TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

OIVA MOISIO

LIGHTWEIGHT IMPLEMENTATION OF LIQUIDIOT RUNTIME

Master's thesis

Examiner: Professor Kari Systä

The examiner and topic of the thesis
were approved on 1 October 2018

ABSTRACT

OIVA MOISIO: Lightweight Implementation of LiquidIoT Runtime
Tampere University of Technology
Master of Science Thesis, 53 pages, 9 Appendix pages
November 2018
Master's Degree Programme in Science and Engineering
Major: Computer Science
Examiner: Professor Kari Systä

Keywords: thesis, liquidiot, iot, runtime environment

The goal of this master's thesis is to design and implement a memory efficient alternative for Node.js based ECMAScript runtime in LiquidIoT system. The reason for this, is that Node.js based version consumes significant amount of memory and is not the best fit for embedded systems, that have limited amount of memory available.

This master's thesis focuses on describing this new runtime in a significant detail as well as verify the memory wise performance when comparing to the Node.js version of the runtime. Verification of the performance is done through rigid measurements with clear intentions.

The creation of memory efficient runtime is successful in demonstrating that this kind of work can be pursued further and even improving the current implementation is reasonable. In measurements the implemented runtime was able to outperform perform Node.js version memory wise in almost all measurements that were thought of by a significant margin.

TIIVISTELMÄ

OIVA MOISIO: LiquidIoT Projektin Ajonaikaisen Ympäristön Kevyt Toteutus
Tampereen teknillinen yliopisto
Diplomityö, 53 sivua, 9 liitesivua
Marraskuu 2018
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Tietotekniikka
Tarkastaja: professori Kari Systä

Avainsanat: opinnäytetyö, liquidiot, iot, ajonaikainen ympäristö

Tämän diplomityön taivoitteena on suunnitella ja toteuttaa muistinkulutukseltaan tehokas vaihtoehto Node.js:ään perustuvalle ECMAScript ajonaikaiselle ympäristölle LiquiIoT projektissa. Node.js versio kuluttaa suhteellisen paljon muistia monien sulautettujen järjestelmien tarkoitukseen.

Tämä diplomityö keskittyy kuvaamaan tämän uuden ajonaikaisen ympäristön yksityiskohtaisesti, sekä vahvistamaan pienempi muistinkulutus, kun verrataan Node.js toteutukseen. Nämä mittaukset suoritetaan yksityiskohtaisesti selkeillä päämäärillä.

Yleisesti muistiltaan tehokkaan ajonaikaisen ympäristön luonti onnistuu osoittamaan, että tämänkaltaisen työn tekmistä on mahdollista jatkaa. Mittauksissa tämä uusi ympäristö onnistui päihittämään Node.js versioon perustuvan toteutuksen muistiltaan melkein kaikissa kategorioissa mitä mitattiin, suhteellisen suurilla marginaaleilla.

PREFACE

It was surprisingly difficult to get all the written stuff related to the master's thesis to be reasonably understandable. Kari Systä helped a lot in pointing out some of the required sections or paragraphs needed to enhance the understanding of the overall picture.

In Tampere, Finland, on 21 October 2018

Oiva Moisio

CONTENTS

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

IoT	Internet of Things
RAM	Random Access Memory
REST	Representational State Transfer
API	Application programming interface
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
UI	User Interface
I/O	Input/output
URL	Uniform Resource Locator

KiB	kibibyte, 1024 bytes
MiB	mebibyte, 1024^2 bytes
GiB	gibibyte, 1024^3 bytes

1. INTRODUCTION

The goal of this master's thesis is to create a memory efficient alternative of a LiquidIoT runtime. This is because the original Node.js based runtime consumes a significant amount of memory when really small IoT-devices are concerned. A secondary objective is to demonstrate that it's possible to create different types of LiquidIoT runtimes than the original one, even if Node.js is not used. This master's thesis focuses on making sure that the architecture and functionality of the implemented runtime is clear enough to be understood and measurements about the performance of both of the runtimes are as fair as possible and if there is any bias it would be favorable towards the Node.js runtime.

Node.js based runtime has the performance speed advantage of V8 ECMAScript engine. However, the memory consumption of Node.js and V8 is too high and low memory version of runtime is explored in this master's thesis. A successful implementation of a new memory efficient LiquidIoT runtime could enable simple and physically small devices to interact with outside world intelligently and this behavior could be changed on the fly with only small effort.

Chapter 2 focuses on giving an understanding of LiquidIoT and its relation to the runtime in addition to all the necessary background knowledge to understand other topics discussed in later chapters. Some of the underlying information about the created runtime and basic tools to understand the measurements gained in chapter 5 are also discussed in chapter 2. Chapter 3 fully focuses on how the new runtime has been implemented and in addition, that particular chapter gives more detailed information about the idea on how LiquidIoT runtime is supposed to work. Chapter 4 is separated from chapter 3 to give a more practical approach on what is needed to run the created runtime and some practical preparations for the measurements in chapter 5.

2. BACKGROUND AND THEORY

Lightweight LiquidIoT runtime (LwLIoTR) created for this master's thesis solves one of the problems presented by the original Node.js [16] based LiquidIoT runtime (NodeJS runtime) [1]. The general functionality of NodeJS runtime is described in section 2.2.3.

NodeJS runtime has significant problem with RAM memory consumption, which is caused by the expensiveness of Node.js environment and V8 ECMAScript engine. Thus, it is not plausible to use NodeJS runtime in low memory environments. A lighter runtime environment might also have a cheaper idle processing usage between ECMAScript executions. Thus, having a lower energy consumption overall in some situations. This, however, is speculative and would need to be measured formally.

In this chapter, the necessary background information, theory related to this master's thesis and the physical implementation related to it are explained. Topics are mostly ordered by understanding required, meaning that the most basic information is placed earlier than some others that require a priori information. REST architecture is described in section 2.1. Section 2.2 and its subsections go through all the necessary information about the LiquidIoT system surrounding NodeJS runtime and LwLIoTR. After understanding how the basic system works, in section 2.4 both of the used ECMAScript engines are described. In section 2.5 the libraries that are used to help in implementing LwLIoTR are presented. Some of the fundamental concepts of Linux memory are explained in section 2.6, in sufficient detail to understand the measurements in chapter 5.

2.1 REST architecture

REST architecture [8, s. 76-106] provides a set of architectural constraints. That make, providing interoperability between computer systems on the Internet possible. RESTful Web-services offer an opportunity to read and manipulate Web-resources through unified and stateless interface. These interfaces can support HTTP-protocol's methods POST, GET, PUT and DELETE for example, as defined in [8, s. 76-106]. However, this does not exclude any additions to the stateless HTTP-protocol. These additions may include new kinds of HTTP methods such as PATCH.

The most common method is the GET method, where a request without payload is sent and the sender will receive some response particular to the REST API implemented by the receiver. This response can be practically be anything from JSON to HTML content. In addition to fetching all information, it is often possible to use query parameters to limit the query. Query parameter is defined by adding a query string at the end of an URL, for example "*/api?**key=value***". This is often used in GET method to limit the amount of

data one can receive from the GET request. However, there is not a single usage of query parameter in this master's thesis and only static URL parameters are used, for example "*/api/**{parameter}**/something/*" where "**{parameter}**" would be the parameter used. GET method is still used very frequently to get information about all sorts of things, such as, if device is registered or similar requests discussed later in different sections and chapters.

Second most common HTTP method is POST, which is reserved for sending new data or in some cases updating the data. In traditional Web programming, POST method is used to send form data to the receiving server from a static form in a Web page to create or update data. Similarly REST architecture's POST method is generally used to create data. However, the actual data that is sent can be in any format not just form data and convention is to inform about the type in the header of the request. The data can be sent with form parameters, with raw JSON or any other format that is agreed by the sender and the receiver. In this thesis form parameter is used only once, when runtime receives a file from the Application Framework described in section 2.2.2. In all other cases the data will be sent in JSON format.

In early days of REST APIs the GET and POST methods were so common that no other methods were really used in REST APIs until relatively recently. New ways of thinking about REST APIs have emerged and some uniform design patterns are starting to form, such as the OpenAPI [18], which is generally used by all components of LiquidIoT.

PUT and DELETE methods are some of the more common additions to the most commonly used HTTP methods. PUT method is generally used for updating information. In LiquidIoT runtime PUT is used to update the application (Section 2.2.5) and updating some of the information to the Resource Registry (RR) (Sections 2.2.4 and 3.1.5). DELETE method is used for deleting data. Thus, DELETE request method is quite self explanatory in its own sense. In LiquidIoT runtime DELETE method is used to delete a single application or all applications (Section 2.2.5) and in addition to this, deleting information from RR (Section 3.1.5).

The REST interfaces are used in every external connection there is in the LwLIoTR whether it is an inbounding connection from Application Framework or an outgoing connection to RR. All of these connections and REST APIs are discussed later in sections 2.2.5 and 3.1.5. The basic idea behind the REST API usage is that the Application Framework connects to the runtime (Section 2.2.3) and runtime connects to the RR.

2.2 LiquidIoT

LiquidIoT is a project that conforms with Web of Objects [20] project's principles with the aspect of being able to easily and dynamically deploy applications to remote devices. This is achieved in LiquidIoT with architecture consisting of 3 parts: Application Framework, runtime and a registry. This architecture and its functionalities are described in section 2.2.1. Application Framework is described in section 2.2.2, runtime in section 2.2.3 and

registry in section 2.2.4. Application Management API, that is the REST API the runtime offers is described in chapter 2.2.5.

2.2.1 LiquidIoT architecture

LiquidIoT architecture consists of 3 major parts Application Framework (AF), runtime and Resource Registry (RR). AF is described in section 2.2.2. AF generally allows the development and deployment of applications to remote devices. Second part of the Liquid-IoT architecture is the runtime described in section 2.2.3. Runtime handles installation and running applications, that are deployed to the device by AF. In addition to installation and running applications, runtime will send status information about applications as well as about the device the runtime has been installed on. Status information is sent to RR, which is described in section 2.2.4. RR registers and stores information about applications and devices that are in use. This information becomes available for use via REST API.

Figure 2.1 describes the basic idea on how the LiquidIoT system works and this will give us understanding of runtime's responsibilities in this system. Operations that are visible in figure 2.1 are:

1. Runtime registers to RR.
2. User develops the application in AF.
3. AF searches for devices from RR and gives them as an option in AFs UI.
4. When application is ready, deploy the application to runtime.
5. Runtime reports the result of deployment to both AF and RR.
6. Now AF can see the application deployed from the RR.
7. AF can manage the application deployed in various ways.

2.2.2 Application Framework

LiquidIoT Application Framework (AF) can be used to develop ECMAScript applications that will later be deployed to a runtime or to several runtimes by AF. AF consists of 3 different UI parts which are, writing the applications (Projects), deploying the applications (Deploy) and defining REST APIs for applications to implement. In general AF will handle the maintenance of the applications remotely in a runtime using predefined REST APIs that are described in section 2.2.5.

When creating a new application, a basic template is provided and the contents of this template are described in detail in section 3.4. However, AF provides 3 predefined and mandatory functions for the applications that should be implemented accordingly. These functions are *initialize()*, *task()* and *terminate()*. After writing applications using the inbuilt IDE. It is possible to see these applications and they can be removed or updated at will.

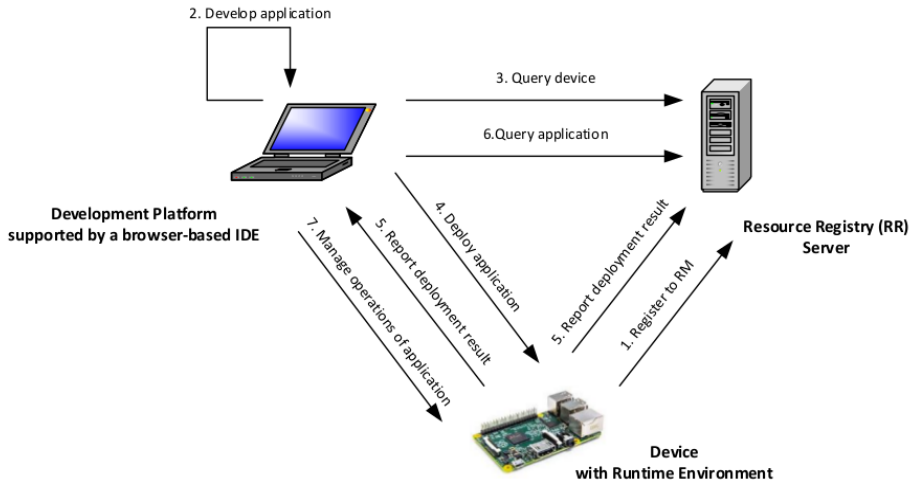

Figure 2.1. *LiquidIoT application development, deployment, and management*

Deploying an application is done by simply selecting a device or devices that we want it to be deployed to. After selecting a device, an application or multiple applications, that have been developed in Projects screen, should be selected. Then, finally deploy selected applications to selected devices. In practice this means that AF sends a separate POST request(s) that contain one application to all of the selected runtimes.

Defining applications own internal REST APIs (Application Interface in section 3.4.2) happens by using a swagger editor that can generate OpenAPI conforming API descriptions with only a little effort. AF generates necessary modifications for the application that the Application Interface is assigned to. These necessary modifications include, adding the desired Application Interface path into ECMAScript application and changing the *liquidiot.json* file to include the API. The runtimes will be able to serve these implemented APIs from an application specific path.

As AF is not in very mature stage and it is not fully documented on how should the runtime respond in different situations. This will cause different implementations of runtimes to behave differently in different situations. For example current NodeJS runtime has a different kind logging mechanism compared to the one used in LwLIoTR thus AF's application logs will not work for the LwLIoTR. Overall the AF does its job well enough so that we can use unified interface to make the necessary comparisons between the runtimes.

