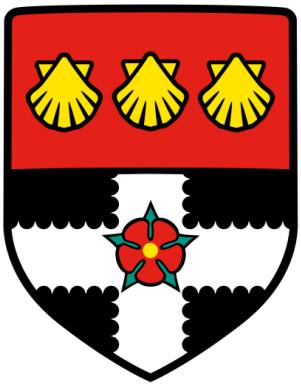
University of Reading
Department of Computer Science



Nihilo: an versatile embedded runtime

Patrick Hingley

*Supervisor:*Atta Badii

A report submitted in partial fulfilment of the requirements of the University of Reading for the degree of Bachelor of Science in Computer Science

May 11, 2020

Declaration

I, Patrick Hingley, of the Department of Computer Science, University of Reading, confirm that all the sentences, figures, tables, equations, code snippets, artworks, and illustrations in this report are original and have not been taken from any other person's work, except where the works of others have been explicitly acknowledged, quoted, and referenced. I understand that if failing to do so will be considered a case of plagiarism. Plagiarism is a form of academic misconduct and will be penalised accordingly.

Patrick Hingley May 11, 2020

1 Abstract

The Internet of Things (IoT) is a large and growing industry connecting home devices to the internet for remote control and monitoring, based on embedded computers. Nihilo (pronounced nie-low) is a runtime designed to simplify embedded development, while also making it more secure. Nihilo supports encrypted remote procedure calls for communication, JSON files for storage, and webassembly as a portable executable format, on a $3 microcontroller with half a megabyte of RAM. A variety of languages can already target Nihilo devices, including C and C++. Although its current form has limits, the main goal of this project was to prove the concept's validity, which it does.

2 Acknowledgements

I would like to thank my tutor, Atta Badii, for guiding me away from pure theory and towards practicality.

I would like to thank my girlfriend for putting up with my increasingly antisocial sleeping pattern as the deadline got closer.

I would like to thank the entire open source community for creating nearly every tool I used to create this project. Through their example, they showed me what to do and in some equally important cases, what not to do.

Keywords: webassembly, embedded computing, Internet of Things, elliptic curves

Word Count: 11342

Contents

List of Figures

List of Tables

3 Introduction

Over the past 50 years, UNIX and its descendent Operating Systems have created a set of standardised patterns for how computers interact with themselves, each other, and the outside world. These patterns; files, threads, device drivers, etc. are so powerful that computing devices are classified first by whether or not they are capable of supporting them. Microprocessors typically support multitasking, networking, and are connected to large amounts of volatile and non-volatile storage. Microcontrollers, on the other hand, tend to be realtime, with primitive networking and EEPROM storage that often cannot be rewritten at runtime.

As Moore's law slows and the graph of computing power over time begins to flatten, semiconductor manufacturers have created new devices. Because waiting for cheaper hardware is no longer a valid 'optimization' strategy, manufacturers have begun adding microprocessor-like capabilities to microcontrollers. Although these new devices are traditionally classified as microcontrollers, their capabili-

ties blur the line between the microcontrollers and microprocessors. The 'super-microcontroller' used in this project was the Espressif systems ESP32.

The goal of this project, named Nihilo, is to create a runtime with a new set of patterns designed from the ground up for the new hardware; to encapsulate storage, code execution and communication in a simple to use and extensible platform. Because implementing an entire platform of this scale is outside of the scope of a single-module dissertation, this project is more a proof-of-concept than a final version. Concessions have been made to simplicity to allow for experimentation, meaning that only limited guarantees can be made about security or speed. The project aims, however, to be well tested and stable.

There are a great number of potential applications of this project, anywhere where microcontrollers must communicate with each other. For example, the IoT (Internet of Things) industry has become notorious for hiring programmers without a security background, who then include trivially exploitable vulnerabilities in code and configuration files[1]. IoT, a concept which overlaps with smart home, is the idea of taking electronic home devices, like toasters, fridges, and thermostats, and connecting them to the internet for remote management. By taking the most difficult and error prone parts of designing for IoT, and hiding them behind an abstraction layer which removes the opportunity for dangerous creativity, greater security can be achieved while also reducing developer workload.

4 Literature Review

4.1 Similar Projects

4.1.1 MQTT

MQTT (Message Queuing Telemetry Transport) is a versatile TCP (Transmission Control Protocol) based application layer protocol[2], designed for compatibility with small network-capable devices including microcontrollers. MQTT is built on brokers and clients. Brokers are similar to servers in a standard client-server architecture, with the difference that they can wait very long periods of time after a subscription to send the client a message. In order to express interest in a certain type of message, a client subscribes to it by sending a request to a broker. Whenever the broker receives a message of that type, it informs the subscribed client. Multiple subscribers can receive messages from a single publisher, and multiple publishers can send messages to a single subscriber.

The split between small, low-footprint clients, which might not even have persistent data storage and large, fixed brokers enables MQTT to exploit the advan-

tages of both. The clients do not need to directly communicate with each other so they only need to switch on networking periodically, and most of the complicated message and error handling logic can be contained within the broker. This method of messaging is inherently asynchronous - replies have not been built into the protocol, and if desired must be implemented by the user. The retain flag can make MQTT even more asynchronous. In this mode, the broker can hold on to a message for a period of time, so that it is sent the next time the broker is able to communicate with a subscriber to that message type.

MQTT has widespread adoption mainly in IoT, but its almost unique abilities for asynchronous messaging over periods of hours or days gives it niche uses away from fast, reliable network infrastructure. It has applications in dial-up where messages are bundled and sent all at once over a phone line for cost reasons, and satellite communication where messages can only be sent when there is line of sight to the satellite.

While MQTT is useful in many applications, it does have downsides. From the beginning it was built around a desire for maximum compatibility, and so there are many useful features which are not included. The first of these is encryption, and while it can be implemented by the user, both approaches to doing so have their own problems:

- MQTT can be directed through TLS (Transport Layer Security) before running through TCP. Although TLS is very secure, it has a large overhead due to being RSA-based. TLS is not end-to-end encryption, so the server will see the message's content (the payload).
- The payload can be encrypted using a user-designed encryption scheme. While this would be end-to-end encrypted, the message type and all other information other than the payload would again be visible to an attacker. Non-experts are also generally discouraged from designing encryption schemes, because guaranteeing security is difficult.

The most serious problem with either of these solutions, however, is that they are non-standard and therefore products are likely to be developed which do not use them. Good encryption schemes should be pervasive and non-optional. The reliance on a central broker is another drawback of this protocol, and becomes unnecessary when developing for a microcontroller as powerful as the ESP32.

4.1.2 CouchDB

Couch DB is a document-oriented database. This means that instead of storing rigid rows and columns of data as in an SQL (Structured Query Language)

database it stores data as files, JSON files in this case. JSON (JavaScript Object Notation) is a format that hierarchically structures data in the form of numbers, strings and booleans into objects and arrays. JSON was originally developed as a serialization format for messages going between websites and servers, but has found use in storing configuration files, metadata, and in this case databases.

The ability to store hierarchical data earns CouchDB the designation of NoSQL (Non SQL). Because SQL essentially represents rows as tuples, fixed length arrays where the datatype of each index is pre-assigned, it cannot handle variable-length data within a single row. In order to circumvent this problem, SQL makes heavy use of row IDs to link records together, which must be queried using complex SQL which can take significant development time. While CouchDB does still support links between documents, there is less need for them because they tend to represent data in a way which is already closer to how it is structured in code.

CouchDB has an API which is exposed over HTTP on port 5984[3]. The various HTTP verbs, like GET, PUT, DELETE and POST are exposed as normal, but with special meanings in the context of a document-oriented database. For example, the verb PUT allows a user to create a new database if run against http://IP_ADDRESS:5984/DATABASE_NAME. Databases can be deleted, new documents inserted, and metadata retrieved in a similar fashion. This method, of using HTTP to expose capabilities over a network is useful. It allows the developer of the database to use pre-existing and well engineered tools like curl for development. It is also likely that any prospective user will already have a good conceptual model of the protocol involved.

CouchDB also has a built-in javascript engine, used to program queries. Instead of sending raw database values to the user, as is usually the case in SQL, a derived value can be calculated, and sent to the user instead. While something like this is possible using SQL views, in practise developers usually forgo it in favour of calculating the derived value in a more familiar language on the application server. This shows the advantage of running code in proximity to the values being queried.

Although CouchDB is an interesting approach to the design requirements at the intersection of JSON and high-performance databases, there are aspects of its design which are not relevant to the ESP32. The microcontroller has extremely limited flash memory, so storing many JSON documents together in a highly optimized file format is not relevant. Any large data storage requirements would likely be fulfilled by external devices connected over a network. Should Nihilo ever expand beyond microcontrollers, however, CouchDB would be a good

example to imitate.

