ZOO-Project GSoC 2022 Proposal by @mmomtchev

Adding Node.js support for service implementation to be run from the ZOO-Kernel

Abstract

The ZOO-Project is a solid WPS server able to handle services implemented in various different programming languages. The existing `ZOO-Kernel` supports C, C++, and JS implementations with the SpiderMonkey engine. With this project, the objective is to add support for NodeJS implementation of the `ZOO-Kernel`.

Mentors: Gérald Fenoy, Aditi Sawant, Rajat Shinde

Contributor Personal Details

Momtchil Momtchev momtchil@momtchev.com, France +33611640937 https://github.com/mmomtchev https://twitter.com/mmomtchev https://mmomtchev.medium.com

Graduated from the Université des Sciences et des Technologies de Lille with a French DEA in Computer Science (Master's equivalent)

Currently unemployed and looking for job in open-source

First time applicant to GSoC

Co-author and current maintainer of the Node.js bindings for GDAL (https://github.com/mmomtchev/node-gdal-async)

Author of the React bindings for OpenLayers (https://github.com/mmomtchev/rlayers)

Author of the Node.js bindings for ExprTk (https://github.com/mmomtchev/exprtk.js)

Occasional Node.js and GDAL contributor

Author of numerous smaller packages and tools (https://www.npmjs.com/~mmomtchev)

Expert C/C++ and JS/TS engineer with Linux, macOS and Windows experience

Existing Software

Currently ZOO-Project supports JS services through the embedded version of the SpiderMonkey engine. It is linked as a shared library and every invocation of a service results in a separate instance of the SpiderMonkey engine.

Proposed Solution

The current latest LTS version of Node.js, Node.js 16.x, is to be embedded in the `ZOO-Kernel` executable following the same architecture as SpiderMonkey.

It is worth noting that the next LTS version of Node.js, Node.js 18 is scheduled to be released during the GSoC timeline. As this version is not expected to replace the 16.x as recommended version until October 2022, it is believed that Node.js 16.x remains the safer choice.

Node.js supports being built as a shared library since version 12.x. This feature is used by the Electron project which is its main maintainer. It allows the JS runtime to be loaded inside the address space of the calling program and to both call JS functions from the native code and expose native functions to the JS code.

There are several different interfaces available for interacting with the Node.js runtime: * By calling raw internal V8 and Node.js methods which are usually not stable across different versions * By using the NAN C++ API C++ which is stable at the source level across different versions * By using the Node C N-API which is stable at the binary level across different versions * By using the Node Addon C++ API which is stable at the binary level across different versions

Of these methods, the Node C N-API seems to be the best suited for ZOO-Project as it is binary stable across different Node.js versions and it does not require C++.

It should be noted that while these APIs/ABIs are usually meant to be used by native addons that are loaded by the main Node.js process as a shared library, they are also perfectly usable in the opposite direction - ie when Node.js is loaded as a shared library by a 3rd party process.

Node.js supports running multiple V8 isolates, each with a separate main thread, offering a completely separate execution environment to each JS instance.

These separate instances have separate event loops but can share the same worker thread pool.

This mode of embedding Node.js matches most closely the existing JS architecture based on SpiderMonkey. It is also the preferred mode for embedding Node.js.

The following alternatives have also been considered: * Run every instance of a service in a separate, external, Node.js process This mode will have a more expensive startup and will present unique challenges when implementing the various ZOO-Project routines which will have to communicate with the `ZOO-Kernel` by some external mechanism - such as RabbitMQ. * Run all instances of a service in a single shared environment This mode offers the potentially best performance but it imposes upon the end-user to use very correctly the Node.js asynchronous mechanisms - failing to do so will result in latency spikes and possibly dropped connections.

Rebuilding `libnode` as part of the `ZOO-Project` build system is considered to be out of the scope of the current project - just as the current SpiderMonkey shared library is expected to be provided by the end-user, so will be `libnode` which is already carried by some major Linux distributions - Ubuntu being one of them.

Testing methodology

`ZOO-Project` already has an existing testing framework which includes the SpiderMonkey services. The new Node.js implementation is to be able to run those services passing the existing SpiderMonkey tests without modifying the services code.

Additionally, an `AddressSanitizer` build, currently absent from `ZOO-Project` is one of the stretch goals of the proposal.

Proposal Timeline

I am fully committed to working on this project during this time period and I am free of any other professional obligations.

Before June 13

- To familiarize myself completely with ZOO-Project functionality and architecture.
- To experiment with using `libnode`, the embedded version of NodeJS and test the compatibility between the packaged `libnode` by the Linux distributions and the existing Node.js native addons

June 13 - July 25

June 13 - June 17

- Implement the creation of the `libnode` context in `service_internal_nodejs.c` to be called from `zoo_service_loader.c:loadServiceAndRun()`
- Include `libnode` in the `ZOO-Kernel` build and link against it in the official Dockerfile
- Use a dummy static JS method

June 20 - June 24

- Test building with the versions included Ubuntu, Debian, CentOS and Fedora

June 27 - July 1

- Implement `ZOORequest`, `ZOOTranslate`, `ZOOUpdateStatus` and `alert` using Node N-API

July 4 - July 8

- Implement the object and the array transforms using Node N-API

July 11 - July 15

- Load the existing 3rd party code - mostly `proj4js` into the environment

July 18 - July 22

- Spare week for testing and debugging

July 25 - July 29

- Phase 1 evaluation: the existing SpiderMonkey unit tests should be passing on Node.js at this point

July 29 - September 4

Stretch goals

August 1 - August 5

- Add builtin Node.js GDAL support
- Allow JS services to use `gdal-async` out of the box
- Reimplement the GDAL profile C++ service in JS as an example

August 8 - August 12

- Create and automate an `AddressSanitizer` build to be run in continuous integration

August 15 - August 19

- Allow services to request to be executed in a single-instance mode by providing an `async` function as entry point

August 22 - August 26

- Add snapshot support to `libnode` - it could benefit from having the same V8 snapshot support as the Node.js main executable. V8 supports creating and restoring snapshots of a JS heap - this feature is used by Node.js to speed-up the initial loading of the JS class library - instead of compiling it at every process startup, it is compiled and initialized once during the build of the Node.js executable and then the JS heap is saved in a snapshot to be reused when launching Node.js. Currently, this feature is not readily available when using `libnode`. Electron re-implements it on its own. This is to be submitted for merging back in Node.js.

August 29 - September 2

- Spare week for testing and debugging

After September 4

- Phase 2 evaluation