2.2.3 Runtime

Runtime is a central piece in the LiquidIoT system and the main focus of this master's thesis. It handles the execution and installation of ECMAScript applications when an application is deployed to the device by AF. These applications will run independently of each other in each device unless application uses other application's public API to

do actions in other applications. A Runtime hosts an Application Management API (Section 2.2.5) for deploying and maintaining the applications received from the AF. In addition to this the applications can also host their own REST APIs, as was introduced in paper Application Development and Deployment for IoT Devices [2] and discussed from LwLIoTR point of view in section 3.4.2. These "internal" REST APIs are called Application Interfaces. Runtime sometimes needs to fetch information on what RR knows about the device's runtime and about the applications that have been registered to RR.

Before LwLIoTR version of the runtime was implemented, there existed a Node.js version of the runtime, which has roughly the same properties as LwLIoTR with the exception of full Node.js support and V8 [9] ECMAScript engine (Section 2.4) support behind the runtime. In practice, Node.js implementation also has relatively large memory consumption, which makes it pretty much unusable or difficult to use on small devices that do not have abundant memory available. The memory consumption is the main reason why LwLIoTR was designed and implemented.

The Application Management API is discussed in full detail in section 2.2.5. There are no major differences in the functionality of the REST API between runtimes and all the application packages, updates ect. are designed to be exactly the same as in NodeJS runtime. This enables the use of both runtimes with same applications, as is seen in the actual measurements in chapter 5. Application Interfaces that an ECMAScript application can host by itself are discussed in section 3.4.2. The general guideline for the Application Interface hosting has not been explicitly defined. However, RR wants SwaggerFragments as the API definition. Which in turn complies with the OpenAPI-specification even though it is not in a fully human readable format.

As a summary of the runtime, runtime hosts applications inside the runtime, handles their resource consumption and communications with outside world. Applications can be deployed, updated and deleted by using the Application Management API. It is also possible to update the application state, which in practice means that it is possible to pause the application and start it again. In addition to all the previously mentioned active functionalities it is possible to fetch information about any of the applications by using the Application Management API.

2.2.4 Resource Registry

The LiquidIoT Resource Registry (RR) [14] manages the information about IoT-devices and ECMAScript applications deployed to these devices. These devices can register to RR and update their information about the applications or simply update the actual device information. RR can give information about what devices and applications have been deployed on devices. This information can be retrieved from known and predefined REST APIs from RR. It is possible to query RR about devices and applications with different criterion. Additionally, the device manager can keep track of Application Interfaces produced by applications registered.

The REST API that RR provides can do all sorts of things, such as registering new Application Interfaces offered by applications by sending a SwaggerFragment with all the necessary information to be able to connect to the Application Interface. Application Interfaces can be fetched with a GET request or deleted with a DELETE request. Similarly devices register themselves by POSTing device information to RR. LiquidIoT runtime handles registering applications and deleting applications from RR when appropriate.

Registering a device to RR happens by sending a POST request with a JSON body describing this device. This will set the device visible to all who use RR for information. Unfortunately, RR does not support deleting a device. However, in practice, it is possible for the configurations to reset. Then, there will be device configurations in RR that are difficult to use and they need to be manually processed to a new device or configuration.

There are few ways in which it's possible to manage application information in RR. Sending a POST request with application information to add a new application or it is also possible to update the information about the application by sending a PUT request to RR. And finally, it is possible to delete the application from the RR by sending a DELETE request. From a practical point of view, when LwLIoTR is used as an example, LwLIoTR POSTs a new application to RR when application is started on the runtime. After starting the runtime, LwLIoTR updates the application information when appropriate (such as *pause* and *running* states) and when LwLIoTR is shutdown or deleted, runtime asks the deletion of the application from RR.

In summary of RR, it performs the task of keeping track of all the devices, ECMAScript applications and Application Interfaces. However, RR does not do any active operations on its own and every update has to be made manually by a runtime or an AF.

2.2.5 Application Management API

Application Management API (AM-API) refers to the interface that controls the higher level functionality of a LiquidIoT runtime. These functionalities include creating a new application, updating an application, deleting an application, getting information about an application and getting logs produced by an application. There are also a few batch operations, that are, getting a list of all applications in a device and deleting all applications from a device. AM-API should have the functionality to do all these tasks in LiquidIoT runtime in order to conform with the LiquidIoT system overall. Defining this kind of API also improves the ability to create different kinds of runtimes as is done with LwLIoTR.

Creation of a new application happens by POSTing a *tgz* [10] packaged application package to url path "*/app/*" with a form parameter *filekey*, which refers to the ECMAScript application package. In case of any errors 4xx or 5xx response status should be sent by runtime otherwise status 200 will indicate success in all cases.

Updating an application is very similar to creating a new application with the difference of URL path and extra application id information passed with URL parameter. Updating the

application uses POST request method to URL "*/app/{applicationId}/*" with *filekey* form parameter, where *{applicationId}* is the id of the application. The request uses the same kind of application package file than when creating a new application.

Setting application status happens by sending a PUT request to "*/app/{applicationId}*", where *{applicationId}* is the id of the application. Possible statuses that can be set with AM-API are *running* and *paused*, which will set the application to running or paused states. Actual possible statuses that are recognized are *running*, *paused*, *initializing* and *crashed*.

Deleting an application or a batch operation of deleting all applications happens by sending a DELETE request to either "*/app/{applicationId}*" to delete a specific application or a simple DELETE request to "*/app/*" to delete all applications currently running on the device.

It is possible to get information about a certain application by using GET request to "*/app/{applicationId}*" or to get a list of all applications by sending a GET request to "*/app/*". Information about requested application describes all the necessary information about the application. This information includes *name*, *version*, *description*, name of *main* file, *id*, *Application Interfaces* and current *status* of the application. Name of *main* file, *id*, *status* and *Application Interfaces* are mandatory information for the runtime to send.

Individual application logs can be retrieved with GET request to "*/app/{applicationId}/log*". Logs have no strict format description and this leaves runtime a lot of freedom to implement suitable logging system that fits the implemented runtime.

Overall the AM-API offers sufficient amount of control over ECMAScript applications. And offers information that is required in application development through logs and status information.

2.3 Node.js

Node.js [16] is an asynchronous event driven ECMAScript runtime. It is designed so that, it is possible to build scalable network applications with it. Being an event driven runtime in this case means that the user uses callback functions to notify about an event that has happened. This also applies to interfaces that wait for input and to similar reactive functionalities. Node.js runs on a single thread that uses a non-blocking event loop. To evaluate ECMAScript code, Node.js uses Google's V8 [9] ECMAScript engine, which will be discussed in section 2.4.

Node.js event loop is a single thread that does not block unless there is a long I/O operation or a long lasting computation, which are the most common things that can block the event loop on Node.js. I/O operation can be a filesystem operation or a network request. Basically, when there is an I/O operation Node.js requires the usage of a callback making other callbacks unable to perform for the duration of the I/O operation. However, Node.js [16] reports that these I/O operations are natively almost non existent.

Node.js is scalable enough to perform multiple network connections, which in this case means tens of thousands of connections. What enables this, is the non blocking event loop that can perform the requests and callbacks without losing time, for example in a blocked thread. It is unclear how Node.js's single threaded architecture performs when compared, for example, to multi threaded network handling. This, however, is not very relevant as the Node.js's speed is acceptable even without comparing it to anything else.

In summary, Node.js is an ECMAScript runtime environment that can execute code outside of a web browser, which has been the traditional use case of ECMAScript code in the past. Each component of LiquidIoT currently uses Node.js excluding LwLIoTR.

2.4 ECMAScript Engines

An ECMAScript engine is a program that is able to execute ECMAScript source code defined by a specific standard or version of the language. The engines that are relevant and used in this master's thesis are Duktape [4] for LwLIoTR and Google's V8 [9] for the NodeJS runtime. Duktape ECMAScript engine is intended to be memory wise lightweight engine, which we can verify in section 5.3 measurements and the speed of the engine itself is also acceptable for a lot of uses that do not require natively fast execution speed. Google's V8 is more focused on the support of latest ECMAScript standards and on the actual efficiency of execution of ECMAScript source code itself, which we can also verify in measurements.

Direct quotation from Duktape's home page [4]: "Duktape is an embeddable Javascript engine, with a focus on portability and compact footprint.". This holds true even in reality, as the memory usage seems to be acceptable on an average ECMAScript application as we will witness in section 5.3. There are 2 major downsides for Duktape engine from which the most significant is the lack of support for modern ECMAScript versions. The latest fully supported version of ECMAScript is E5/E5.1[5]. This said, some features are implemented from later versions of ECMAScript 2015 (E6)[6] and ECMAScript 2016 (E7)[7]. Secondary downside is the speed and efficiency of ECMAScript code evaluation, which in turn limits the usage of Duktape in more calculation heavy applications.

Googles V8 engine is relevant because it is the ECMAScript engine behind Node.js. The basic principle of V8 is to compile the ECMAScript code just in time when evaluation is needed. V8 engine tries to detect patterns in this compilation to optimize the machine code even further and faster. In Node.js 8 and later, the V8 architecture changed from Crankshaft to Ignition + Turbofan architecture [11], which in turn increased the variety of optimizations the engine could do. This might matter in LwLIoTR measurements as Raspberry PI has older 4.x version and Laptop has 10.x version of Node.js. The power of V8 engine really shows on the performance speed measurements in section 5.3.2. However, as V8 has been designed to be runtime efficient, it naturally uses more memory even though in later versions of V8 memory issue has been somewhat addressed. To address the memory

usage of optimizations in V8 engine for both Crankshaft and Turbofan architectures. They both use as much memory as it is viable to optimize ECMAScript functions, basically the engine consumes an arbitrary amount of memory to store the optimizations. Even though these optimizations are not as aggressive in Turbofan, later version of V8 is still behind in Duktapes memory usage efficiency by a significant margin.

The ECMAScript engine will often be a focal point in a runtime and runtime has to be build around the choice of the engine whatever the choice is. The Duktape was the engine of choice in LwLIoTR from the beginning and it seems to do its job relatively well in being a memory efficient ECMAScript engine. In small applications the overhead of the runtime will be the most dominant factor and it is unclear how big the application has to be in order for ECMAScript engine to dominate in the actual memory consumption of the runtime. This said, when intentionally using the ECMAScript heap it is easy to overwhelm the normal memory consumption of a runtime by allocating a large array.

In LwLIoTR, Duktape is used to evaluate ECMAScript code. More accurately, these evaluations include ECMAScript files unique to LwLIoTR, such as application header and similar interfaces with the main program. The second usage of Duktape is the evaluation of an actual ECMAScript application that the runtime has received from AF. Third usage is general parsing of JSON from configurations and descriptions that appear in several places around the runtime.

In general the ECMAScript engines provide a baseline on how the overall runtime performs as well as how the ECMAScript applications perform. They also determine what features of the ECMAScript standard can be used in actual ECMAScript applications and sometimes in the runtime. The most important factor when considering this master's thesis is the memory usage of the engine or the framework around the runtime. As LwLIoTR does not use any external framework the memory consumption of the runtime is mostly limited to program architecture and the ECMAScript engine.

2.5 Libraries

In LwLIoTR there are 2 relevant libraries that use significant amount of memory during the execution of the runtime and these libraries are not part of any standard. These 2 libraries are **libwebsockets** [13] and **libarchive** [12]. Libwebsockets is mandatory to form HTTP connections and it is a lightweight library to be able to handle web connections without a lot of overhead. Libarchive on the other hand is a general library used to extract the application files received in tgz format and is only used to conform with general design principles of LiquidIoT Runtime described in section 2.2.3. In addition to these 2 libraries, **Boost** [3] library is used to simplify some of the operations related to filesystem.

A direct quote from libwebsockets home page [13]. "Libwebsockets (LWS) is a flexible, lightweight pure C library for implementing modern network protocols easily with a tiny footprint, using a nonblocking event loop.". A tiny footprint sounds good, however it

was not the main reason why libwebsockets was the chosen library for handling the web connections. To go past that, there actually are not many libraries with small overhead for handling web connections. Instead, often, there is a larger framework for handling these connections, thus, increasing memory consumption by a significant amount when introducing a framework and its related libraries. The choice of library that can handle both client and server side web connections was in the end made by the popularity of libwebsocket's library and, with libwebsockets it is actually possible to do almost anything related to HTTP connections including client and server side connections. A small downside in libwebsockets is, that it is a pure C library.

Libwebsockets library is used to handle all incoming and outgoing connections in LwLIoTR. Each application in LwLIoTR has their own HTTP client to communicate with outside world if needed. In this case the outside world practically means RR. More detailed description of the client usage can be found in sections 3.1.3 and 3.1.5. The most visible usage of libwebsockets is in Application Management API and Application Interfaces for each application that has their own API defined. More detailed descriptions of the REST API interfaces are described in sections 2.2.5 and 3.4.2.

Libarchive is only used to extract application packages and in comparison to libwebsocket's usage it is not a very fundamental part of LwLIoTR. However, it does use a small chunk of the memory to be able to extract both tar and gunzip files (tgz). This in mind, there is a possibility to save up some of the memory usage by optimizing the extraction process by manually implementing the tgz extraction, however, this of out scope of this master's thesis and is left as it is.

Boost is a collection of cross platform libraries that provide support for variety tasks. These tasks might include linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, or unit testing. In LwLIoTR, Boost *filesystem* module is used to provide some of the common tasks needed for filesystem operations, such as managing paths to files. A secondary usage Boost in LwLIoTR is getting the true path to the executable by using *dll* header library. It is possible to remove the secondary dependency in LwLIoTR by restructuring how and where the files are read, instead of relying on the location of the executable.