4.1.3 IoT-DSA

IoT-DSA (Distributed Services Architecture) is a relatively immature set of protocols and libraries, designed to transform sets of devices into a single continuous system. The DSA website goes as far as claiming that this process will one day result in "a fully connected world in which all objects are interoperable with one another regardless of manufacturer, specific device characteristics or functionality" [5]. However realistic the vision of bringing competing manufacturers with diverging material interests each other into alignment is, the DSA community have created a unique project as a consequence of that vision.

DSA's Link-Broker architecture is topologically similar to MQTT's Client-Broker architecture. It is based on HTTP and WebSockets, rather than TCP, and allows more fine-grained control over how a message is received. DSA allows a client (called a DSLink) to publish a node, which is a variable that has a single value at any point in time. Like MQTT, other links can subscribe to the node and be updated when its value changes. In MQTT, clients have no control over messages after they have been sent but in DSA, the link has more control over the node and can see if anyone has subscribed to it. DSA also allows for the persistent storage of these values, as a JSON file. This method of data persistence, where the values are saved to a single JSON file on disk, is much closer to how small, embedded devices need to store data than CouchDB.

Another point of difference between MQTT and DSA is that DSA has a greater focus on communication between brokers. Normally, brokers behave in the same way as they do in MQTT, forwarding nodes from one link to another, but nodes can also be forwarded from broker to broker to reach any link in the network. The use case of this, however, is unclear as any internet connected device should be visible to any broker, without needing forwarding. Either a client can directly contact a broker, or there is no path from the client to that broker. The only use of this functionality would be to accelerate the Domain-Specific Query Language, which allows a client to define a query to run on a link (as in CouchDB), and then distribute it over the various brokers to be run more quickly.

Although DSA has many interesting concepts, it suffers most from a lack of clear design parameters and a desire to make everything capable of doing anything. DSA has all of the same drawbacks as MQTT, such as lack of encryption and dependence on brokers, but for different reasons. They exist in MQTT because of a lack of physical resources on the device, but they exist in DSA because of a lack of systems thinking in the high level design of the platform. The desire

to create a unified, simple protocol which can communicate with web browsers as it can with microcontrollers leads to a solution that does neither well. The desire to homogenize the brokers together into a single system leads to scaling issues when many devices are each running queries on many nodes.

Nonetheless, DSA does work in low-demand environments like as in an IoT home automation hub, which is its primary purpose.

4.1.4 Node.js

Until the late 2000s, the division of work between the various languages of a website was rigidly defined. HTML, CSS and javascript interact with the browser to create the client-side behaviour of the webpage. Of these, the only language capable of general-purpose computation is javascript. On the server-side, languages such as PHP specify the interaction between the web browser and the resources of the website such as files and databases. Although Node.JS has changed architecture of the server side somewhat, the basic architecture is unchanged.

In the years prior to the release of Node.js, an arms race between the major browser manufacturers known as the browser wars [4] were occurring. Web browsers are designed to be interchangeable, so competing by adding features was of limited value. This left competing by improving performance, and one of the biggest targets for this optimization was the javascript engine. What was originally intended as a minor, non-performance critical language suddenly became of interest to large tech organisations like Microsoft, Mozilla and, in the year before the release of Node.js, Google. Because Google's Chrome browser was partly open source, Chrome's javascript engine (V8) was available as an independent, embeddable component that could be used in any project, and on any operating system that chrome targeted. As will be discussed in the WebAssembly section, this engine is closer to a compiler than an interpreter, translating the javascript into optimized machine code at runtime.

In contrast to the rate at which the client side was advancing, the server side was stagnating. Most web servers at the time functioned synchronously, each individual thread receiving a connection, then handling only that connection until the action has been completed. Because processors move much faster than a network round-trip, much of the time was spent waiting (called blocking) for a reply to be received. Multithreading did somewhat mitigate this, but introduced its own problems with thread switching overheads.

The creator of Node.js, Ryan Dahl, created a new system. It was based on two components, a task queue, which held work to be done and a javascript

engine (V8), which did the work from the queue. The reason why this approach can be faster is because instead of blocking until a reply has been received, it makes a note of what function to queue when the reply is received and moves on to the next task. This design pattern is known as asynchronous I/O[6]. The reason why Node.js includes javascript on the server side is because many web developers were more comfortable with javascript, being highly optimized and developed, than server side languages like PHP.

In Node.js there is much to emulate, asynchronous I/O is an efficient way to handle communication, and the use of a portable language already familiar to target users made it easy to switch to. It can, however, suffer from its own version of spaghetti code in which a gap between an asynchronous operation and the functions called when it completes make code longer and hard to read.

5 Methodology

5.1 High-level Requirements

These requirements are defined in terms of two key words, **must** and **should**. **Must** is used for qualitative requirements, which are either met or not. **Should** is used for quantitative requirements, whose criteria will be specified prior to testing.

A good solution to this project will:

- Compile code on a computer, and then execute it on an ESP32. This code:
    1. **Must** support mature languages, compilers and tooling
    2. **Should** execute tasks quickly.
- Save data to their device. This capability:
    1. **Must** be able to save strings while the device is off
    2. **Must** be able to structure the saved data
    3. **Should** write data quickly
    4. **Should** read data quickly
    5. **Should** retain data accurately
- Communicate between devices. This communication:
    1. **Must** be reliable, ensuring messages are received and causing an error if they are not.

2. **Must** allow the user to call a function on the target device, using a string as a parameter and as the return value
3. **Must** be secure, the content of a message should not be visible to an attacker
4. **Must** be stable over long periods of time
5. **Should** be low-latency, sending a message and receiving a reply quickly

- Adhere to the general principle that beyond these basic requirements, any eventual implementation will be dependent on factors not yet known at the time the requirements are specified. The project should therefore be designed in a manner that is conducive to experimentation and learning about the nature of the problem.

5.2 Technologies used

5.2.1 ESP32

Released in 2016, the Espressif Systems ESP32 is a powerful dual-core microcontroller. It is based a Cadence IP core, which is in turn based on the Xtensa instruction set [7]. This instruction set is often used for specialist Digital Signal Processing hardware because it supports SIMD and customised instructions [8]. Although it is RISC, it generally complies programs into fewer instructions than arm because it allows for more flexible memory manipulation.

	Arduino Uno	ESP32	Pixel 3a
Clock Frequency	16 MHz	240 MHz	1700 MHz
RAM	32 KB	520 KB	4 GB
WiFi	N	**Y**	**Y**
Hardware multithreading	N	**Y**	**Y**
Bluetooth	N	**Y**	**Y**
Rewritable storage	N	**Y**	**Y**
External SD Card	N	**Y**	N
Ethernet	N	**Y**	N
IO pins	**20/22**	**39**	N/A

Table 1: A comparison between the ESP32, Arduino Uno (with an ATmega328P microcontroller), and Pixel 3A (with a Snapdragon 670 microprocessor)

Note: Missing functionality can be added using external electronics

In addition to the I/O in the above table, the ESP32 also supports SPI, I^2S, I^2C, CAN, as well as dedicated hardware for accelerating hashing, encryption,

signing, decryption, and cryptographic random number generation. The entropy source for the RNG is background noise collected from the RF module.

For convenience, the ESP32 chip is available both on its own, and in pre-assembled modules that can be soldered on to a board. These modules offer a number of advantages over designing a customised solution, as they already include flash storage, a PCB antenna, and most of the support circuitry required to make the ESP32 chip run. Some modules also include external SPI RAM. This circuitry is shielded under a metal can which prevents RF (Radio Frequency) interference from the ESP32 from exiting the module. These modules have already been approved by regulators, including the FCC (United States) and CE (Europe). Because a large cost in creating a new device is getting RF regulatory approval, these modules can make developing new devices easier.

The ESP32 module used in this project is the ESP32-WROOM-32D, containing 4MB of flash memory. This module alone, however, is insufficient as a development tool because it lacks a programming interface. In normal operation the ESP32 boots with GPIO0 (General Purpose Input/Output Pin 0) held high (+3.3 V), but to enter bootloader mode GPIO0 must be held low (0V). This allows the ESP32 to receive a program over UART (Universal Asynchronous Receive/Transmit). UART is an industry standard embedded serial connection, sending data one bit at a time over a bidirectonal protocol. Once the ESP32 is in bootloader mode, a python program called `esptool.py` allows a developer to send compiled code to the device.

