol then we would write a second layer inside Caché that maps these requests to concrete methods and returns the response as JSON.

To start with we need an HTTP server, the server will take an HTTP request and push its payload onto an internal queue, essentially just a JavaScript array. When a new message is put on the queue it uses the Node.JS event emitter to broadcast that a new message has arrived.

```
let httpServer = function(httpRequest,httpResponse) {

    let payload = [];

    httpRequest.on('data',function(data){
        payload.push(data);
    });

    httpRequest.on('end',function(){

        if (httpRequest.method !== 'POST') {
            payload.push(JSON.stringify({
                "method" : httpRequest.method,
                "url" : httpRequest.url
            }))
        }

        queue.push({
            res : httpResponse,
            payload : payload.join()
        });
        payload = [];
        eventEmitter.emit('check-queue');

    });

};

let server = http.createServer(httpServer);

server.listen("80", function() {
    console.log('HTTP server listening on port 80');
});
```

Next we need a TCP server. Node.JS will create a TCP listener and Caché will connect into it.

The idea is to allow HTTP requests to stream into Node.JS whilst messages are dealt with one at a time at the pace that Caché can process them. This can be scaled out by adding more Node.JS processes and attaching a unique Caché process to each Node.JS process.

Here we create a TCP server. The server will subscribe to the HTTP queue and will pop requests when it has an opportunity to process the next message on the queue. It effectively manages a flag called ready, when its waiting for a TCP response it sets ready to false. When a TCP response comes back the TCP message response is written out to the original HTTP request response which is attached to the queue.

```
const tcpServer = net.createServer( (socket) => {

    let tail;
    let ready = true;
    let task = null;
    let header = true;

    const socketWriter = () => {
        if (socket.writable && ready && queue.length>0) {
            task = queue.shift();
            ready = false;
            socket.write(task.payload);
        }
    };

    eventEmitter.on("check-queue", socketWriter);

    const onConnectionProblem = () => {
        if (task !== null) {
            if (header) {
                task.res.writeHeader(200, {"Content-Type": "application/json"});
                header = false;
            }
            task.res.end(JSON.stringify({
                status : "error",
                error : "database connection closed"
            }));
            ready = true;
            task = null;
            header = true;
        }
        eventEmitter.removeListener('check-queue',socketWriter);
    };

    socket.on('data', (buffer) => {
        if (task===null) return;
        if (header) {
            task.res.writeHeader(200, {"Content-Type": "application/json"});
            header = false;
        }
        task.res.write(buffer.toString('utf8'));
        tail = buffer.toString('utf8',buffer.length-2,buffer.length);
        console.log("tail is EOL " + (tail===EOL));
        if (tail===EOL) {
            task.res.end();
            ready = true;
            task = null;
```

```
            header = true;
            eventEmitter.emit('check-queue');
        }
    });

    socket.on("error", (err) => {
        onConnectionProblem();
        console.log('ERROR: ' + err.toString());
    });

    socket.on('close', () => {
        onConnectionProblem();
        console.log('CLOSED');
    });

});

tcpServer.on('listening', function() {
    console.log('TCP server listening on port 55000");
});

tcpServer.on("connection",function() {
    console.log('TCP client connected');
    eventEmitter.emit('check-queue');
});

tcpServer.listen("55000", "127.0.0.1");
```

How you implement the Caché TCP client code really depends on the library that you are developing. In this code I have a JSON-RPC 2.0 that maps onto Caché classmethod's via compile time linking.

```
ClassMethod StartWorker(pIP As %String, pPort As %Integer, pRestartOnError As %Boolea
n = 0, pStopOnInterrupt As %Boolean = 0)
{
    set io=$IO
    try
    {
        set dev="|TCP|"_$JOB
        write !,"Creating client device: "_dev
        write !,"Waiting for server"
        set connected=0
        while 'connected {
            open dev:(pIP:pPort:"C"):5
            set connected=$TEST
            if connected=0 Write "."
        }
        write !,"Connected"
        do ..SetStatus(dev,"UP",pIP,pPort)
        use dev
        write !
        while (..GetStatus(dev)="UP")
        {
            do ..OnMessage(dev)
        }
    } catch err {
        do ..LogError(err)
        if $Get(dev)'="" do ..ClearDown(dev)
        use io
```

```
        write !,"Error: ",err.DisplayString(),", connection closed"
        if pStopOnInterrupt,err.DisplayString()["<INTERRUPT>" goto END
        if pRestartOnError do ..StartWorker(pIP,pPort,1)
    }
END
    use io
    quit
}

ClassMethod OnMessage(dev)
{
    set uuid=""
    set json=""
    read message:5
    if message="" write ! quit
    try {
        set rpc=##class(Cogs.CoffeeTable.Request).deserialize(message,"rpc")
        set class=$P(rpc.method,".",1,$l(rpc.method,".")-1)
        set obj=$classmethod(class,"call",rpc.method,rpc)
        write obj.toJSON(),!
    } catch err {
        do ..LogError(err.DisplayString())
        write "{""status"":""error"",""error"":"""_err.DisplayString()_"""}"
    }
}
```

Running some benchmarks, it turns out that processing JSON-RPC messages can be up to 2x quicker via a Node.JS connection than via a CSP connection. This is probably due to the cost of establishing a new CSP request and response object each time vs streaming data in and out via TCP.

These examples are experimental and have not actually been used in production. They would require a little more defensive code, but it does demonstrate that there are alternatives for consideration.

One interesting consideration of this approach is that Caché could be fully firewalled off and only allows messages into Caché from the TCP queue collection. Any weaknesses from the O/S up would be hard to exploit since there is no way to jump onto the box.

Sean.

#Caché #Node.js