Overall the libraries do exactly what they should be doing and they have an explicit function to perform. In the measurements we will see that the total memory usage of the libraries is practically insignificant even in the most basic of cases.

2.6 Measuring memory usage of a Linux process

Understanding how Linux handles the memory and how it counts the memory usage in different scenarios is important to understand and have a clear and objective view about the memory usage. In most Linux systems there is a program called *ps* which is often used to look at the Resident Set Size (RSS). However, this is not enough to have a clear view about

the memory consumption of a single process. Although, RSS is approximately correct if measured process is the only process running in the measured system. Often RSS is not a very good representation of actual memory usage of the process and we actually want Unique Set Size (USS) that is a memory usage unique to a process and Shared Memory that is divided amongst all the other processes that use same libraries as the measured process. The combination of USS and Shared Memory is the default measurement called PSS or Dirty memory in Linux systems. For all intents and purposes when talking about process heap size, it will be considered to be same as USS.

A small detour to virtual memory is taken to open up some of the concepts described later in this section and in future chapters. Virtual memory in Linux is a non physical segment of memory reserved for a process. Process simply uses this virtual memory as it were a real memory block that a process can allocate. This virtual memory is *mapped* to a mapping table and mapping table reserves needed amount of physical memory when required. Physical memory can be located anywhere in memory space and is often scattered in RAM in an unordered manner. While virtual memory is perfectly aligned for user space process or library.

Size of the virtual memory is determined by the process and limited by system architecture, in 32 bit architecture the size of the user space virtual memory is approximately 3 GiB while in 64 bit architecture, user space virtual memory size is in the scale of 10^{19} bytes. User space virtual memory is divided between all the processes and libraries that are currently in use. For this master's thesis it is good to see that when referencing to mappings, they reference to physical memory that is used.

In addition to user space virtual memory, there exists a kernel space virtual memory, which takes rest of the available virtual memory that user space hasn't reserved for its use. For this master's thesis only user space virtual memory is relevant, although it is good to know that the running kernel will also reserve a chuck of the virtual memory for itself to use. All the measurements are in actual physical memory used as virtual memory is only measured in relatively large chucks and have no real relevance to the actual usage of physical memory.

RSS is, in its core the whole memory usage that is held in main memory (RAM) of a single process in an isolated environment. This includes the whole memory usage of all the libraries that the process is currently using. The actual memory usage of the process itself is naturally included in RSS. Using only RSS in measurements is not recommended and places a large bias toward processes that do not use many libraries, as shared memory often dominates the overall calculation of RSS memory.

USS is the portion of RAM memory that is guaranteed to be private to a process. This means that no external libraries are counted for its memory consumption and is practically the heap size of the process. Using USS as measurement is reasonable, but it still requires taking into account the libraries that are used by the process itself, especially if process is

designed to be run on a low resource environment as is the case with LwLIoTR.

PSS is a combination of USS and Shared Memory divided by the amount of processes that use the resource at hand. To be more exact, the formula is $USS + \frac{TotalSharedMemory}{amountofprocesses}$. One example of this would be that, a process A uses $50KiB$ of USS and a process B uses $300KiB$ of USS and both use $100KiB$ of shared memory that is located at the same memory region. Then we have $50KiB + 100KiB = 150KiB$ of RSS for process A and $300KiB + 100KiB = 400KiB$ of RSS to process B. Similarly $50KiB + \frac{100KiB}{2} = 100KiB$ of PSS for process A and $300KiB + \frac{100KiB}{2} = 350KiB$ of PSS for process B. In future chapters if a direct memory reference is used PSS is the default way of interpreting the memory usage unless otherwise stated. As a measurement PSS is acceptable on average, however, it is likely that we will have to slightly adjust how much does the shared memory weigh in our case.

Measuring memory usage of a Linux process is not very straightforward and requires one to think about both RSS memory calculation and PSS memory calculation. Often USS is a good way to measure a single process for purposes of understanding on how much memory does it use locally and this improves the ability to understand the process itself. However, in the end Shared Memory is the only variable component in calculations regarding the memory consumption of a Linux process. Shared memory should be treated with some caution as it is not the absolute truth about the memory consumption. However, on average it is a good way to get current memory usage by combining it with USS.

3. LIGHTWEIGHT LIQUIDIOT RUNTIME

The original goal of this master's thesis work was to create a lightweight runtime for ECMAScript applications to run on and this created runtime would be able to integrate into the LiquidIoT system. The goal was, that one would be able to run ECMAScript applications on something smaller than a consumer grade PC or powerful devices such as Raspberry Pi. Raspberry Pi was the environment the original runtime that runs on Node.js was designed to be run on and would become a good benchmark for the new runtime. In the following sections the structure and functionality of the Lightweight LiquidIoT Runtime (LwLIoTR) will be explained in great detail. Starting from the internal structure of the application in sections 3.1.1 through to section 3.1.4. External communications by LwLIoTR to RR will be described and explained in section 3.1.5. Moving on to Application Management API (AM-API) in section 3.2 and then going through on how the Duktape has been used in LwLIoTR in section 3.3 and then going to the detailed description of the ECMAScript application in section 3.4 that will be deployed by a user to a runtime. And finally the fully detailed thread structure of LwLIoTR will be described in section 3.5.

3.1 Internal structure

This section describes full inner workings of LwLIoTR, the general design philosophy behind LwLIoTR and how is LwLIoTR structured and built. Some general guidelines that were used during the development will be discussed when appropriate. Overall structure of LwLIoTR is presented in figure 3.1, where it is possible to see the distinction between general LiquidIoT in figure 2.1 and how it translates to LwLIoTR implementation. Figure 3.1 is explained in detail in section 3.1.1 and in upcoming sections all of these parts are discussed further in detail.

3.1.1 Overall structure of LwIoTR

Overall internal structure of LwIoTR consists of 4 major classes that are *HttpServer*, *HttpClient*, *Application* and *Device*. For each class there is a specific role for it to fulfill. *HttpServer* described in section 3.1.4 handles serving the REST APIs that are described in sections 2.2.5 and 3.4.2. *Application* (Section 3.1.2) will handle controlling the ECMAScript application and communicating between the native classes and ECMAScript classes with the help of Duktape. *HttpClient* class is deeply connected to *Device* class (Section 3.1.3) so that the *Device* spawns *HttpClients* to establish connections with RR.

External communications are handled by *HttpServer* and *HttpClient*. Each time an application is created and RR needs to be updated a new *HttpClient* will be spawned. However,

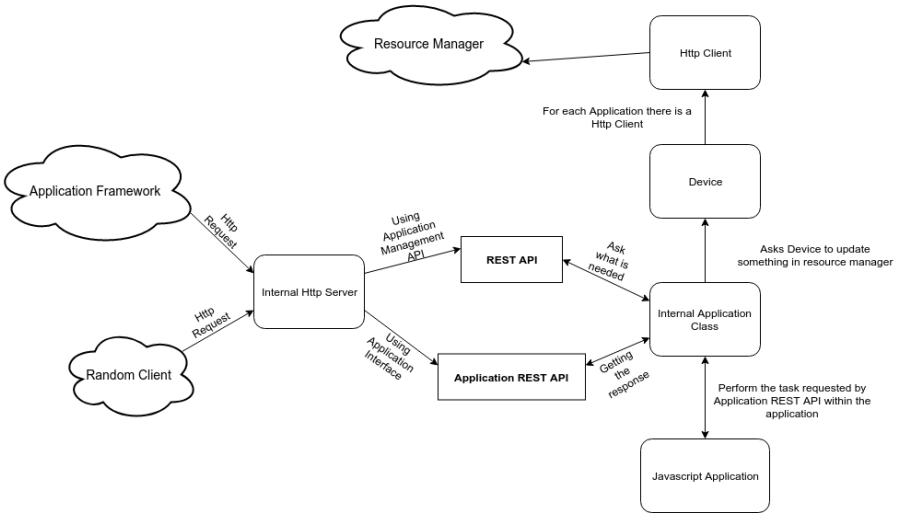

Figure 3.1. *Internal structure of the Lightweight LiquidIoT Runtime*

HttpClient is only spawned once per application and it will remain in memory until the application is destroyed. A thread for the *HttpClient* will be created each time a connection to RR is required. *HttpServer* on the other hand will handle all the incoming connections and *HttpServer* has only one instance running at a time. *HttpServer* thread is spawned when LwLIoTR is started and continues to run until LwLIoTR process is stopped.

3.1.2 Application Class

Application class handles basically everything related to running an ECMAScript application. In addition to running the ECMAScript application code, Application class can be asked to format information about the ECMAScript application in JSON format. Also, Application class sends data to RR when it is needed. For example, AM-API will send data back to the client in JSON format, which was generated by Application class. There are few control functions that can be called, such as *start()* application and *pause()* application. Some static control functions are also present, such as *shutdownApplication()* and *deleteApplication()*. In practice, *pause()* function will pause the event loop. Thus, effectively pausing the application from executing and *start()* will start executing the event loop again. Functions *shutdownApplication()* and *deleteApplication()* are more final as they will remove the application from memory, thus, require a recreation of application in the runtime.

The control flow of the Application Class proceeds as follows:

1. The application reads the application data from the received package file (this is actually a folder of extracted application data and not a tgz file)
2. Application is initialized accordingly by reading the *package.json* file which is roughly the same as Node.js's *package.json* file for applications.

3. With some added source code the *main.js* file is executed on a separate thread and the *initialize()* function will be executed.
4. If everything is acceptable with the application configuration and initial executions, the event loop will be started and the application will start running the *task()* function defined in ECMAScript application.
5. Event loop will execute *task()* function until, it crashes, task was supposed to be executed only once or interrupt signal was detected.

With a few exceptions, anything the ECMAScript application does is handled by the Duktape ECMAScript engine. These exceptions include setting up the *requests* and *responses* for the application. Also any prints applications make will be written to their personal application logs by a native implementation. These logs can be fetched from AM-API in JSON format.

As a summary, Application class simply acts as an interface between ECMAScript application and rest of the classes. Few of the most notable unique interfaces between Application class and ECMAScript application in LwLIoTR are *ResolveResponse* and *RegisterAIPath*. Where *ResloveResponse* handles parsing of the *response* object received from ECMAScript application's Application Interface. And *RegisterAIPath* will simply register the Application Interface path inside the Application class so that HttpServer can check if the path exists in the application.

3.1.3 Device and Http Clients

The job of the Device class is to handle communications with outside world from the actual device. LwLIoTR Device class is a singleton instance. It is designed to be this way, because there is only one instance of Device class at all times and it makes relying on the Device class easier. Decision to make Device class as a singleton in LwLIoTR was the intention of making the overall design little bit clearer and more manageable as there are no questions about how to get access to the Device as it is generally needed by most of the components of LwLIoTR to get different pieces of information related to device. The communication is done by reserving a HttpClient for each of the ECMAScript applications or more accurately for each Application class instance.

The communication happens when anything changes within an Application class, device is then asked to send appropriate information to RR and this information can be as small as application state changes from *running* to *paused* and vice versa. Bigger changes might be managing the application information in RR by creating new applications, updating application information or to simply delete an application. When the device is started it checks if it has already been registered to RR and if that is not the case it registers itself to RR.

As a summary, Device class handles the information delivery from applications to RR using HttpClients as a vehicle. This happens when the application wants to make the

initiative to inform about some change in the application or simply application creation. It is also possible to retrieve information from RR, which is done before registering a Device or registering an application.

3.1.4 Internal Http Server

Internal Http Server (HttpServer) is the focal point in all incoming connections and it handles all the interactions that come from the outside world. In our case this means handling and forwarding requests, that come to both of the REST APIs. In addition to REST APIs, HttpServer serves the description of the AM-API at the root URL.

Internal functionality of HttpServer class is based on the libwebsockets structure of handling requests. Libwebsockets uses callbacks for each connected protocol (HTTP, HTTPS ect.) and currently this means only HTTP protocol is supported for the initial versions of the LwLIoTR. Libwebsockets handles each part of the request one iteration at a time (header, body, completion ect.) in a non blocking event loop. During the request, a write callback function can be called to write to a response, which is often done in several pieces determined by predetermined block size. Once the write is done, the execution will still continue and finish the request. After the request is finished, normal execution of libwebsockets starts again, which will wait for the next HTTP protocol callback in its internal event loop.

How URL paths are recognized is defined in an *urls.h* header, where there is a constant list of definitions to URLs and their callbacks. A definition on the mentioned list consists of a *regular expression*, a *callback function* and a *name*. As the URL paths are defined with regular expressions basically any limitation to URLs can be expressed. Thus, enabling quite flexible URL pathing. Callback function is a mimic of libwebsockets callback function to make the design more consistent. Name is simply to help finding a correct callback if ever needed.

Internally REST APIs are implemented by redirecting each HTTP request to correct class that can handle the request appropriately. These classes are *GetRequest*, *PostRequest*, *DeleteRequest*, *PutRequest* and *AppRequest*. The specialized classes that together create the AM-API are *GetRequest*, *PostRequest*, *DeleteRequest* and *PutRequest* and these classes will handle the request types they have been named after.

Application Interfaces are handled a bit differently compared to AM-API, even though the interface of handling the request remains similar. A request that has the form of Application Interface is handled by *AppRequest* class. This particular request does not distinguish between different HTTP methods. However, it recognizes the method for later usage by the ECMAScript application. *AppRequest* parses the request, supporting GET method's *query strings* and POST method's form parameters. This parsed information is gathered to a bundled ECMAScript class called *Request*, which can be read by the Application Interface function that has been defined by an ECMAScript application. ECMAScript application

handles the data and send a lump of data as a response. And as a last step, HttpServer will handle and send this response forward properly.