Similar to how the ESP32's functionality is packaged into modules to make integrating it easier, the modules are then attached to boards to make programming them easier. These boards typically have buttons to restart and enable bootloader mode, a USB to UART chip which enables programming the device over USB, and a voltage regulator which converts the 5 volt USB power into the 3.3 volt power which the chip requires.

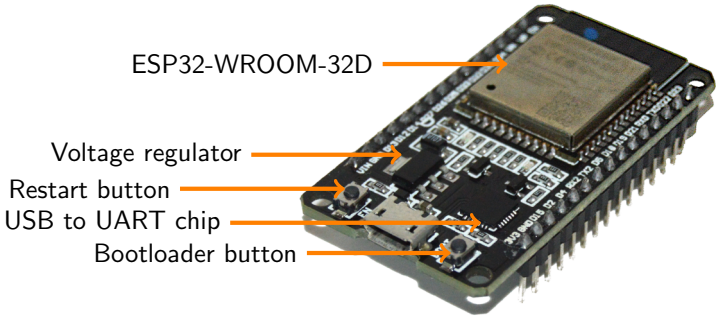


Figure 1: The ESP32

The ESP-IDF (IoT Development Framework) is a library provided by Espressif systems. It integrates into and extends the dedicated hardware provided by the processor. For example, Writing code against the WiFi transceiver requires using the IDF's TCP, UDP, or HTTP library. Similarly, cryptography support is provided through the IDF's port of mbedtls, and the SPI flash is exposed through several libraries inside of the IDF.

5.2.2 SPIFFS

One particularly useful part of the IDF is SPIFFS (SPI Flash FileSystem). It exposes a section of the SPI non-volatile flash storage as a basic filesystem. SPIFFS does not support directories, so a file saved as `/spiffs/a/b/c.txt` would be in the same directory as `/spiffs/d.txt`. SPIFFS also does not support journaling, so if power is removed halfway through a write operation, it must be reformatted. This means that the ability to restore from a complete format is important. SPIFFS also limits filenames to 31 characters.

5.2.3 WebAssembly/wasm3

Although the target of this project is the ESP32, the goal is for it to be architecture agnostic. This includes both running on different instruction sets and running on different types of computer including desktops and servers. The problem of targeting different types of device using the same standard is similar to the problem websites were made to solve. The environment of the web adds the bonus that browser-based runtimes are sandboxed (isolated) and have been aggressively tested for vulnerabilities.

Javascript, though popular, is a bad fit for creating this platform. Efficient runtimes are large and are usually themselves very architecture-dependent. In

addition, embedded developers tend to favour strong, static typing and DIY memory management, historically meaning C but a variety of newer languages such as Rust, Golang, and C++ have also been gaining popularity. Emscripten and later WebAssembly were created to enable C-style languages on the web. The former compiles code to a highly optimizable subset of Javascript, asm.js, which is backwards compatible with any Javascript engine but runs at near-native speeds in a supported one [9]. The latter is an extension to asm.js which abandons the link to Javascript entirely, compiling code to a new instruction set running on an idealised processor [10].

WebAssembly (often shortened to WASM) is intended to be executed using a AOT (ahead of time) or JIT (just in time) compiler, translating the abstract WASM instructions into real instructions to be executed on the processor. The difference between JIT and AOT is that AOTs translate into code before the program is run, while JITs compile just before the code is executed. JITs are generally faster because they can optimize based not just on static analysis but also the actual, runtime values of the variables being used. Slower than both types of compiler are interpreters, which iterate over the commands one by one and execute them by calling functions associated with each command. Although compilers are prefereable, they are harder to implement, harder to secure, harder to debug, and require porting to each instruction set they must run on. Because of WASM's relative immaturity, only two partially implemented runtimes exist for embedded architectures;

- WebAssembly Micro Runtime (WAMR), the official Bytecode Alliance runtime for running webassembly on microcontrollers. WAMR can run both in AOT and interpreter mode. WAMR is well implemented, but lacks ESP-IDF support and porting the entire runtime is beyond the scope of this project.
- Wasm3, an unofficial interpreter for WebAssembly. Although wasm3 is less professionally implemented, it has excellent support for ESP-IDF. Its simplicity also aids in modifications and experiments.

Wasm3 was chosen for this project. Prior to running WebAssembly using wasm3, it must be generated. In order to keep the project as a single programming language, C++ was chosen as the source language to compile WebAssembly from.

5.3 System Design

5.3.1 Machines

The fundamental, indivisible unit of Nihilo is the machine; many machines can be stored on each physical device. Machines are an idealised and abstract version of a computer; communication, execution, and data storage all happen at the machine level. Communication between and within devices happens from one machine to another, using remote procedure calls. Within SPIFFS, each machine is represented by a JSON and WASM file; the JSON storing data and the WASM storing code.

Many machines can be stored on one device, potentially including machines providing self-contained units of functionality developed by third parties. In order to be useful, however, the user must create a single, master machine, tying together all of the others. When the device boots it will connect to WiFi, load the master machine, and execute its `entry()` function. This can then be used to instantiate functionality in other machines.

5.3.2 Security Model

For security purposes, machines are sandboxed from each other, but have absolute power over their internal environment. This is because machines are from a single origin, analogous to how javascript has absolute power over its own HTML under the same origin policy, but can only communicate with other websites through specific channels. Every machine in Nihilo has an asymmetric keypair. Because every communication in Nihilo has an origin and destination machine, it can be cryptographically verified that;

- The message was sent only by the stated origin.
- The message can be read only by the stated destination.

RSA (Rivest–Shamir–Adleman)[11], the industry standard algorithm for asymmetric cryptography, was initially considered for this task. Although RSA is very secure, developing for the microcontroller platform exposes a set of issues with RSA which are usually hidden by the power of modern computers. The first of these problems is that key generation (keygen) is profoundly computationally intensive. RSA's hardness assumption is a stronger version of its keygen process; both involving the factorisation of very large numbers. RSA keygen on the ESP32 takes in excess of 20 seconds.

ECC [12](Elliptic Curve Cryptography) is a set of cryptographic primitives based on modular elliptic curves. ECC was built into an asymmetric cryptosystem

called ECDH (Elliptic Curve Diffie-Hellman), which is far more computationally efficient than RSA. It has a number of advantages over RSA;

- Keygen is finished in 200ms, a 100 times speedup.
- ECDH is not an encryption algorithm, but a key agreement protocol where all of the necessary information is already embedded in the public key of the peer. Both RSA and ECDH need a symmetric encryption algorithm for the bulk exchange of data, but RSA requires a dedicated protocol to send the encrypted symmetric key from one peer to the other while ECDH does not.
- ECC curve25519 public and private keys are 32 bytes long, as opposed to at least 128 bytes with RSA.
- Every pair of peers has its own secret, which requires the private key of one and the public key of the other. This negates the need for a dedicated signature, which would be required under RSA.

The sum of these advantages is a massively streamlined communication process compared to one implemented using RSA. The 200ms keygen means that machines can be created much faster, while the implicit signature and key exchange negate the need of an SSL (Secure Sockets Layer) style intermediate layer.

Because the cryptographic verification requires holding the keypair, ownership of the keypair is for all intents and purposes ownership of the machine. The security model breaks down if multiple devices hold the same keypair, so transferring private keys between devices or exposing them to WASM code is **explicitly forbidden**.

There are two desired capabilities of any asymmetric encryption scheme; security and authentication. Security is the knowledge that only the recipient of the message can decode it. Authentication is the knowledge that the recipient is who they say they are. The ECDH algorithm used by the Nihilo runtime implicitly provides security, but not authentication, which would require one of;

- A dedicated Public Key Infrastructure (PKI), built on ECC, allowing any claims of identity to be traced back to a common trusted third party.
- An integration into an existing PKI, such as SSL.
- A hybrid approach, where trust is rooted in SSL but becomes transferred to Nihilo at some point down the chain to the device attempting to authenticate.

Ultimately, the alternative to building a PKI chosen was using the public keys as the primary identifier of a machine. Because there is no known mechanism

to efficiently forge a public key, the resulting keys are almost guaranteed to be unique.

Once ECDH has been run, transforming the keypair of one machine and the public key of the other into a shared secret, the message must be symmetrically encrypted. The algorithm chosen was AES[13], an efficient block cipher. AES works one block at a time, applying encryption to 16 bytes, then moving on to the next 16 bytes. The original implementation of AES, called ECB mode, had a flaw where identical blocks were encrypted as identical ciphertext, exposing ECB encryption to a variety of attacks, which were rectified with CBC mode. This mode uses the previous block to help encrypt the next, so requires an initialization vector to randomise the first block.