Overall the internal HttpServer handles all the network operations that LwLIoTR will receive. Thus, it is relatively easy to configure it to perform more modern tasks such as handling HTTPS protocol. Additionally, HttpServer is not a very resource heavy implementation and is more than acceptable part of the LwLIoTR.

3.1.5 Updating the Resource Registry information

Updating Resource Registry (RR) information is a straightforward operation, which is done several times during the lifetime of an ECMAScript application and Device class. First step, is the device registration to RR. Then, applications will be registered if they do not already exist. When LwLIoTR is shutdown, it will delete applications from RR as a sign that they are unavailable. When application's state changes it will notify RR that its state has changed. Although, RR is not updated in every single instance when the state changes (in cases when application is not ready such as initialization).

Device registration happens when LwLIoTR is started. And the actual procedure goes as follows:

1. Device checks if it exist in RR and if it exists we skip rest of the steps as it is unnecessary to do anything.
2. Delete any previous identification information locally.
3. Send the device information to RR and save the received identification information.

This procedure will ensure that the device will always be registered. However, if for some reason, the id received from RR is lost, the device cannot be removed with RR's REST API.

Application registration procedure logic will be handled by the individual instances of application. The procedure goes as follows:

1. Check if application exists and skip rest of the steps if it does.
2. Register the application to RR.

Deleting application from RR will happen in 2 possible scenarios, when interrupt signal is sent to the runtime or if the application is being deleted or shutdown locally. These 2 cases are the only cases when it makes sense to delete the application from the registry as there is no way the application can be reached from outside after these operations. In case of an error within the application, application will shutdown and delete itself from registry in current version of LwLIoTR. This is a stability measurement and should be handled differently in future versions of LwLIoTR.

Overall the update procedures are pretty simple but necessary part of runtimes communication when considering the whole LiquidIoT system. This is because, AF uses RR to discover devices and applications from the LiquidIoT network.

3.2 LwLIoTR Application Management API

Required functionality of AM-API has already been discussed in a general way in section 2.2.5. In this section we will go through some of the practicalities that LwLIoTR has used to implement this API. The structure will be same as in section 2.2.5, starting from creation of application. Next will be updating application, deleting application and finally application logs.

When creating an application and a package from AF has been received by LwLIoTR, LwLIoTR will store the package temporarily to the filesystem and then, extract the application to "*applications/{appName}/*", where *{appName}* is the name of the received ECMAScript application. After successful extraction the application will be started if no errors occur. In case of any errors during the initialization (file is missing or similar) the application will not be started and application will not be visible anywhere. In case of application error the application will still offer information about application such as logs for debugging the application. In current version of LwLIoTR live logs are not possible if application shutdowns and *logs.txt* has to be read manually.

Updating application is implemented similarly to creation of a new application in LwLIoTR with the exception of shutting down the previous application. The update will override the previous package files and then create a new application instance from the files received. After the update, previous application will disappear from existence when application is updated.

Updating application state from *running* to *paused* will cause LwLIoTR applications event loop to pause and wait for something to wake it up. And when setting the state from *paused* to *running* will cause notify call to occur, which will wake the event loop. After each state change, RR will be updated with latest application information.

Deleting an application causes it to be removed from all the relevant data structures in current runtime. In addition to cleaning the application from the memory, actual files are deleted from the filesystem. These files include the whole application's folder, including logs produced by an application. In the case of deletions of multiple applications, the amount of successful and failed deletions will be sent as a response, which is slightly different when comparing to NodeJS runtime, where only if the request was successful or not is sent.

The information sent about the applications is approximately same as received from AF via *package.json* file, when creating or updating an ECMAScript application. Differences in data might include the application *id*, *author*, *license* and anything that is not included

into the *getDescriptionAsJSON()* function. This information format is hard coded into *getDescriptionAsJSON()* function, which fetches the information from native variables and reformats the data into JSON by using Duktape and finally sends the data back.

Applications will store logs based on prints in ECMAScript, such as *console.log*. *Console*, *print* and *alert* are not part of the ECMAScript standard. However, they are often used in Web-browsers and are included in LwLIoTR to give additional options for printing. For example, *console.err("Error")* will write an error line into the logs in addition to all the console prints in ECMAScript. In addition to console prints, *Print()* and *Alert()* ECMAScript functions will write information to the logs with their respective tags. Logs have a certain format with *"[timestamp] [type] Message"* where **timestamp** is the time when the log was created, **type** is the type of message (Error, Print, ect.) and **Message** is the content text of the log item. These logs can be retrieved in JSON format by using a GET request to "*/app/{application id}/log*". This request will generate JSON formatted logs from the *logs.txt* file that is located in ECMAScript application's root.

In summary, LwLIoTRs AM-API implements the AM-APIs description to the letter with the exception of logs, from which there is not too much information available on how and what format are the logs retrieved by AF. Current version of LwLIoTR does not use any distinct JSON libraries, that would isolate JSON properly. Each JSON object is created and read with the Duktape engine. Thus, there is possibility that LwLIoTRs implementation of AM-API is not currently very flexible to change.

3.3 Duktape as ECMAScript Engine of LwLIoTR

Duktape [4] is the ECMAScript engine of choice in LwLIoTR. This means that the runtime will only fully support E5/E5.1 [5] version of the ECMAScript standard with some small additions from newer standards. In LwLIoTR Duktape is used internally in 2 scenarios, first scenario is reading and executing of source code, such as internal ECMAScript files and applications that are loaded to runtime. Second usage is reading and writing of JSON formatted files and text such as logs and different configurations. Duktape is also used to format JSON responses for AM-API.

The source code of an ECMAScript application is executed by the Application class (Section 3.1.2) as a Duktape implementation of Node.js modules [17]. The implementation of this Node.js compatible module system is from *extras/module-node* package of Duktape and is used to evaluate the source code we have generated from application package. Module-node package requires implementations of *cb_resolve_module*, which is essentially the module search of Node.js and *cb_load_module* to physically load the module to ECMAScript environment. Both of the functions mentioned are ignored when loading the main module with *duk_module_node_peval_main* unless require is used.

When ECMAScript application is executed as a Duktape *node-module*. It will add the Application object to the Duktape environment, from where it will be called later as a

variable when required. Duktape *module-node* is used to enable Node.js style *require* system in applications. It does not usually understand standard Node.js packages unless they are pure E5/E5.1 in which case it is possible to use them if no native Node.js libraries are needed. The raw source code of the applications is kept in Application class for later use if there ever will be a time when re-execution of source code is necessary. However, currently this is not needed.

Task function is a special function that is executed between certain intervals. This is done by creating a custom versions of JavaScript functions *setInterval* and *setTimeout* used in Web-browsers as Duktape doesn't offer them natively. The *task()* function is registered to native *EventLoop* implementation via interface that uses *setInterval* and *setTimeout* functions. The *EventLoop* will then register the *task()* function to Duktape heap where it is possible to be called later.

Errors that are generated by applications are caught by redefined *Duktape.errThrow* function and it will log all the errors that are not fatal (such as syntax errors in applications) to *logs.txt* file to be used later. Fatal errors are similarly stored in logs. However, the application will be shutdown because fatal and often unknown error has happened. Thus, disabling the ability to read logs via the AM-API. Quite often fatal errors will happen if the applications file structure is not conforming with the defined standard on how the application should be built. However, it is more likely that the fatal error is caused by some bug in LwLIoTR itself.

ECMAScript application is executed as a *module-node* from Duktape, thus, enabling some properties from Node.js. It mainly enables the *require* system of Node.js. However, as having fully compliant Node.js *require* system is out of scope of this thesis it will only be discussed minimally when needed and will not be used in performance tests. Implementation of the *require* system in current version of LwLIoTR is very basic and this is one of the main reasons of this decision to leave it out of the measurements. Applications themselves can use the *require* system to extend the functionality of their applications or simply manually add external libraries using *require()*. *Require* function uses the application's root as a base for the path that can be added with *require*. LwLIoTR also has support for native modules that are located in LwLIoTR executables path in *lib/* folder. Path search of *require* works the same way as module search in Node.js with few exceptions related to *index* files, which are not dealt with at all.

There are several instances where Duktape is used to *encode* or *decode* JSON data. Files are the most common usage of JSON *decode* functionality of Duktape. These files include *config.json* for the overall configuration of the runtime and application's own *package.json* file to configure ECMAScript application itself. While *encode* functionality is used to generate responses from Application class and Device. These responses can potentially be description of the application or application logs, which have been discussed in section 3.2.

Overall the Duktape performs all the necessary tasks in a sufficient manner as long as there are no errors in JSON syntax as Duktape seems to fatally fail every time with a syntax error. Normal errors are handled properly, thus, the expected usage is always going to perform as expected. If no later version of ECMAScript syntax is needed, Duktape is more than sufficient in handling all the necessary operations required by LwLIoTR.

3.4 LiquidIoT Application

LiquidIoT application refers to the actual application that is run on the runtime, runtime in this case can be either the NodeJS runtime or LwLIoTR as both of the runtimes should be able to run the applications. There are some minor differences such as what is the supported ECMAScript version and some minor Node.js compatibility issues. LiquidIoT application has specific guidelines on how it should be built. At a basic level a LiquidIoT application is divided into 3 functions *initialization*, *task* and *terminate*, which are all used in different execution stages of the application. These basic functions are also described in "Application development and deployment for IoT devices[2]". In addition to the basic functions that are defined, there is a possibility to define an Application Interface for the LiquidIoT application and do an implementation for it.

3.4.1 LiquidIoT Application Structure

LiquidIoT application consists of 3 functions that are invoked natively by a runtime. This structure makes it simple to design an application as it is guaranteed that, atleast these 3 functions will be executed in correct order and time. Deviating from the basic structure of *initialize()*, *task()* and *terminate()* cycle described in section 2.2.2 is possible, however, not recommended and might result in undefined behavior. The 4th component of the LiquidIoT application is the Application Interface that can be defined by adding the name of the API to *liquidiot.json* file's *applicationInterfaces* JSON array. After configuring the name, it can be used in the application by defining a function that has all the necessary properties and AF will create this structure automatically. A configuration function for the task function can be called by the application to configure if *task* repeats and what is the interval of the repeat. This function is implemented by the runtime.

The *initialize()* function is designed to be executed before anything else is executed. In LwLIoTR this is implemented through a custom *agent.js* file, which is similar to the *agent.js* file provided by AF. The difference to this provided file is the structure of not treating the contents as a node module, but as a pure ECMAScript code. Also, a different implementation of *start()* function is created. This function is attached at the end of the LiquidIoT application source code to start and initialize the application. After the combined source code is created, it is executed as described in section 3.3. It is also possible to rely on the *agent.js* file provided by AF, however, this approach needs the full support of the *require* system and some additional adjustments to native code to be able properly execute the *start* function. The purpose of the *initialize()* function is to give a

clear place to initialize the application to a state it can be run without problems. However, the function is not designed to initialize any other necessary initializations such as setting up the repetitions for the *task()* function. As a conclusion for the *initialize* function, it is a function that is literally designed to initialize the application for the task.

The *task()* function is designed to be executed periodically at certain interval of time or just once after a certain period of time (which can be 0). In LwLIoTR this execution is bound to an event loop, which is initialized at the same time as when the application is initialized. The *task()* function is designed to conform with the overall design principle of the ECMAScript application and internally *setInterval* and *setTimeout* functions have been implemented and used in the ECMAScript application. These functions allow bypassing the original design principle, however, this is not recommended because of the undefined behavior.

Last mandatory function is *terminate()* and it only exists to clean up the application when it will finish its execution if necessary. In LwLIoTR the function itself is called after everything else has finished, so it is guaranteed to be executed as a last function execution in ECMAScript application. This also means that the application class will terminate itself right after calling the terminate function.

Overall the ECMAScript application's structure is very stable as long as the user does not deviate from the standard that has been established. In figure 3.2 we can see the paradigm picture of LiquidIoT application on how the application is supposed to be executed in a standard runtime environment.

3.4.2 Application Interface

ECMAScript application's REST APIs (Application Interface) are not part of the execution cycle that is happening with rest of the application. However, Application Interfaces can give information on how the execution of the application is performing. Or they can even influence the execution cycle of an application by changing the variables that are present. Thus, enabling more control over the applications even after the deployment. In LwLIoTR, design of the Application Interface is quite simple. LwLIoTR Application Interface allows different request types and passing of simple parameters in string form. The response from the Application Interface can be a bit more elaborate and basically it can send anything the ECMAScript can produce back in a string (HTML, JSON, ect.).

In LwLIoTR Application Interfaces are defined by adding an *applicationInterface* parameter into *liquidiot.json* file only to define the name of the Application Interface. After defining the name, it is possible to write a function and pass that function to *Router* class with the url path information. *Router* class is provided by a runtime.

In LwLIoTR ECMAScript *Request* class consists of basic parts that can be defined. These parts are request-type, headers and parameters. Request-type can only be one of the

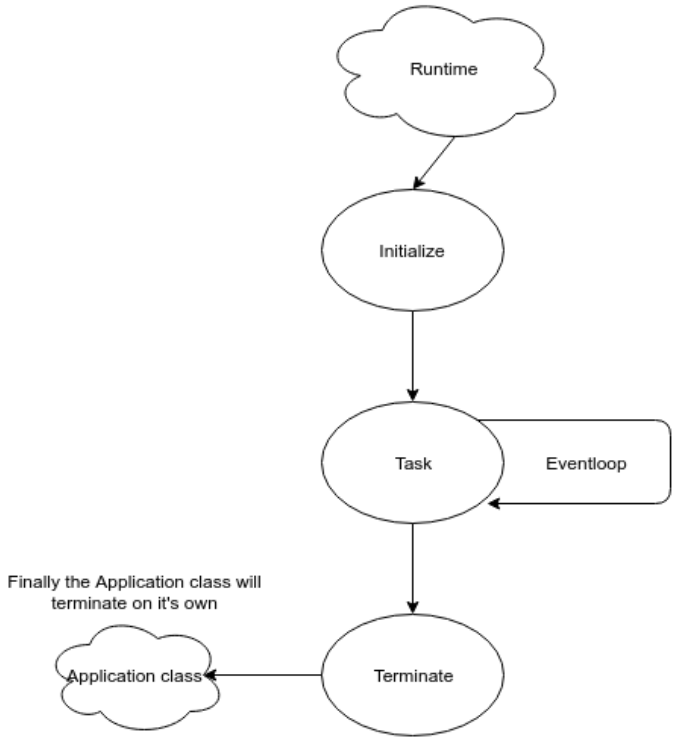

Figure 3.2. *Paradigm of LiquidIoT application*

following types GET, PUT, POST and DELETE. These types are recognized by LwLIoTR, additional restrictions to this, are parameters where only GET and POST parameters are read from the request. POST parameters are read with a regular expression that tries to parse the parameters from the request. This in mind, it is recommended to only use basic parameters in requests (string and number parameters).

In LwLIoTR ECMAScript *Response* class, it is possible to set a lot of necessary parameters, as the response object is mostly complying with Node.js's *response* object, excluding the *response.write* and *response.end* functions. Although, some rare or unnecessary attributes might be missing from the *response* implementation. The usual usage of the response will go like this:

1. Setup headers and the status code
2. Write the response with *response.write()* as raw data
3. End the response with *response.end()* to evaluate it

The *response.write()* will simply take the raw data as a string and pass it as a payload when *response.end()* is called.

As a summary of Application Interface, it is a simple implementation of some of the common functionalities of REST APIs. There are no clear limits to what the interface can do. However, there is a limitation that you cannot read only parts of the request object like you can in Node.js, instead LwLIoTR does this automatically and restricts some of the

control over what is read. Responses are more flexible as they can basically send any kind of data through headers and a raw message body.

3.5 Threading

LwLIoTR uses threads in a simple and concise manner. There are always atleast two threads running at the same time even if no LiquidIoT applications are running at that moment. These two threads are the main thread that every other thread is running on and a secondary thread for the HttpServer that has been described in section 3.1.4. If a LiquidIoT application is created a new thread will also be created when applications event loop is started, also a short lived thread is spawned for instantiating an application. The application thread is destroyed if the event loop ends for some reason. During the initialization of application a separate thread to instantiate the Duktape evaluation of LiquidIoT application is started, for performance reasons during a HTTP request. The final way that new threads can and will be created, is by communicating with the RR as the application will need a HttpClient to send the data to the RR, see section 3.1.5 for all the scenarios when a new thread will be created when forming a connection to RR. The cross usage of resources within the ECMAScript application is protected by three types of mutexes in Application class and a single mutex in Device class.

Main thread has a simple task of holding and controlling other threads on a high level. This means that the execution and join order is determined by the main thread. In our case the main thread first starts all the ECMAScript applications and then it starts the HttpServer thread. Then the main application waits for the HttpServer to finish its execution and then wait for applications to finish executing. In practice this execution flow will break when applications are created and deleted on the fly because applications handle breaking out of the event loop on their own and interrupt signal (SIGINT) is currently the only way to safely stop LwLIoTR execution.

HttpServer thread is a dedicated thread for the HttpServer class. Its job is to make sure HTTP connections do not disturb the flow of other functions in LwLIoTR and do not block any fundamental parts of the runtime. Also separating the HttpServer as its own thread means that it can actually handle any HTTP requests immediately. Only interrupt signal will terminate the HttpServer, thus keeping the runtime alive indefinitely if desired so.

Application receives its own thread when an event loop is started. An instantiation thread will be started to make the initialization to look faster as some LiquidIoT applications will take some time to initialize and when AF will send an application package it will expect the response to be relatively fast. The instantiation thread will then start the event loop when the LiquidIoT application has finished loading into memory. Event loop will end after the first execution of the provided function if the repeat parameter of the application is set false. The event loop thread itself will also end in two different scenarios, if interrupt signal is sent to the runtime or if some error has occurred and exit is requested by the application itself.

HttpClient threads are spawned quite frequently for each outgoing connection to RR. However, there can only be one HttpClient thread for each ECMAScript application at one time. This arrangement of only one HttpClient per ECMAScript application improves the ability to maintain thread secure environment overall.

Mutexes in LwLIoTR are used to control the flow of threads overall and making sure no thread will be doing dangerous operations on top other threads. The design principle behind using mutexes in LwLIoTR is relatively simple but effective in doing what they were meant to be doing. Application itself has its own mutex that is used every time a class variable is being modified or blocking an initialization function that requires everything to stay the same at all times. Application's own mutex is used within Application class in situations when it is dangerous to let other threads interfere with the execution and is most often used to block member variables when they are being changed.

Application class also has a secondary instance mutex that is bound to a Duktape context and is used every time the Duktape context is being modified making it easier to create atomic operations with Duktape engine. Duktape mutex is making sure that Duktape heap is not touched by anything else during the executions and would probably be necessary even if no other applications were present in current implementation, as HttpRequest classes from HttpServer use Application class's JSON conversion functions, which are based on Duktapes *duk_encode_json* and *duk_decode_json* functions.

Application class has a third mutex for static variables and static operations. This mutex will make sure that all static variables and data structures are protected by this mutex. This is one of the most important protections in principle, as the static data structures control all the applications and when they are touched it is good to know nothing else is doing anything to them.

Overall mutex control flow design principle, ensures that each operation has a direct meaning and is protected by correct mutex. In some cases to have maximum protection all mutexes can be locked, as is done in Application class's initialization function. This flow enables a relatively free usage of variables and functions as long as the design principles are followed.

As a summary of the thread structure, it is relatively simple but effective way to gain access to multithreading and its upsides. With a clear design principle behind the usage of threads and mutexes eliminates most of the downsides of multithreading. But as always, design principles are only guidelines and human errors will occur in the end. Although, bugs related to multithreading have not been found after applying these design principles in LwLIoTR.

4. PHYSICAL ENVIRONMENT FOR LWLIOTR

This chapter contains most of the hardware and software requirements on how to run LwLIoTR. In addition to these technical requirements, requirements for the measurements are discussed in detail. Target environments that LwLIoTR has been designed to be ran on, will be discussed in section 4.1. Then we move to technical requirements in section 4.2, that are mostly gained from the measurements as well as some general observations. Environments that the measurements are ran on, are described in section 4.3. Section 4.4 will describe software and their versions used in measurements. And finally descriptions of measurement tools will be in section 4.5.

4.1 Target environment for the Runtime

LwLIoTR is designed to be a fully cross platform runtime. However, there are a few relatively new libraries in use such as C++11 and Boost library version 1.61. These libraries might limit some of the systems that LwLIoTR can be used in. The main goal of LwLIoTR is to be a low memory runtime environment, in which it succeeds sufficiently as we will see in section 5.3.

Cross platform runtime in this case means that LwLIoTR should compile to any platform that can support the underlying libraries that LwLIoTR uses. LwLIoTR requires the libraries that have been discussed earlier, libwebsockets and libarchive on top of the mentioned libraries C++11 and Boost. Boost library version 1.61 is required for its *filesystem* module, which could be replaced with C++17 and its *filesystem* module. However, the 1.61 version of Boost is specifically required for the ability to find out where is the real executable is located, with Boost *dll* header library. In addition to these functional libraries a relatively new version of C++ standard is required (C++11). C++11 is used quite a bit with *thread* library and everything related to threads. In addition to threads, the *chrono* library is used to calculate time in different scenarios. Duktape ECMAScript engine is very portable and it is very unlikely that there would be any problems with Duktape and system compatibility if every other dependency is met.

Before the measurements, LwLIoTR was tested on few Linux platforms without any compatibility issues, when required dependencies are met. However, this does not mean that there will not be any problems when moving away from modern Linux platforms to other types of platforms, thus, it is currently recommended to use a modern Linux distribution as main type of platform with the LwLIoTR. Although, the recommended platform is Linux there should not be many problems with other types of systems although they are not tested.

As a summary, the design principle of using standard libraries and trying to minimize the amount of required external libraries, it is not unreasonable to expect LwLIoTR to work in most major environments. And the relatively small memory consumption can open up some new possibilities regarding on what kind of environment LwLIoTR can be placed on.

4.2 Technical Requirements of the Runtime

In this section we will go through some of the results gained from the measurements in chapter 5 without trying to analyze them too much. This is an attempt to gain some insight on what would be the minimal requirements for a device to run LwLIoTR on.

Modified PSS memory consumption described in section 5.2.2 is used to evaluate memory consumption and memory measurement and values are taken from table 5.3. Without any ECMAScript applications the base memory consumption is approximately $2932KiB$ on a 64 bit Laptop running ArchLinux and $2284KiB$ on a Raspberry PI. This would make the minimal memory consumption of the runtime to be approximately in $2-4MiB$ range. If we look at Basic application's (Section 5.1.1) memory consumption it will increase the usage to $4272KiB$ and $3072KiB$ respectively, making the consumption to be between $3-5MiB$ which might even be the lower limits on how much memory the runtime will use in an expected scenario. For a small comparison using Basic application, NodeJS runtime uses $56836KiB$ and $40768KiB$ of memory on their respective environments.

In theory, the V8 engine should have an edge on the performance department as it is designed to be fast, where as Duktape has been designed to save as much memory as possible. The actual performance difference of ECMAScript applications between NodeJS runtime and LwLIoTR is almost solely determined by the ECMAScript engine running the actual applications and functions. In this case it will be a comparison between V8 and Duktape ECMAScript engines. As we can see in table 5.4 the NodeJS runtime has significant advantage on performance department and even on Raspberry PI the NodeJS runtime will have an edge on performance over LwLIoTR running on the Laptop.

Sometimes it is necessary to look at the RSS memory consumption as it is technically the amount of memory a process uses on a Linux system. Even when using RSS the memory consumption with Laptop and a Basic application is $10372KiB$ and $7948KiB$ on Raspberry PI. Which is still acceptable amount of memory as NodeJS runtime still uses approximately 5 times more memory.

As a summary, the minimal requirements in memory department on actually running LwLIoTR are acceptable with clearly under $16MiB$ usage of memory. However, if some processing heavy application is used on LWLIoTR it is possible that it uses quite a bit of processing power for an extended period of time when Duktape evaluates the ECMAScript code, which might give some unforeseen problems.

4.3 Physical Measurement Environments

The measurements are performed in two types of environments. The first one is a consumer grade PC (Laptop), where there are no practical limits to environments performance, as it is not limited by lack of memory and has sufficient amount of processing power to compute almost anything related to the measurements. Second environment is a slightly more limited Raspberry PI platform with specifications that are low enough so that they might matter in measurements. As the original goal was to create a runtime environment that can perform with limited resources (mostly memory). Raspberry PI is still too powerful to properly measure these very low resource environments, so the data that is gained from the measurements is analyzed to see how the performance correlates between the two environments and extrapolate from these results.

The general procedure described in section 5.2.1 is done in both of the environments. However, the measurements done with Laptop are not the most important measurements as both of the runtimes are designed to be used in embedded devices. That said, if the results are similar in both environments it reduces the doubt if the LwLIoTR can be used in even more restricted devices.

The specifications of Laptop are described in table 4.1, as we can see RAM memory is more than enough to perform any measurement we can think of. As for the Raspberry PI, memory consumption should be taken into accord when defining how large allocation is done in Memory consuming application described in section 5.1.2. We can conclude here that the measurement environments are sufficiently different, that the generalization of results, that are gained from the measurements will be good enough to draw some conclusions from.

Table 4.1. *Measurement environment descriptions*

Device	Processor	RAM
Laptop	4 x Intel Core i5-3317U CPU @ 1.7GHz	7.7 GiB
Raspberry PI	700 MHz ARM1176JZF-S	512 MB

4.4 Software used in Measurements

In previous chapters and sections we have discussed about the general properties of runtimes and how do they conceptually work. This, however, is not enough to define how the exact results are obtained and in this section we will concentrate on the software versions and individual properties of runtimes, that are used in measurements. LwLIoTR uses the *0.2.1* release version of LwLIoTR, it has acceptable support for AF and is stable enough to be used and run measurements with. On the other hand, we have the NodeJS runtime, where we have to define the used version to be a shorthand of git commit hash as there is no official versioning for that runtime. All the information about the versions that are used in measurements are listed in table 4.2.

Table 4.2. *Software versions of runtimes*

Software Config	**Runtime**	**Software name**	**Software version**	**Node.js version**
Lw-runtime	LwLIoTR	liquidiot-duktape-server	0.2.1	
Node-runtime-10	NodeJS Runtime	liquidiot-server	03fb0f6	10.9.0
Node-runtime-4	NodeJS Runtime	liquidiot-server	03fb0f6	4.2.1

LwLIoTR version *0.2.1* has approximately the same interface support with AF as the NodeJS runtime, which has been discussed in section 3.2. As LwLIoTR is based on Duktape and its version *2.2.1*, this means that the version *0.2.1* of LwLIoTR is limited to ECMAScript E5/E5.1 and some of the interfaces are relatively primitive compared to their Node.js counterparts as they have only been designed to work as interfaces to mimic Node.js's behavior.

The difference between Node.js versions in *Sofware Config* is only relevant if the performance between versions would differ significantly. Performance wise this might the case as the underlying V8 architecture has changed after the release of Node.js 8.0.0 this was discussed in section 2.4. However, there are no reports of very significant memory improvements between versions. As an upside of using different versions of Node.js, it is possible to get little bit more information about its effects in NodeJS runtime as well as get a more general picture, on how LwLIoTR places in performance.