Because the initialization vector is required to decrypt the data, it is included in Nihilo as the 0th block of any encrypted data. If x is the length of the unencrypted data and b is the block size (16 bytes), the formula for calculating the length of encrypted data is as follows;

$$(\lceil x/b \rceil + 1) \times b$$

32-byte public keys can be unwieldy, for instance SPIFFS filenames are limited to 31 characters long including the extension. In order to simplify this, and to make user interface easier, a 12-byte ID can be used. This is the first 12-bytes of the SHA256 digest of the public key. It is worth noting that while creating a fraudulent key with the same ID is extremely computationally intensive, it is not outside the realm of possibility with future machines, so IDs cannot provide the same security guarantees as public keys. Between devices, Nihilo always uses the full 32 byte public key.

The most important factor in creating a random identifier is minimizing the risk that two entities are randomly assigned the same ID. Poorly engineered identification schemes are vulnerable to a class of exploits known as birthday attacks, in which a forged entity that happens to have the same ID can impersonate the target entity. Thanks to the pigeonhole principle[15], it is known that any mapping which reduces a large space (all public keys) down to a smaller space (all IDs) must have distinct items in the large space which map to the same item in the small one. In the context of hashing, this is known as a hash collision.

Protecting against a targeted birthday attack, attempting to forge an ID for one specific target, can be achieved with an 8-byte ID, corresponding to a search space $2^{8\times8}$ hashes wide. If it was possible to check at a rate of 1,000,000 hashes per second, the search space would be exhausted after more than 500,000 years.

It is, however, insufficient to ensure that attackers cannot artificially create a fraudulent ID. The 12-byte length of the ID was not chosen to prevent brute forcing of a **specific ID**, but to prevent any **pair of IDs** happening to be identical. This is both more likely on a per-ID basis and the risk comes up more often, every time a machine is created on any device. Assuming the hardware random number generator and SHA256 algorithm are both mathematically ideal, the ID space is $2^{12\times8}$ hashes wide. If n is the number of assigned hashes, the probability that there exists at least one pair which are the same, p, is approximated by the following formula[14];

$$n = \sqrt{2 \times 2^{12\times8} \times ln\frac{1}{1-p}}$$

p	n	≈ equivalent to assigning one hash to
10^{-10}	1.26×10^{10}	every person alive today (7.8×10^9)
10^{-7}	1.26×10^{11}	every person who has ever lived (1.09×10^{11})
0.01	3.99×10^{13}	every red blood cell in the human body (2.63×10^{13})
0.1	1.29×10^{14}	every cell in a human body (including bacteria) (10^{14})

Table 2: Number of 12-byte hashes for a given collision probability

12-byte IDs allow for expansion to tens of trillions of machines before collision becomes an issue, so there is much room for expansion.

5.4 Implementation

5.4.1 Tooling

There are three major pieces of tooling that were required to implement Nihilo;

- A pass-through WiFi access point, to streamline network access for the ESP32s
- A directory server, to facilitate discovery of peers
- The compiler, programmer, and associated toolchains

Allowing for a simple connection to private resources is a common problem in developing open-source software. WiFi credentials, API keys and bitcoin addresses are often left in code which is pushed into public repositories. In order to develop without adding support for Eduroam, and exposing university login details in the code, create_ap was used. This turns WiFi-enabled linux

computers into repeaters, taking in one WiFi network while creating another with dependable, specified parameters that can be developed against.

The other component was an HTTP-based directory server, implemented in python. This is simply a public list of known machines, registered using POST requests, and retrieved using GET requests. To simplify handling, the server will only send back machines belonging to other devices.

On boot, the Nihilo runtime registers owned machines to this service, and queries it for those belonging to other devices. If the runtime decodes a packet from an unknown machine, it will add that to the list.

Wasic++, a clang-based C++ to WebAssembly compiler was used to create the code which the machines can run. This is then run through wasm-opt, which reduces the webassembly's size. Finally, it is run through xxd, which converts the bytes of the optimized wasm into a header file that can be compiled into the runtime. On boot, the runtime writes this into the ESP32's SPIFFS as a .wasm file.

Makefiles were used to unify the different build systems and tools (ESP-IDF uses both ninja and cmake). A single command in the src directory can compile the machine's code into a header file using wasic++, compile the runtime into an xtensa ESP32 executable, program the device, and then execute the program printing any logs to the screen. Makefiles use file modification time to only compile files that have changed, so this process usually completes quickly. Using the 'reflash' and 'remonitor' commands, as opposed to the 'flash' and 'monitor' commands, the whole flash memory, including the wasm code and machine data can be wiped.

5.4.2 Adding a simple heap to wasm3

One serious shortcoming in the wasm3 interpreter, which is not documented in their git repository, is the lack of runtime-allocated memory. The `void* malloc(size_t size)` call in which memory is dynamically allocated from the heap, has not been implemented. Although only call-stack based memory is necessary for Turing-completeness, heap-based memory can simplify programming.

Wasm3 allocates memory in 65536 byte pages. In order to maintain the integrity of the sandbox, accessing memory outside of these pages halts execution, so any implementation of malloc must return a pointer somewhere inside of this page. The Nihilo runtime is built on small functions, which are instantiated, run, and asynchronously instantiate other functions. Each individual function has a very

small memory footprint, and each wasm3 runtime exists for a short period of time.

It was decided, as a consequence of the small memory footprint of each function, and the limitations to where memory can be allocated, to reserve page offsets 0x0000 to 0x7FFF (32768 bytes) for the stack, and to reserve offsets 0x8000 to 0xFFFF (32767 bytes) for the heap. Because these pages are cleaned up after a short time, no mechanism for freeing memory was created. A `uint16_t` at page offset 0x8000 tracks the amount of allocated heap, and is incremented every time more is allocated. Breaking the Nihilo coding model, and creating highly recursive functions, or functions which allocate and free a lot of memory, would in turn break this memory allocation scheme.

5.4.3 Persistent Storage

Persistent storage was the easiest of the three requirements to meet. Once a metadata JSON file was added to keep track of of cryptographic information about hosted machines, adding a data field to that JSON file containing user specified data was straightforward. This data is itself a JSON object, essentially a dictionary, containing keys and values. The dictionary is addressed using strings, and its elements are either strings or other dictionaries. These dictionaries can in turn contain other dictionaries, so the whole structure is a tree.

The values in this tree are referenced the same way that they are referenced in javascript. The root is referenced using the empty string, and the storage system iterates down the tree to find a desired node, using full stops (.) to split parent node from child node. As an example, the string "A.B.C" refers to the C string, under the B object, under the A object, under the data object. The system currently has no functions that deal with the objects directly, and they can only be created by creating a string in an object that does not yet exist. They can only be deleted by replacing them with a string. Strings cannot yet be directly deleted, but replacing them with another string works.

5.4.4 Task Queue and Communication

One problem encountered early on in the development of Nihilo was the limited RAM of the ESP32. This is exacerbated by the fact that ESP-IDF only exposes half of this already constrained RAM as heap, with the other half being reserved for the call stack. There is only enough dynamically allocated RAM for one complete wasm3 runtime, which would seem to exclude the possibility of one runtime calling another, and then using its result in an operation. Encryption, and network communication are both also dependent on the heap. Although external

RAM chips do exist for the ESP32, requiring these would break compatibility with most off-the-shelf ESP32 modules.

The solution is to use a FIFO (first in, first out) task queue. This allows WASM function calls to be placed onto the queue as tasks and executed one by one. These tasks may optionally also specify callbacks, placing new tasks onto the queue once execution is complete. Two machines can have an asynchronous 'dialogue' through callbacks without ever loading more than one of them at a time. This model can also be extended to communication if the queue also contains calls to machines on other devices, in which the execution is replaced with encrypting and sending the call.

For simplicity, tasks in Nihilo take exactly one char* parameter (which can be nullptr), and either return a char* value, or return nothing (which is treated as a function that returns nullptr by the runtime).

Tasks are encapsulated within the `task` struct, containing;

- Origin public key
- Destination public key
- A retry count
- A pointer to the return value
- The name of the target function
- The name of the function to call on success
- The name of the function to call on failure
- The parameter to be passed to the function (a null terminated string, immediately following the struct in memory)

Because the last four are encrypted for any transmission, they are contained within a struct inside of `task` called `wire_task`. If the task executes successfully, and a success function has been set, a new task will be created. This task will have the return value (if there is any) of the previous task as its parameter, and the on success function as its target. The origin of one task becomes the destination of the next and vice versa. Failures are treated in the same way, with a description of the error as their parameter.