4.5 Measurement Tools

The tools that are used in measurements are *pmap* for getting general information about RSS memory usage and PSS memory usage, which are described in section 2.6. *Massif* is a tool in valgrind framework which can measure the internal heap usage of a process during the lifetime of a process. *Massif* is used to indicate the distribution of memory within the LwLIoTR. It is not possible to use *massif* to measure NodeJS runtime successfully, as the results gained are not even close to what they are expected to be, eg. basic application (Section 5.1.1) and memory consuming application (Section 5.1.2) have approximately the same heap size when trying to measure with *massif*. Because of this, only some light comparisons are made with NodeJS runtime and LwLIoTR, which are done in section 5.3.3. The list of tools are composed to table 4.3.

Table 4.3. *Tools used in measurements*

Tool	**Description**	**Runtime measured**
pmap	Reports a memory map of a process	Both
valgrind	Valgrind is an instrumentation framework for building dynamic analysis tools.	
massif	General heap memory profiler	LwLIoTR

Pmap can show the whole memory map of a process including all the libraries that are used by a process. However, it can only create one snapshot of the whole memory structure at a time. Thus, we should take that into account when analyzing the results. The most

```
Address           Kbytes     RSS   Dirty Mode  Mapping
0000000000400000     952     936       0 r-x-- liquid-server
0000000000400000       0       0       0 r-x-- liquid-server
00000000006ed000       4       4       4 r---- liquid-server
00000000006ed000       0       0       0 r---- liquid-server
00000000006ee000       8       8       8 rw--- liquid-server
00000000006ee000       0       0       0 rw--- liquid-server
00000000006f0000       4       4       4 rw---   [ anon ]
00000000006f0000       0       0       0 rw---   [ anon ]
000000000157a000     456     300     300 rw---   [ anon ]
000000000157a000       0       0       0 rw---   [ anon ]
00007f43f0000000     164     164     164 rw---   [ anon ]
00007f43f0000000       0       0       0 rw---   [ anon ]
00007f43f0029000   65372       0       0 -----   [ anon ]
00007f43f0029000       0       0       0 -----   [ anon ]
00007f43f4000000     132     128     128 rw---   [ anon ]
00007f43f4000000       0       0       0 rw---   [ anon ]
00007f43f4021000   65404       0       0 -----   [ anon ]
00007f43f4021000       0       0       0 -----   [ anon ]
00007f43fb7ff000       4       0       0 -----   [ anon ]
00007f43fb7ff000       0       0       0 -----   [ anon ]
00007f43fb800000    8192      16      16 rw---   [ anon ]
00007f43fb800000       0       0       0 rw---   [ anon ]
00007f43fc000000     172     172     172 rw---   [ anon ]
00007f43fc000000       0       0       0 rw---   [ anon ]
00007f43fc02b000   65364       0       0 -----   [ anon ]
00007f43fc02b000       0       0       0 -----   [ anon ]
00007f440020f000       4       0       0 -----   [ anon ]
00007f440020f000       0       0       0 -----   [ anon ]
00007f4400210000    8192      16      16 rw---   [ anon ]
00007f4400210000       0       0       0 rw---   [ anon ]
00007f4400a10000       4       0       0 -----   [ anon ]
00007f4400a10000       0       0       0 -----   [ anon ]
00007f4400a11000    8192       8       8 rw---   [ anon ]
00007f4400a11000       0       0       0 rw---   [ anon ]
00007f4401211000      44      44       0 r-x-- libnss_files-2.23.so
```

Figure 4.1. *Example pmap output on LwLIoTR*

important fields that *pmap* shows are RSS, Dirty (or PSS) and Mapping, example of this is shown at the top of figure 4.1. RSS and PSS have been discussed in section 2.6. Mapping can define the file backing the map, these are usually the libraries that runtime will use. Mapping can also point to *[anon]* for allocated memory by the process itself or *[stack]* for the program stack. Mapping to runtime itself will practically refer to the size of the runtime executable. In LwLIoTR, libwebsockets and libarchive are separated as external libraries. The procedure on how the *pmap* is exactly used is described in section 5.2.2. There is an example output of *pmap* in figure 4.1.

Valgrind tool *massif* is a heap profiler. It measures how much heap memory does a process use during the execution lifespan of a process. It will "profile" the memory of the process and it is possible to look and analyze local memory usage throughout the execution of a process. It is called a memory profiler for this exact reason. An external program called *massif-visualizer* [15] is used to generate memory heap graphs from the data generated by *massif*. *Massif* is used to get a clear picture on how much memory does Duktape use in comparison to other parts of LwLIoTR.

Pmap and *massif* together are good enough tools for measuring the memory consumption of native applications from different angles. Node.js applications cannot reliably be

profiled with *massif*. Thus, *pmap* and some internal Node.js functions are used to generate a sufficient picture of memory usage of NodeJS runtime, this process is explained in detail in sections 5.2.2 and 5.2.4.

5. PERFORMANCE EVALUATION OF RUNTIMES

This chapter will go over all the results gathered from using the procedures described in section 5.2 and its sub sections. The structure of this chapter will be following. First, ECMAScript applications that are used are described in section 5.1 and its subsections. Then, methods and procedures that are used to measure the performance of LwLIoTR and NodeJS runtime are described in section 5.2. Then, obtained results are fully described and presented in section 5.3. And finally, analysis of the results is in section 5.4.

5.1 Applications used in measurements

In order to get comparable results on memory consumption and performance speed, runtimes should be measured from several different angles. Three different ECMAScript applications put runtimes to their limits, so it is possible to see how runtimes perform. Basic application in section 5.1.1 is simply designed to test the basic functionalities of an ECMAScript application. Memory consuming application in section 5.1.2 tries to allocate significant chunk of memory, so it is possible to see how both of the runtimes handle allocation of large data structures. Performance heavy application in section 5.1.3 executes performance heavy code and measures its time consumption. All three types of applications measure some aspect of the runtimes performance and their underlying ECMAScript engines performance, mostly the engines. All of the applications have been designed, programmed and deployed with AF that we have introduced in section 2.2.2.

5.1.1 Basic application

The idea behind the Basic application is to have a baseline on what to measure against as well as to get a good understanding on how each of the runtimes performs when using a simple and relatively normal application. If *process* object is defined in ECMAScript application, memory information for heap memory measurements (Section 5.2.4) is calculated and printed. This measurement is designed to use as little memory as possible. Functionally Basic application increases a counter in each iteration of the *task()* function. It also has an Application Interface that reveals the counter information when using a GET request to the defined Application Interface. Full source code of Basic Application is in Appendix A.

Application starts by initializing its local *counter* variable and initialization of *task()* function to be repeatable on 3 seconds interval. In *initialize()* function, *counter* is reinitialized and although the initialization of *counter* is not entirely necessary it will give some consistency for the design. In addition to this, the value of *counter* is printed to *console.log* in the

initialize() function. In LwIoTR *console.log* will write to log file and in NodeJS runtime an actual console is used. The task that is executed in *task()* function is very simple. It will simply increase the counter by 1 and print its value to *console.log*.

This is the only application where Application Interface feature is defined in measurement applications. Application Interface itself is simple and generally it will simply receive a request and respond with the *counter* value. To be more precise, it will create a new object to represent the *counter* value and then format the new variable with *JSON.stryingify()* function while calling the *Response* class's *write()* function. As a final step, *Response* class's *end()* function will be called, which will finish the response. In LwLIoTR this is the exact usage that will work and any deviation from this style will most likely not work properly on LwLIoTR.

As a summary of the Basic application, it uses almost all the basic properties of the LwLIoTR, excluding POST request as it is not straightforward to replicate measurements when direct user influence is required. NodeJS runtime has the advantage of Node.js framework and LwLIoTR has tried to adapt to the original design. However, the Basic application itself has been designed to be as compatible as possible with LwLIoTR and it happens to also work on the original NodeJS runtime.

5.1.2 Memory consuming application

The idea behind Memory consuming application is to measure if either of the runtimes use more memory for each allocation and to make sure that there are no significant differences between runtimes when memory allocation is concerned. This was the expectation before measurements. Basic idea of the application is to dynamically allocate memory multiple times with a small data package, which is a small string. These small allocations should prevent any garbage collection to activate for the array during the measurements. Full source code of Memory consuming application is in Appendix B.

To control how many times memory will be allocated, a variable *max_reserve* is defined, which will indicate the amount of additions to the allocated array. Allocation itself is done in *initialize()* function. A *for loop* pushes strings to an ECMAScript array the amount of times *max_reserve* variable is set to, in case of measurements this value is $5 * 10^6$. This process will most likely trigger ECMAScript engine's memory allocation procedure multiple times and will ensure that something is allocated into physical memory by ECMAScript engine. The actual string that is used is "Some random text".

As a summary of Memory consuming application, it gives a clear picture on how efficiently each of the runtimes (or ECMAScript engines) will allocate memory. Also it will give a clue, if there is any significant overhead when allocating memory in either of the runtimes.

5.1.3 Performance heavy application

The idea of the Performance heavy application is to consume as much time as it is reasonable for the measurements to be reliable enough. This performance heavy operation is done multiple times in *task()* function to get enough data by running the operation multiple times. Operation in its simplicity is the *Array* class's *reverse()* function ran multiple times. Full source code of Performance heavy application is in Appendix C.

The *task()* function is executed once every second, so there would be enough time to recover and stabilize from the previous execution of the operation and make the measurements in a reasonable time. The execution time is measured with 2 different functions, as Node.js does not support the *performance* interface [19], that gives access to performance-related information. LwLIoTR uses *performance.now()* function to measure the time in milliseconds and NodeJS runtime uses *process.hrtime()* function to measure the time in nanoseconds, which is later converted to milliseconds to print out the result.

Measurement operation starts by allocating a small array and assigning numbers to it. Then the array is reversed multiple times in a row to achieve artificial calculation. *Console.log* is not included in time measurement for fairness purposes as LwLIoTR uses file I/O operations of the operating system for that particular command and that might distort the results in a manner that is not relevant.

When performance measurement task has finished, 5 different values will be updated. These values are, maximum execution time, minimum execution time, mean execution time, variance of measurements and confidence interval of measurements. If a measurement fails for some reason, for example time calculated is negative a mean of previous measurements will be used.

The overall performance is expected to favor NodeJS runtime as Node.js uses Google's V8 ECMAScript engine, which has been optimized for performance overall. However, the application gives a good baseline on how much is the actual difference between runtimes performances.

5.2 Methods of Measurement

There are three different procedures used to measure both of the runtimes. General Measurement Procedures in section 5.2.1 describe the measurements in a more general way in order to reduce repetition and give a more general understanding, on how procedures are designed. In each procedures respective sections, definite procedures are defined for each measurement. Procedures for Memory Measurement in section 5.2.2 refers to procedure of getting a single *pmap* snapshot of a runtime and processing it. Measurement Procedure for Performance Speed in section 5.2.3 refers to the raw execution of Performance heavy application described in section 5.1.3 and then recording the results obtained from the executions. Measurement Procedure for Heap Memory in section 5.2.4 describes the

procedure of getting heap memory results for both of the runtimes. Tools and their intrigues that are used in these measurement procedures were discussed in section 4.5.

5.2.1 General Measurement Procedures

This section describes how the measurements are performed from a practical point of view. Before the measurements, an instance of RR and an instance of AF should be running in a different location than runtime. Reason for this separation is to have only one instance to Node running to be certain that correct Node.js application is measured. AF is used manage the installation and removal of applications during the measurements. Only one measurement application is ran at a time and runtime is restarted after each application measured. In the actual measurements, RR and AF were located in the same computer. After these initial steps, measurements can be made for each runtime and environment. The exact procedures for each measurement category are defined in sections 5.2.2, 5.2.3 and 5.2.4.

In measurements, there exists 16 different configurations for the environments, which are listed in table 5.1. Hardwares in table 5.1 are introduced in table 4.1 and Softwares are introduced in table 4.2. One major difference between the configurations is the Node.js version difference between different hardware environments. However, as is seen in the actual measurement results, it will make a difference, but is not a major factor. **Measurement vector** is used to indicate a set of measurements bound to a measurement configuration.

Table 5.1. *Measurement configurations*

Configuration	Hardware	Software	Application
Lap-LwLIoTR-none	Laptop	Lw-runtime	None
Lap-Node-none	Laptop	Node-runtime-10	None
Rasp-LwLIoTR-none	Raspberry PI	Lw-runtime	None
Rasp-Node-none	Raspberry PI	Node-runtime-4	None
Lap-LwLIoTR-basic	Laptop	Lw-runtime	Basic application
Lap-Node-basic	Laptop	Node-runtime-10	Basic application
Rasp-LwLIoTR-basic	Raspberry PI	Lw-runtime	Basic application
Rasp-Node-basic	Raspberry PI	Node-runtime-4	Basic application
Lap-LwLIoTR-memory	Laptop	Lw-runtime	Memory heavy application
Lap-Node-memory	Laptop	Node-runtime-10	Memory heavy application
Rasp-LwLIoTR-memory	Raspberry PI	Lw-runtime	Memory heavy application
Rasp-Node-memory	Raspberry PI	Node-runtime-4	Memory heavy application
Lap-LwLIoTR-perf	Laptop	Lw-runtime	Performance heavy application
Lap-Node-perf	Laptop	Node-runtime-10	Performance heavy application
Rasp-LwLIoTR-perf	Raspberry PI	Lw-runtime	Performance heavy application
Rasp-Node-perf	Raspberry PI	Node-runtime-4	Performance heavy application

Memory measurement procedure is done 4 times to each runtime on each environment. ECMAScript applications that are used in Memory Measurements are *no application*, *Basic application*, *Memory heavy application* and *Performance heavy application*. The procedure used to measure these applications is described in section 5.2.2 and it is done to each application once. This means that there will be 16 measurement vectors in total. This

means that, there are $16 * 8 = 128$ different measurement values where 8 is the size of the measurement vector and 16 is the number of configurations (or measurement vectors). This way, a realistic picture on how runtimes behave memory wise is gained. *No application* gives a basic understanding on how the runtime consumes memory by default. *Basic application* gives the memory consumption, that we should be expecting on an average use of the runtime. *Memory heavy application* forcefully uses a lot of memory and a picture on how would the runtimes scale is gained, when applications get a bit more memory heavy. *Performance heavy application* is used in this measurement to give a second baseline in addition to *Basic application*, as memory consumptions should be fairly similar.