As with the rest of the runtime, the primary design goal of the Nihilo protocol was simplicity. Nihilo communicates using an encrypted, stateless, single-verb, asynchronous protocol. It is encrypted because every message can only be read by its sender or receiver. It is stateless because messages are not part of a sequence; a connection is made, used to send one message, and then closed. It

is single-verb because unlike HTTP, the protocol only knows how to invoke a single type of action; calling a function. It is asynchronous because it does not block until a reply has been received, rather invoking a new function when this happens. Nihilo uses TCP because a high reliability is desired, through Berkeley sockets.

Due to the stateless nature of the protocol, the runtime will only use any individual connection once. It is opened by the origin of the call, the call is sent, and then the connection is terminated. Only incoming connections and other tasks can enqueue tasks, so once the task queue is empty there is nothing to do but listen for connections. This single-threaded approach does have the downside that connections can only be received while the task queue is empty, but a well designed system should be able to work around this flaw.

In order to explore how the runtime communicates, consider the following (An API reference is in the next two sections for specific definitions of individual functions);

Device A has master machine x, with the following code;

```
NIH_VOID(entry, param){}
NIH_CHARS(RPC, param){
  //write "hello, world" into M.N in persistent JSON
  writeString("M.N", "hello, world");
  //read value back out of JSON
  char* ret;
  readString("M.N", &ret);
  return ret;
}
```

Device B has master machine y, with the following code;

```
NIH_VOID(entry, param){
  unsigned char* ids;
  //find the peer:
  int known = knownPubs(&ids, 0, 1);
  //if the peer is found, call RPC against it
  if(known > 0)
    queue(ids, "RPC", nullptr, "success", "fail");
}
NIH_VOID(success, param){
  logStr("success");
  if(param != nullptr)
    logStr(param);
}
NIH_VOID(fail, param){
  logStr("failure");
  if(param != nullptr)
    logStr(param);
}
```

If A then B is turned on, the following events occur from the perspective of the user;

1. Directory server is reset.
2. Device A turns on, connects to wifi, and registers its master machine.
3. Machine x executes its entry function, which is empty.
4. Device B turns on, connects to wifi, and registers its master machine.
5. Machine y executes its entry function, querying a non-local machine from the runtime, and sending it a call for the `RPC()` function, with the `success()` and `fail()` functions as the possible outcomes of the call. The call has no parameter.
6. Device A receives the call, records the IP address of Machine y (previously unknown because A turned on before B) and begins to execute `RPC()`.
7. Device A writes then queries 'M.N' from machine x's persistent JSON storage.
8. Machine y returns the string to Machine x by calling the `success()` function.
9. Device A prints out 'hello, world'.

If Device B were to encounter a fatal, non-recoverable error in its execution of `RPC`, it would call `fail()` instead of `success()`. Giving too much information in an error message such as the specific exception has been known to create security flaws, including ones that can be used to extract sensitive data [16]. For this reason, any error internal to the users code calls the fail handler with the parameter set a generic all-encompassing message, 'execution error'.

On a lower level, however, this process is more complicated. The code touches communication, encryption, JSON storage, and WASM execution. Sequence diagrams can be used to help explain registering and querying machines to the directory, which are contained in steps 2 and 4;

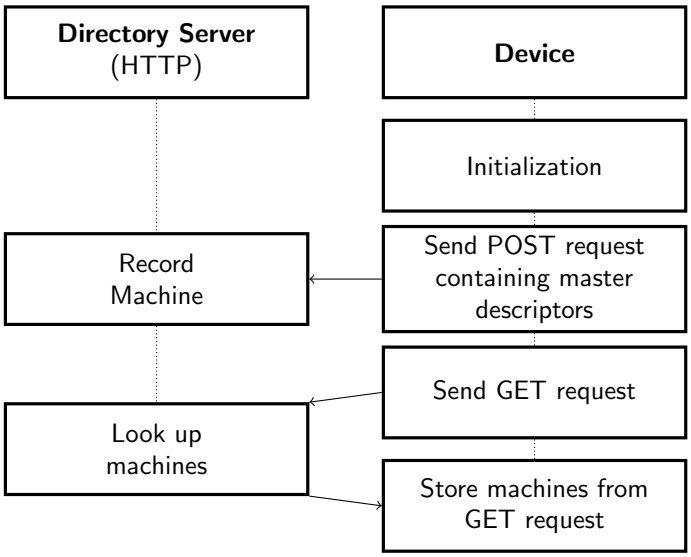


Figure 2: The machine registry and discovery process

The POST request is a machine descriptor, containing the IP address and public key of the target machine in the form `IP_ADDR:HEX_PUBLIC_KEY`. Only a single machine may be registered at once, but several POSTs may be made to register several machines. If a given machine is re-registered with a new IP address, the old IP address is deleted and the new IP is added instead. Because devices may turn on or off, or lose internet connection, the IP address in the directory may not be the most up to date, and may not work at all. All directory server operations happen on port 80 (the standard HTTP port), and against the root (http://address/) file. Because the directory server gives publicly available information, it is not encrypted with HTTPS.

When a get request is made to the HTTP server, it will collate all known

machines not at the requesting device's IP address, and send them to the device. It does this by first sending a count of how many descriptors will be sent, followed by a newline, then the descriptors also separated by newlines;

```
4
IP_ADDR:HEX_PUBLIC_KEY
IP_ADDR:HEX_PUBLIC_KEY
IP_ADDR:HEX_PUBLIC_KEY
IP_ADDR:HEX_PUBLIC_KEY
```

Steps 5 and 8 involve sending, and 6 and 9 involve receiving messages. Because these messages are encrypted, they have two packets; outer (unencrypted) packets directing them to their destination while providing information required to decrypt them, and inner (encrypted) packets containing the actual function to call.

Field Name	**Start byte**	**End byte**	**Description**
Origin Pub	0	31	Public Key of origin machine
Destination Pub	32	63	Public Key of destination machine
Contents Length	63	79	True pre-encryption length of inner packet
Body	80	N	Encrypted data (first 16 bytes are the Initialization Vector)

Table 3: Outer packet structure

The first three (Origin and Destination Pub, Contents Length) fields are contained within the `packet_header` struct for ease of handling. The only thing that can be told from intercepted packets, without the shared secret, is the message endpoint machines and the message length and frequency. While it may allow an attacker to deduce the general nature of the communication, they should not be able to tell its content. SSL exposes similar information through port numbers and message lengths, and is considered highly secure.

Once the packet has been decrypted, is parsed as an inner packet.

Field Name	Start byte	End byte	Description
Destination ID	0	11	ID of the target machine. Used to verify packet has been decoded correctly.
Function name	12	31	Name of the function to call. Null-terminated string.
On Success	32	51	Name of the function to call on origin machine if this task completes successfully. Null-terminated string.
On Failure	52	71	Name of the function to call on origin machine if this task does not complete successfully. Null-terminated string.
Parameter Null	72	72	Is the parameter null?
Parameter	73	N	Null-terminated parameter

Table 4: Inner packet structure

Like the outer packet, Function name to Parameter Null are incorporated into a struct for ease of handling. This is called `wire_task`, and incorporates all of the information needed to execute the task, other than the origin and destination machine. These public keys, as well as device-specific data are in the `task` struct. Both `wire_task` and `task` are usually followed directly in memory by the null-terminated parameter if any exists, but this cannot be inside of the struct because it is of varying length. How these packets are used can also be explained with a sequence diagram. This one covers steps 4 to 9;