Performance measurements are done with the *Performance heavy application* for each environment configuration. Practically this means, that there will be 4 measurement vectors in total for the *Performance heavy application*. *Performance heavy application* makes the runtime spend some time computing and it will evaluate the performance speed and print it in milliseconds. This will also give some insight on the difference between Duktape and V8 ECMAScript engines.

When measuring heap memory only applications *no application*, *Basic application* and *Memory heavy application* are used to measure heap sizes. The environment that is used for the measurements is Laptop. There are 6 different measurement vectors. However, these measurement vectors are not thoroughly compared with each other as there are some problems with integrity of the procedure, which is discussed in greater detail in section 5.2.4. *No application* will give heap memory's baseline, which can be compared against and to analyze how memory is used when an ECMAScript application is loaded. *Basic application* will give heap consumption in an expected situation. *Memory heavy application* will give infomation on how the heap scales when a lot of memory is consumed by an ECMAScript application.

As a conclusion of the overall measurement procedures, is that the measurement procedures try to ensure as fair of a comparison between NodeJS runtime and LwLIoTR as possible. Performance heavy application can be used as a sanity check for the basic applications results in memory consumptions as they should have roughly similar memory consumptions. In Heap memory procedure, *Basic application* gives a sufficient baseline on how the runtimes behave. Performance measurement procedure is basically there to verify, that V8 is more powerful engine performance wise.

5.2.2 Procedures for Memory Measurement

When measuring memory consumption of runtimes, *pmap* is used to generate snapshots of runtimes memory map at one time instance. This allows the calculation of estimated memory consumption overall. The tools used in these measurements are described in section 4.5. The method to calculate and store the memory measurements for the LwLIoTR is as follows:

1. On a Linux command line "*pmap* -x PID" (where PID is the process id in Linux system and -x is argument for getting the extended information, that is needed). This prints memory information in significant detail about the application at that time.
2. Record total RSS and Dirty (PSS) memory.
3. Calculate RSS of executable (name of the process), [anon] and [stack] mappings together and record the result.
4. Calculate PSS of executable (usually 0), [anon] and [stack] mappings together and record the result.
5. Find libwebsockets and libarchive mappings. Calculate their RSS and record the result.
6. Calculate libwebsockets and libarchive mappings PSS and record the result.
7. Calculate every other libraries PSS mappings together and record the result.
8. Sum items 3, 5 and 7 together to get **modified PSS**.

Logic behind the *modified PSS* composition is an attempt to get maximum memory usage, while not including maximum sizes of common libraries. Thus, from item 3 it's possible to get the size information about the application's internal memory usage and RSS version of this calculation is used to include the size of the executable. Item 5 gets the reserved size to rarely used libraries, and finally at 7 adding the shared usage of common libraries. The result gained from item 8 slightly overestimates the memory usage. However, not significantly and it gives a better understanding of LwLIoTRs memory consumption when there are not many processes running on the same system, which is often the case with embedded systems.

The process of using *pmap* for the NodeJS version of the runtime is similar to the one described for LwLIoTR. The process is as follows:

1. Use "*node manager_server.js*" to start the node process and make sure no other node processes are running at the same time.
2. On a Linux command line "*pmap* -x PID" (where PID is the process id of the node process).
3. Record total RSS and Dirty (PSS) memory.
4. Calculate RSS of executable (name of application), [anon] and [stack] mappings together and record the result.
5. Calculate PSS of executable (usually 0), [anon] and [stack] mappings together and record the result.
6. Calculate all the libraries mappings PSS together and record the result.
7. Sum items 4 and 6 together to get the **modified PSS**.

Logic behind the *modified PSS* composition is an attempt to get maximum memory usage, while not including maximum sizes of common libraries. Thus, from item 4 it's possible to get the size information about the application's internal memory usage and RSS version of this calculation is used to include the size of the executable. Item 6 adds the shared usage of common libraries. Differences in procedures when calculating the memory values are not big between runtimes. Basically the only thing that really is different in calculations is the calculation of libwebsockets and libarchive libraries, which are taken into account in LwLIoTR because of the rarity of the libraries.

When using *pmap*, it reveals that both of the runtimes use a lot of the same external libraries, which is relatively surprising given that both of the runtimes have completely different base idea behind the architectures. NodeJS runtime is based on Node.js, where as LwLIoTR is designed to be a native C/C++ application. On the other hand, this makes sense as Node.js uses a lot of common libraries and is designed to be portable between platforms and LwLIoTR is also designed to be relatively portable even though it has not been the main focus.

Values that are gathered from the measurements are, total RSS of both of the runtimes, PSS of the runtimes and other memory usages that are gathered by using the procedures described. All of the memory measurements are done with each application to gather enough data to draw relevant conclusions.

5.2.3 Measurement Procedure for Performance Speed

There is only one procedure that is used to measure the speed of execution of runtimes. How the measurements themselves are taken, is described in section 5.1.3. However, the overall concept is to measure the execution time sufficient amount of times, so it is possible to be confident that performance measurements are valid for the system that the runtime is running on. The procedure goes as follows:

1. Deploy the *Performance heavy application* to runtime.
2. Wait for the *iterations* counter to reach 100.
3. Record the values gained at iteration 100. These values are maximum time, minimum time, mean time, confidence interval and variance.

For the actual measurements, a tool from statistics called confidence interval is used to make sure that we have a good chance of getting reasonable results from the performance measurements. A z value can be calculated with inverse of cumulative distribution function for a random variable $F^{-1}(p) = \mu + \sigma\Phi^{-1}(p) = \mu + \sigma\sqrt{2}erf^{-1}(2p-1)$, where μ is mean of the distribution, σ is the standard deviation of the distribution and erf^{-1} is the inverse of mathematical error function [21]. In the measurements, a 0.95 probability that measurements land into the confidence interval is desired. Thus, $p = 0.975$, which is measured from both sides of the distribution. As only the assurance that the measurement

procedure is working in a reasonable manner a cumulative normal distribution is used, thus, $\mu = 0$ and $\sigma = 1$. Finally a z value of $z = F^{-1}(0.975) = 1.959964 \approx 1.96$ is gained.

As z value is now defined, it is possible to get the interval by a simple formula. Lower endpoint can be formulated as $X - 1.96\frac{roo}{sqrt(n)}$, where X is the mean of the measurement set, *roo* is the standard deviation of the measurement set and n is the amount of items in measurement set. Similarly getting the upper endpoint with $X + 1.96\frac{roo}{sqrt(n)}$. When using a set of 100 measurements the interval should be stable enough so that results obtained can be used.

The values we gather from the measurements are minimum, maximum, mean, confidence interval and variance of the measurement set. Mean is calculated iteratively by using $c(t+1) = c(t) + \frac{1}{t+1}(x - c(t))$ formula to calculate it. Selecting mean is the most common and obvious choice for measurements and variance is for reference and generality and as we are calculating it anyways, we can just add it to the results. Confidence interval is used to monitor the actual process of how accurate can the measurements be.

Because there are differences in how the logs are printed in different runtimes. The results have to be read from a different location when using different runtimes. NodeJS runtime uses console to print the information, so the information has to be read on the fly. LwLIoTR uses a log file to print the results to, the results gained are simply read from that file.

Interesting values gained from this procedure are especially the maximum execution time and mean execution time. Because with these, it is possible to see what has happened during the execution and how fast the task has generally been executed.

5.2.4 Measurement Procedure for Heap Memory

Heap memory measurements for LwLIoTR are done with *massif* and for NodeJS runtime Node.js's internal memory statistics are used to analyze heap consumption. This method is unfortunately biased towards the NodeJS runtime. However, it will give some insight towards the internal memory usage of runtimes even if the results are not fully reliable.

Because LwLIoTR is a pure C/C++ application that only uses internal threads, it is possible to use *massif* tool to measure heapsize effectively and accurately. The process that is used to measure heap statistics in LwLIoTR is the following.

1. Start the runtime with *valgrind –tool=massif ./liquid-server*.
2. If no application run is done, wait for 60 seconds, quit runtime and move to item 6. Otherwise move to next item.
3. Deploy ECMAScript application desired and wait 60 seconds.
4. Delete the ECMAScript application and wait 60 seconds.
5. Quit runtime.

6. Massif file has been generated and analyze it with *massif-visualizer*.

The procedure to measure heapsize of LwLIoTR is straightforward and recorded values are maximum heapsize and mean heapsize when analyzing the massif file. Also a heap profile picture is generated with *massif-visualizer* to give insight on how the heap behaves in certain situations.

When measuring NodeJS runtime's heapsize, we have to rely on Node.js function *process.memoryUsage()*, as it is fairly difficult to measure the heapsize of a Node.js application without using the internal function. The process of getting results from the NodeJS runtime is as follows.

1. Start runtime with *node manager_server.js*.
2. Deploy desired ECMAScript application and wait for 100 iterations.
3. When 100 iterations have been done, record the results from the console.

All the memory values are stored and recorded. However, for the heap size, maximum and mean *total heap* as well as maximum and mean *heap used* are the most important values gathered from the NodeJS runtime heap measurements. As other values have already been analyzed in section 5.2.2 with a more reliable method.

Overall the heap measurement processes give good insight on how the internal heap behaves during the execution of an application. However, any conclusions drawn from the NodeJS runtime heap measurements should be taken into careful consideration, as the runtime is essentially measuring itself.

5.3 Measurement Results

This section will summarize the results one section at a time and the measurements have been divided into 3 different subsections to make reading of the results overall easier. These subsections are *Pmap Results and Calculations* described in section 5.3.1, *Performance speed results* described in section 5.3.2 and finally going through the results from heap measurements in section 5.3.3.

In table 5.1 are the defined configurations that were used in measurements to make presenting the results easier and clearer by referencing a configuration instead of all the fields that are required to uniquely present a configuration. The measurements configuration table refers at the *Hardware* at table 4.1 and *Software* at table 4.2, which together will compose an unique configuration for the measurements.

5.3.1 Pmap Results and Calculations

In this section the results gained from the Procedures for Memory Measurement described in section 5.2.2 are presented. The measurements are divided into 2 tables because of

horizontal space limitations. These tables are 5.2 for the first half of the measurement vectors and 5.3 for the second half.

Possible errors in measurement results can be caused by shutting down the runtime and then restarting it, which will cause that particular runtime to have slightly different results, which is inevitable because there seems to be small problems with the memory management of both runtimes and they leak memory. It is actually unlikely to get exactly the same results after each instantiation of a runtime, especially with the NodeJS runtime, where Node.js framework brings its own memory management complexity with it. In NodeJS runtime the difference between test runs was observed to be quite high as *Rasp-Node-basic* measurement vector has lower memory consumption when compared to *Rasp-Node-none* and *Rasp-Node-perf* measurement vectors. Although, this kind of error might be indication of some deviation from the measurement procedure or simply Node.js garbage collector activating and removing something from the memory. Secondary error source is likely to be in the actual calculations of the values, as the values were manually inputted into the calculation, there is a possibility of human error.

The **measurement vector** is organized to sets of two, so that RSS and PSS values are bundled together, excluding last two values. First set is the overall RSS and PSS. *Local* keyword refers to the executable, [anon] and [stack] calculation, which can be formulated as internal memory usage of a process. *Rare* keyword refers to memory consumption of *libwebsockets* and *libarchive*. There are 2 measurements that do not follow this pattern and they are *General PSS* and *Modified PSS*. *General PSS* refers to the PSS calculation of all external libraries excluding *libwebsockets* and *libarchive*. Calculation of *Modified PSS* was described in section 5.2.2.

First set of measurements are shown in table 5.2. This table has the first half of the measurement vector, holding values *RSS*, *PSS*, *Local RSS* and *Local PSS*. *RSS* is quite self explanatory as it simply tells the total RSS of the runtime that was measured. *PSS* presents total PSS of the runtime. *Local RSS* refers to the executable, [anon] and [stack] RSS calculation in memory measurement procedures in section 5.2.2. *Local PSS* refers to same calculation with PSS values.

Second set of measurements are shown in table 5.3. This table has the second half of the measurement vector, holding values *Rare RSS*, *Rare PSS*, *General PSS* and *Modified PSS*. *Rare RSS* refers to *libwebsockets* and *libarchive* libraries and their memory consumption within the LwLIoTR and this field will always be 0 when NodeJS runtime is concerned. *Rare PSS* also refers to *libwebsockets* and *libarchive* libraries just with PSS values, and again, this field is always 0 when NodeJS runtime is concerned. *General PSS* refers to all other libraries excluding the *libwebsockets* and *libarchive* libraries. *Modified PSS* refers to the last item of either of the memory measurement procedures in section 5.2.2. Where *Modified PSS* essentially is a value that holds maximum local memory usage and average memory used by libraries.