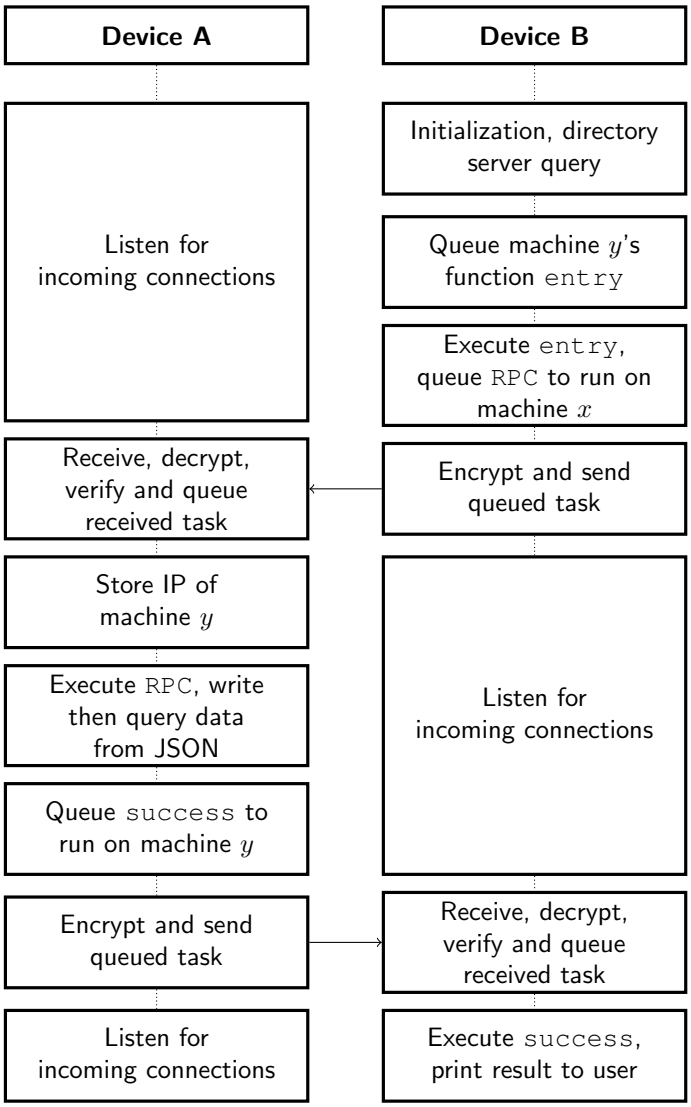


Figure 3: How a function and its return are queued

Device A stores machine y's IP address and public key when the task is received, because device A turns on before B. Because the directory server is only queried by the Nihilo runtime when the device boots, device A has no knowledge of device B or machine y until it receives the first packet. Device B does not do the same when it receives its message because it already knows about machine x.

5.5 Nihilo API

The Nihilo API, contained in `include/api.h`, is the means though which user code interacts with the runtime.

5.5.1 Nihilo API Macros

`NIH_VOID(NAME, PARAMNAME)`
Create a new Nihilo-exposed function with name `NAME`, and a `const char*` parameter called `PARAMNAME`. Replaces the normal function declaration. If it is called with no parameter, `PARAMNAME` will equal `nullptr`.

```
NIH_VOID(entry, param){
  logStr("hello, world!);
}
```

`NIH_CHARS(NAME, PARAMNAME)`
Same as `NIH_VOID(NAME, PARAMNAME)`, but returns a `const char *`.

```
NIH_CHARS(entry, param){
  return "hello, world.";
}
```

`WASM_IMPORT(MODULE,NAME)`
Internal macro, assisting with exposing functions to the user.

5.5.2 Nihilo API Functions

`void mallocWasm(void** target, uint32_t size)`
Replacement for malloc(). Pointer pointed to by `target` is is set to point to a new buffer of length `size` bytes. Allocates heap inside of wasm3's memory space.

```
NIH_VOID(entry, param){
  char* buf;
  mallocWasm((void**)&buf, 50);
  strcpy(buf, "test");
  logStr(buf);
}
```

`void writeString(const char* path, const char* value)`

Saves `value` in `path` in the current machine's JSON. Only objects and strings are currently supported to save, and only strings to directly edit, delineated by full stops(`.`). If the write is under an object that doesn't yet exist, it will create that object.

```
NIH_VOID(save_value, param){
  writeString("M.N.C.Str", param);
}
```

`void readString(const char* path, char** target)`
Looks up `path` in the current machine's JSON. Pointer pointed to by `target` is set to point to the value read out of JSON.

```
NIH_VOID(log_value, param){
  char* val;
  readString("M.N.C.Str", &val);
  if(val != nullptr)
    logStr(val);
}
```

`int knownPubs(unsigned char** IdsOut, int include_local, int include_non_local)`
Lists machines currently known to this device. if `include_local` is set to 0, machines on the device will be excluded from the list. Similarly, if `include_non_local` is set to 0, machines not on the device will be excluded. `IdsOut` is set to point to a list of the 32-byte public keys of the requested devices, and the function returns the number of machines which have been included in `IdsOut`.

```
NIH_VOID(log_value, param){
  unsigned char* ids;
  int known = knownPubs(&ids, 0, 1);
  if(known > 0)
    logStr("Devices knows of non-local machines");
}
```

`void logStr(const char* tolog)`
Log a string to the user, using the ESP-IDF monitoring program.

```
NIH_VOID(log_value, param){
  logStr("hello, world");
}
```

void queue(const unsigned char* target_pub, const char* fname, const char* param, const char* onsuccess, const char* onfail)
Add a new task to the back of the queue, to be executed after every other task currently on the queue. The task will be sent to `target_pub`, either directly if it is on this device, or through an encrypted socket if it is on another. If `target_pub` is nullptr, it will be sent to the current machine. The target device will execute `fname(param)`, and pass its result to `onsuccess` in this machine if it is a success, or `onfail if not. param`, `onsuccess` and `onfail` may be set to `nullptr`, but `fname` must be a valid string.

```
NIH_VOID(entry, param){
  queue(nullptr, "test", "hello!", "success", nullptr);
}
NIH_CHARS(test, param){
  logStr(param);
  return "response";
}
NIH_VOID(success, param){
  logStr(param);
}
```

This example prints;

```
hello!
response
```

uint64_t uptime()
Returns the number of microseconds since the system booted.

```
NIH_VOID(log_value, param){
  logStr("uptime:");
  logStr(decstr((int)uptime()));
}
```

char* decstr(int input)
Returns the string version of the passed integer, allocated into a new buffer.

```
NIH_VOID(log_value, param){
  logStr(decstr(5));//prints 5
}
```

int strdec(char* input)
Returns the integer corresponding to the passed string.

```
NIH_VOID(log_value, param){
  if(strdec(param) == 0)
    logStr("passed zero");
  else
    logStr("passed non-zero");
}
```

void setReturn (const char* ret)
Internal function, assisting with returning data to the runtime.

void getParam(char param)**
Internal function, assisting with passing the parameter from the runtime to WASM code.

6 Testing and Results

In section 5.1, the high-level requirements are listed. There are two types of requirements, **must** and **should**. **Must** level requirements do not require much experimental setup, only a single verification. **Should** level requirements, on the other hand, are more relative and require experimental parameters to be set beforehand.

6.1 Execution

1. **Must** support mature languages, compilers and tooling. This is a fact of the design of Nihilo, and so does not need to be tested.
2. **Should** execute tasks quickly. It should take under 5 milliseconds (ms) to create an execution context, execute a function, clean up the memory, and then move on to the next task. It was be calculated by taking the time

between the start and end logs of the following code, averaging over 10 runs, and dividing by 100:

```
NIH_VOID(test, param){
  if(strlen(param) > 0) logStr("end");
}
NIH_VOID(entry, param){
  logStr("started");
  for(int i = 0; i < 100; i++){
    char* arg = (char*)(i==99?"end":"");
  queue(nullptr, "test", arg, nullptr, nullptr);
  }
}
```

Start Time(ms)	End Time(ms)	100 Tasks Time(ms)	Single Task Time(ms)
2137	2537	400	4
2627	3057	430	4.3
5657	6067	410	4.1
2127	2527	400	4
3127	3547	420	4.2
2147	2557	410	4.1
2147	2557	410	4.1
2147	2567	420	4.2
2177	2597	420	4.2

Table 5: Queueing test results

The average single task time is 4.13ms.

6.2 Storage

1. **Must** be able to save strings while the device is off. This is proven as a part of tests 3, 4 and 5.
2. **Must** be able to structure the saved data. This does not need to be tested, as the structure of the saved data is an implicit part of the design.
3. **Should** write data quickly. It should take under 5ms to write a short string to its persistent store. It was calculated in the same way as the execution data, using the following code, which writes "Hello, world" into JSON keys "0", "1", "2" ... "99".

```
NIH_VOID(entry, param){
  logStr("started");
  for(int i = 0; i < 100; i++)
    writeString(decstr(i), "Hello, World");
  logStr("finished");
}
```

Start Time(ms)	End Time(ms)	100 Writes Time(ms)	Single Write Time(ms)
2887	5137	2250	2.25
2137	4377	2240	2.24
2137	4397	2260	2.26
15117	17367	2250	2.25
6827	9067	2240	2.24
2147	4387	2240	2.24
12137	14377	2240	2.24
2637	4897	2260	2.26
2167	4417	2250	2.25
2857	5117	2260	2.26

Table 6: Writing test results

The average single write time is 2.49ms.

4. **Should** read data quickly. It should take under 5ms to read a short string from its persistent store. It was calculated by taking the 100 items written to the disk in the previous test and reading them back out.

```
NIH_VOID(entry, param){
  logStr("started");
  char* buf;
  for(int i = 0; i < 100; i++)
    readString(decstr(i), &buf);
  logStr("finished");
}
```

Start Time(ms)	End Time(ms)	100 Reads Time(ms)	Single Read Time(ms)
2697	2777	80	0.08
4867	4967	100	0.1
2587	2667	80	0.08
5797	5877	80	0.08
5297	5387	90	0.09
3857	3927	70	0.07
4797	4887	90	0.09
2137	2207	70	0.07
3847	3927	80	0.08
7297	7377	80	0.08

Table 7: Reading test results

The average single read time is 0.082ms.

5. **Should** retain data accurately. It should be able to read 1000 values out of the persistent store without making any errors. It was tested similarly to the previous step, except checking the output of the read against the known input.

```
bool same(const char* a, const char* b){
  int i;
  //iterate over two strings
  for(i = 0; a[i] != 0 && b[i] != 0; i++)
    //false if different char
    if (a[i] != b[i]) return false;
  //false if different length
  if(a[i] != 0 || b[i] != 0) return false;
  return true;
}
NIH_VOID(entry, param){
  char* buf;
  logStr("started");
  for(int i = 0; i < 100; i++){
    readString(decstr(i), &buf);
    if(!same(buf, "Hello, World")){
      logStr("not same");
      break;
    }
  }
  logStr("finished");
}
```

This test was repeated 10 times, and did not hit the "not same" failure condition.

6.3 Communication

1. **Must** ensure messages arrive as they are sent. This does not need testing, as it is a part of the TCP protocol, which is assumed to work.
2. **Must** allow the user to call a function on the target device, using a string as a parameter and as the return value. This is also an implicit part of the design, which is tested as a part of 4 and 5.
3. **Must** be secure, the content of a message should not be visible to an attacker. This cannot be proven, only disproven. If many researchers cannot find a flaw, but a single researcher can, the content of the message is visible to an attacker. All that can be used for this criteria is an assurance that the code was tested and audited thoroughly.
4. **Must** be stable over long periods of time. This was tested by leaving device

on for 24 hours, then executing the code in test #5. This was repeated 3 times over 72 hours.

5. **Should** be low-latency, sending a message and receiving a reply in under 10 seconds. The following code was put on two devices, one of which acted as a server and the other, the client. The client queues `RPC()` on the server, and when it executes successfully, the server queues `success("finished")` on the client, which is then printed out. This test is repeated 10 times.

```
NIH_CHARS(RPC, param){
  return "finished";
}
NIH_VOID(entry, param){
  unsigned char* pubs;
  if(knownPubs(&pubs, 0, 1) > 0){
    logStr("started");
    queue(pubs, "RPC", nullptr, "success", nullptr);
  }
}
NIH_VOID(success, param){
  logStr(param);
}
```

Start Time(ms)	End Time(ms)	Single Call Time(ms)
5557	27287	21730
10767	19627	8860
11257	20537	9280
4757	16697	11940
2047	21717	19670
2087	11947	9860
11257	23507	12250
7787	17367	9580
5747	14927	9180
2577	23817	21240

Table 8: Calling test results

The average call time is 13359ms.

7 Analysis and Conclusion

7.1 Analysis

The ESP32's logging system appears to buffer results and emit them every 10ms. While this likely affects the results, it accounts for less than 15% of even the smallest result (70ms). Because the tests are repeated 10 times, and the results are at least 7 times the logging interval, it is assumed that any distortions to the result are evened out when it is averaged.

For the execution test, the average of the single task time is 4.13ms which beats the minimum requirement by almost 20%. The results are also tightly grouped in duration differing by at most 0.3ms, indicating that the code is well optimized. While this is a good result, it is worth noting that performing the same test by calling it directly rather than queuing the task is so fast that start and end logs are printed in the same millisecond. Although calling functions, and managing the queue have some amount of associated overhead, by far the greatest factor is constructing the sandbox. The wasm3 runtime has to read the webassembly from disk, parse it, and link all of the functions. It also has to allocate and then set over 65Kb of memory to zeros to ensure the integrity of the sandbox. Finding a way to load a module once, and then keep it cached in ram without having to continually re-parse it would significantly accelerate this test, as would adding some kind of dynamic memory expansion to wasm3. Unfortunately, there may be no easy way to significantly accelerate speeds inter-sandbox speeds so they almost match calling a function within the sandbox, or natively.

The next three tests were against the JSON persistent store. The first was a write speed test, and while it met the speed targets well, there is likely significant scope for optimization. The next was the read speed test, which was over 5x faster than the target. Finally, the accuracy test was perfect for the 1000 values it ran against.

The final quantitative test was associated with communication. This was the only test that did not meet the baseline set, taking an average of more than 13 seconds when the target was 10. Dividing to give an average of 6.7 seconds for one leg of the communication, from one peer to the other rather than a round trip. While this is very bad for most communication protocols, it is still applicable for many applications. While this would not be suitable for running a user interface, it would be for data being logged back to a central server from an IoT device, or for checking for software updates from a central server.

A variety of factors are at play here, each contributing to the round trip time (RTT). It is clear from the 4.1ms single task time that the time required to

create and execute the tasks once they got to their intended destination was not a significant factor. The first is that both key derivation and encryption take a significant amount of time. While accelerating the AES encryption is unlikely, replacing the ECDH key derivation step with a cached key would allow the peers to communicate much faster after the initial connection, representing a large amount of computation saved for at most 72 bytes of key and secret. The second factor is that creating sockets is both long and of an uncertain duration. Sometimes, a TCP connection goes through straight away, sometimes it requires several tries to complete the connection. Similarly, the packets themselves, although small, might require several retransmissions to be received. The TCP-based factors were also exacerbated by the variability in the quality of the WiFi hotspot being emitted from the development computer. Sometimes, it would run anomalously slow, and occasionally it would fail silently, only indicating this failure through refused connections. The sockets could be more available if they were held open longer after receiving a message and this would especially help with replies to messages which are generated quickly, but it would also require a change in the architecture of Nihilo to support multitasking, so that the sockets could be managed while also executing the task.

7.2 Conclusion

What the raw timing results do not show is which parts of Nihilo used a disproportionate amount of development time due to pre-existing technical debt. In computer science the phrase 'technical debt' has come to be mean the spiralling costs associated with not fixing a problem, building up similarly to how compound interest builds up on unpaid debt. Whether these costs are financial or merely personal, they are avoidable losses to a poorly designed system. As workaround builds on workaround, the whole system expands to become difficult to understand and difficult to maintain. The wasm3 webassembly runtime was chosen out of a desire to avoid the clearly defined effort of porting a more competently implemented runtime, but what wasn't known at the time was the less clearly defined cost of paying off the wasm3 developer's technical debt.

The wasm3 runtime would reject code which compiled and executed in other environments, apparently at random. This caused tests to have to be rewritten several times before they would execute at all. It also created an extra layer of obfuscation that needed to be navigated before the runtime's other bugs could even be found. It had no heap, no implementation of the C or C++ standard library, and very intermittent support for other implementations of those libraries compiled in with the wasm executable. Going forward with development, the first priority will be to excise this runtime from the codebase in favour of

WAMR.

Although it was a part of the specification, along with execution and communication, storage did not receive as much development time as the other two components. Data is stored very simply, a JSON file is read from SPIFFS. Writing moderate amounts of data with a machine, into the the kilobytes, would likely be much slower than writing a short string. Every write requires opening the whole file, parsing the JSON, modifying it, converting it back to a string, then saving it again, meaning that writing many values to a file requires linearly more time for each write. It is unsurprising, given that SPIFFS and cJSON are well-tested and professional libraries that the accuracy test was perfect. A better persistence system would abandon the file-based storage all together, and focus instead on trees. Trees are, after all, what is being indirectly stored through JSON either in the machine metadata, or the user-defined data. They can be efficiently represented using a key-value store, for which the ESP-IDF already has a library, and do not require all of the data about a machine to be queried and then rewritten to storage when one value must be changed. A machine data caching system based on the current design would have to work caching whole machines at once, so would be of limited value, but a tree-based system would enable the granularity for caching to be effective.

Other priorities in the ongoing development of the Nihilo runtime include re-evaluating the core event-loop on which it is based. Just because only one execution context can fit in RAM at a time does not mean that other, smaller threads can't be handling communication, storage, garbage disposal, and other similar tasks. In addition, there are many situations in which more runtimes could fit in the RAM;

- The ESP-IDF could abandon its limit on only allowing half of the memory to be dynamically allocated
- WAMR could have a smaller RAM footprint
- A new ESP32 with more internal RAM could be released
- Nihilo could start supporting external RAM chips
- One runtime could be loaded at a time, but temporarily save its memory to flash to allow for multitasking.

Some combination of these factors is likely to emerge in the continued development of Nihilo. The event queue is unlikely to go away, although it was introduced as a workaround its versatility in dealing with communication between machines is extremely useful. What is more likely is that the event loop will be folded into a multithreaded architecture, with multiple runtimes removing

tasks from the queue, using them, and placing new tasks back onto it. This introduces its own problems with thread safety, but large parts of Nihilo have already been written with mutexes which prevent multiple threads accessing the same resource. Mutexes in turn introduce their own problems, with deadlock (when a pair of threads are each waiting for the other to unlock to proceed), but the current architecture of Nihilo is based on locking one resource at a time, making this impossible.

Such a change would also be a prerequisite to extending beyond microcontrollers to phones, computers and servers. Although the Nihilo communication model is interesting, to become useful for IoT devices it must support communication between different types of devices. An internet connected toaster is of little use if it can only talk to other toasters. Because modern servers derive much of their power from supporting many threads, running a single single threaded runtime on a server would be a waste. Private keys are banned from transferring from one physical device to another, so this change would also require the ability to create named identities which could be applied to several machines or change over time, in the a similar way an SSL certificate is useful as a named identity for an IP address. This could be achieved by exposing the directory server over HTTPS rather than HTTP, and applying name tags to the returned machines, creating a primitive hybrid PKI as discussed in the security model section.

Although many developers like the languages that compile directly to webassembly, there are more developers who don't. According to the StackOverflow developer survey[17], only 23.5% of developers know C++, 20.6% know C, and 8.2% golang. This is compared to 67.8% knowing javascript, and 41.7% python, which are both dynamically typed languages that do not directly compile into webassembly. While the majority of current embedded developers favour the former type of language, there has been a large push to enable a new generation of developers in embedded development using the tools and patterns that they are used to. One way to cater to these people without changing the architecture of Nihilo would be to create a lightweight javascript to webassembly JIT compiler, perhaps adapted from V8. In this pipeline, javascript would be compiled to webassembly by a JIT packaged in a special machine, which would then be further compiled to xtensa machine code by WAMR. The main advantage of this method is that it would support any platform currently targeted by WAMR, without needing modifications. New languages could be supported by devices simply by pushing new language machines to them.

The final place where Nihilo could be expanded is package management. Packaging functionality into easy to install units is popular in many places. Debian, Python, and Node.js being fundamentally different types of software does not

stop them having competing package managers, and there is no reason why Nihilo couldn't have its own. Machines already exist as pre-existing analogies to packages, and even though private keys cannot transfer between devices, devices could still store immutable prototype machines, signed but without a private key and create verified instances of those prototype machines on request. This would require a more Copy-On-Write approach to handling data storage, however, as only storing changes to machines would be prefereable to creating an entire copy.

Aside from these future prospects and minor if frustrating issues with wasm3, the author is satisfied with what Nihilo was made into. Meeting or exceeding all of the requirements except one, which it missed by less than 35%, this project is a success if a qualified one. Beyond the High-level requirements, the fundamental idea of this project has been proven; that a computer can still perform basic tasks if the patterns of storage, execution and communication are made simpler and more user friendly. The most important problem moving forward is not one of features but one of systems design; how to implement the required features without either making what has already been implemented more complicated, or making the project sprawl to the point that it is unattractive to learn.

8 Reflection

The objectives in the Project Initiation Document (PID) were;

- Gain a better understanding of Rust
- Use as many existing tools as possible, to avoid reinventing the wheel
- Finish on time

Rust was the language initially specified to create the project in, before the target microcontroller was specifically chosen as the ESP32, supporting only C++ and C. While I have not gained a better understanding of rust, I can say that I have gained a better understanding of C++ and C. I now have a greater knowledge both of how to use the language, and how it works.

In terms of using as many pre-existing tools as possible, I now see that creating a solution using as much as possible of what already exists has limits. A smart developer uses what exists and **works**. Giving the components of my project a test to find obvious bugs before committing to them would have saved a great deal of pain and hassle, even if it might have cost me some time. Sometimes, modifying a good solution can work better than using a bad one.

Finishing on time is the third objective, and the only one which is in the future

as I type this sentence. While it is extremely unlikely that I will not finish on time, the scope of this project has highlighted some deficiencies in my ability to manage time. Sticking more closely to the schedule I planned in my PID, perhaps by setting reminders for myself or using a calendar would have lessened the stress of finishing quickly.

To me, by far the most interesting, and frustrating part of this project was writing the code. It required problem solving, reading obscure and outdated documentation, and searching for header files and other implementations in open source projects on github. Many problems could only be solved through combining these approaches, and using brute force to try several solutions. Unlike higher level languages like python or javascript, low-level languages often return cryptic error messages, if any at all, and do not at all make it clear how they can be resolved. Avoiding memory leaks added another layer of challenge not usually encountered. Between all of the issues posed, it was not uncommon to spend days working on a problem, only to find that the solution required changing other things not initially obviously connected to the problem. I did find these situations frustrating, but that frustration increased the reward of finding a solution.

The report, on the other hand, was more incremental. There was much more of a guarantee that spending x hours sitting down and typing the report brings me a distance proportional to x closer to finishing. There was less chance that I would find a whole section fatally flawed, and have to rip it out and start again like happens so often with code. This certainty, however, bred complacency, and I spent too long trying to get the code working before starting on the report. If I could do this project again, I would write the report closer to when I wrote the code, so it would be fresher in my mind.

Looking back over my university and entire academic career, it is now clear to me that the times when I have been given a problem and wide latitude to find a solution are when I do best, not necessarily in terms of grade but in terms of learning. I become more motivated and more passionate about my assigned task. I can take what I have learned academically, supplement it with my own research, and create a project which is unique to me.

9 References

References

[1] Osborne, C. (2020, April 22). Smart IoT home hubs vulnerable to remote code execution attacks. Retrieved May 11, 2020, from

https://www.zdnet.com/article/smart-iot-home-hubs-vulnerable-to-remote-code-execution-attacks/

[2] MQTT V3.1 Protocol Specification. (n.d.). Retrieved from https://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html

[3] API Quick Reference - Apache CouchDB® 3.1. (n.d.). Retrieved from https://docs.couchdb.org/en/stable/http-api.html

[4] Bott, E. (2014, December 19). Did the browser wars finally end in 2014? Retrieved from https://www.zdnet.com/article/did-the-browser-wars-finally-end-in-2014/

[5] User, S. (n.d.). What is DSA? Retrieved from http://iot-dsa.org/get-started/how-dsa-works

[6] Nemeth, G. (2020, February 8). Node Hero: Understanding Async Programming in Node.js. Retrieved from https://blog.risingstack.com/node-hero-async-programming-in-node-js/

[7] ESP32 Series Datasheet (n.d.). Retrieved from https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

[8] Xtensa LX6 Customizable DPU (n.d.). Retrieved from https://mirrobo.ru/wp-content/uploads/2016/11/Cadence_Tensillica_Xtensa_LX6_ds.pdf

[9] asm.js Specification. (n.d.). Retrieved from http://asmjs.org/spec/latest/

[10] WebAssembly Specification. (n.d.). Retrieved from https://webassembly.github.io/spec/core/_download/WebAssembly.pdf

[11] Rivest, R. L., Shamir, A., & Adleman, L. (1978). A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. doi: 10.21236/ada606588

[12] Koblitz, N. (1987). Elliptic curve cryptosystems. Mathematics of Computation, 48(177), 203–203. doi: 10.1090/s0025-5718-1987-0866109-5

[13] Announcing the ADVANCED ENCRYPTION STANDARD (AES) (n.d.). Retrieved from https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf

[14] Birthday attack (n.d.). Retrieved from http://www.winlab.rutgers.edu/comnet2/Reading/documents/Birthday_attack.pdf

[15] The Pigeonhole Principle (n.d.). Retrieved from https://www.math.ust.hk/ mabfchen/Math391I/Pigeonhole.pdf

[16] Improper Error Handling. (n.d.). Retrieved from https://owasp.org/www-community/Improper_Error_Handling

[17] Stack Overflow Developer Survey 2019. (n.d.). Retrieved from https://insights.stackoverflow.com/survey/2019#most-popular-technologies