Table 5.2. *Results from Memory Measurement Procedures part 1*

Configuration	RSS (KiB)	PSS (KiB)	Local RSS (KiB)	Local PSS (KiB)
Lap-LwLIoTR-none	9008	1664	2236	1252
Lap-Node-none	58740	23368	48088	22696
Rasp-LwLIoTR-none	7076	1360	1764	1064
Rasp-Node-none	48880	36280	45959	35164
Lap-LwLIoTR-basic	10372	2588	3224	2176
Lap-Node-basic	66608	29268	56180	28604
Rasp-LwLIoTR-basic	7948	1860	2256	1564
Rasp-Node-basic	43588	30808	40676	30716
Lap-LwLIoTR-memory	89020	81172	81708	80760
Lap-Node-memory	174956	137692	164596	137028
Rasp-LwLIoTR-memory	47732	41552	41960	41256
Rasp-Node-memory	123624	110700	120552	110596
Lap-LwLIoTR-perf	10716	2804	3440	2372
Lap-Node-perf	67408	29124	56512	28460
Rasp-LwLIoTR-perf	8204	2116	2524	1820
Rasp-Node-perf	54148	41464	51308	41364

Table 5.3. *Results from Memory Measurement Procedures part 2*

Configuration	Rare RSS (KiB)	Rare PSS (KiB)	General PSS (KiB)	Modified PSS (KiB)
Lap-LwLIoTR-none	308	32	388	2932
Lap-Node-none	0	0	704	48792
Rasp-LwLIoTR-none	244	12	276	2284
Rasp-Node-none	0	0	88	46044
Lap-LwLIoTR-basic	668	32	380	4272
Lap-Node-basic	0	0	656	56836
Rasp-LwLIoTR-basic	532	12	284	3072
Rasp-Node-basic	0	0	92	40768
Lap-LwLIoTR-memory	668	32	380	82756
Lap-Node-memory	0	0	664	165260
Rasp-LwLIoTR-memory	572	12	276	42808
Rasp-Node-memory	0	0	100	120652
Lap-LwLIoTR-perf	668	32	388	4496
Lap-Node-perf	0	0	664	57176
Rasp-LwLIoTR-perf	532	12	284	3340
Rasp-Node-perf	0	0	88	51396

5.3.2 Performance measurement results

In this section the results from the Measurement Procedure for Performance Speed, described in section 5.2.3 are presented. These measurements are in table 5.4. *Configuration* field will present the configuration used from table 5.1. *Max* is the maximum time used. *Min* is the minimum time used. *Mean* is the mean time used. *CI* is the confidence interval calculated. *Variance* is the variance of the measurement set.

Possible errors in measurement results can be caused by shutting down the runtime and then restarting it, which will cause that particular runtime to have slightly different results, which is inevitable because there seems to be small problems with the memory management of both runtimes and they leak memory. It is actually unlikely to get exactly the same results after each instantiation of a runtime, especially with the NodeJS runtime, where

Node.js framework brings its own memory management complexity with it. Secondary error source in table 5.4 is the error in the actual calculation of time difference, which happened few times when executing the measurements. This will cause the mean of previous set to be taken and added, thus lowering the CI and Variance artificially. This particular error was only observed when measuring NodeJS runtime.

***Table 5.4**. Performance speed results*

Configuration	Max(ms)	Min(ms)	Mean(ms)	CI	Variance
Lap-LwLIoTR-perf	648	620	623	[620, 626]	959
Lap-Node-perf	30	18	23	[21, 25]	266
Rasp-LwLIoTR-perf	6162	6000	6089	[6048, 6130]	168133
Rasp-Node-perf	540	487	503	[490, 516]	16220

5.3.3 Heap memory results

In this section the results from the Measurement Procedure for Heap Memory, described in section 5.2.4 are presented. Table 5.5 shows the results measured for the NodeJS runtime and values calculated from *massif* files for LwLIoTR. For LwLIoTR this basically means that only heap data is recorded. In figures 5.1, 5.2 and 5.3 the exact heap graphs that were generated with *massif-visualizer* for LwLIoTR are displayed, and as mentioned in section 5.2.4 there are time intervals for these measurements. First figure is measured with time frame of 60 seconds and second, and third figure are at 60 + 60 seconds time frame.

Possible errors in measurement results can be caused by shutting down the runtime and then restarting it, which will cause that particular runtime to have slightly different results, which is inevitable because there seems to be small problems with the memory management of both runtimes and they leak memory. It is actually unlikely to get exactly the same results after each instantiation of a runtime, especially with the NodeJS runtime, where Node.js framework brings its own memory management complexity with it. However, the scale will still be approximately the same. There is also a possible human error with the calculation of mean *Total heap* in LwLIoTR results as, they were calculated by hand.

The difference with *Total heap* and *Used heap* comes from the Node.js memory usage when *Total heap* is the reserved heap size and *Used heap* is the actual internally used heap. This means that only *Total heap* is calculated for LwLIoTR for any comparisons. Other values recorded with NodeJS runtime are for reference and intrigue.

5.4 Analysis of Measurements

In this section information gained from measurements in section 5.3 are gathered and analyzed. The order of analysis will be *pmap results and calculation* from section 5.3.1, *performance speed results* from section 5.3.2 and *heap memory results* from section 5.3.3.

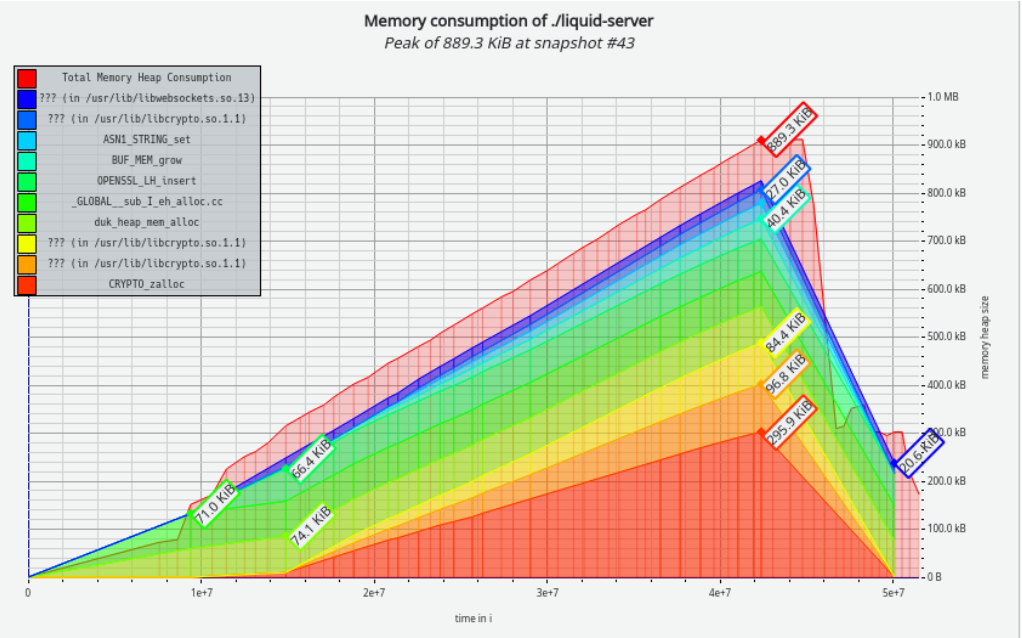

Figure 5.1. *LwLIoTR heap memory for none application*

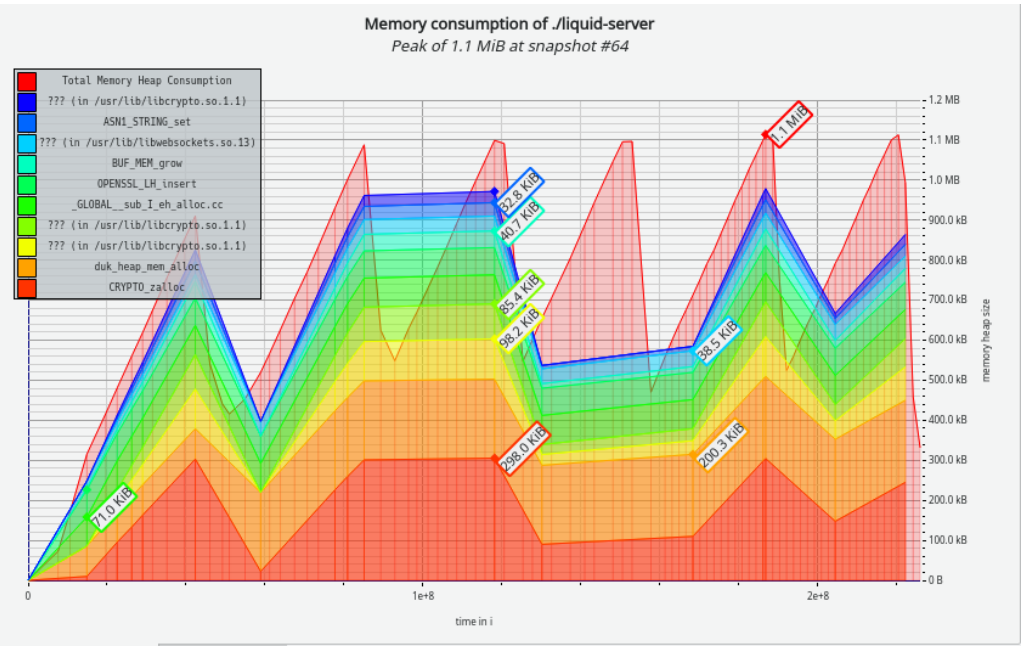

Figure 5.2. *LwLIoTR heap memory for basic application*

Table 5.5. *Heap memory results*

Configuration	Max rss (KiB)	Min rss (KiB)	Total heap max (KiB)	Total heap mean (KiB)	Used heap max (KiB)	Used heap mean (KiB)
Lap-LwLIoTR-none	-	-	889	498	-	-
Lap-Node-none	60428	59589	37156	21269	15775	15262
Lap-LwLIoTR-basic	-	-	1126	742	-	-
Lap-Node-basic	62944	62761	37668	23261	24329	17921
Lap-LwLIoTR-memory	-	-	148480	49585	-	-
Lap-Node-memory	207336	115211	186000	80865	162910	69850

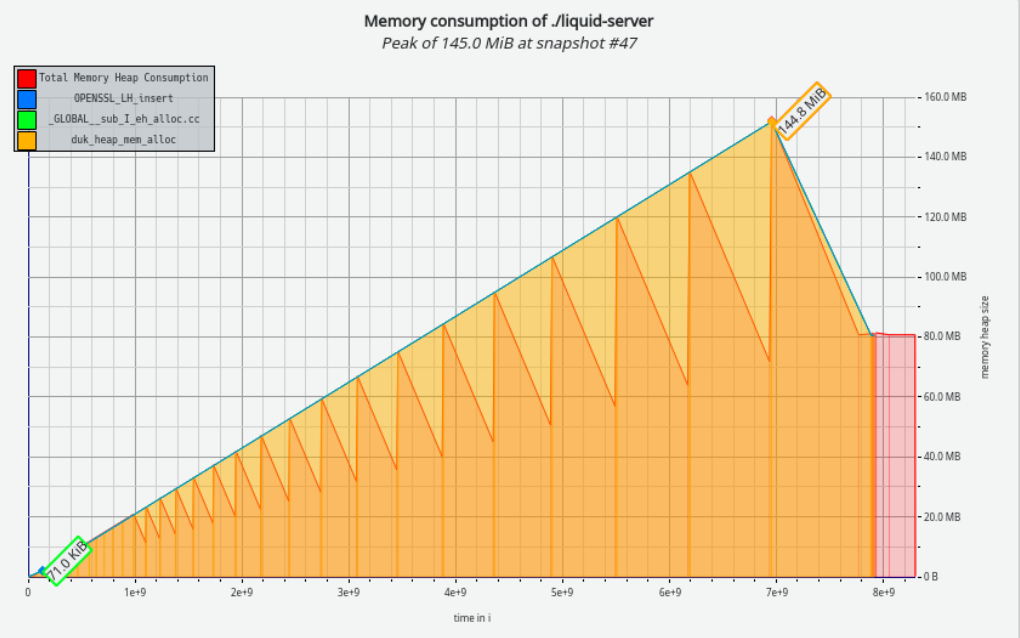

Figure 5.3. *LwLIoTR heap memory for memory consuming application*

5.4.1 Pmap results analysis

Pmap results from subsection 5.3.1 are divided into 4 categories. These categories are called *none*, *basic*, *memory* and *performance* and their respective applications are *no application*, *Basic application*, *Memory heavy application* and *Performance heavy application*. Tables that are used to gather the information from, are tables 5.2 and 5.3. The calculated analysis results are gathered to table 5.6.

When calculating *none* application analysis results, formula $\frac{noneNodeJS}{noneLwLIoTR}$ is used to calculate relative memory consumption of the runtimes when there are no applications present. NodeJS runtime uses approximately 6.5-7 times more memory when looking at both platforms respectively when using RSS. The difference with PSS memory consumption calculation is even more drastic as the difference between runtimes is up to 27 times more memory used and same with modified PSS. Although, the difference in *none* case is only the overhead between the 2 runtimes. In the future this overhead is taken into account when comparing the memories.

When *basic* application is used, similar amounts of memory is used between the runtimes

as with *none* application. However, there seems to be a case of runtime instance having a noticeably larger memory consumption to none application, in Raspberry PI measurements. This is likely caused by some deviation from the measurement procedure or simply Node.js garbage collector activating and removing something from the memory. Thus, in this case Raspberry PI measurement is partially ignored, unless *performace* application shows similar results. Although, when looking at table 5.6 the scale is actually similar when comparing to other results, just in negative, probably just a coincidence. If differences between *basic* application and *none* application are compared in Laptop for each relevant memory calculation. By using formula $\frac{basicNodeJS-noneNodeJS}{basicLwLIoTR-noneLwLIoTR}$, RSS memory difference is approximately 5.8 times more memory used, PSS memory difference is approximately 6.4 times more memory and finally for the modified PSS, 6 times more memory is used by NodeJS runtime. Thus, one instance of an ECMAScript application is significantly larger in NodeJS runtime.

When *memory* application is used, overheads become almost meaningless, so it is easy to look at the memory consumptions and see that NodeJS version uses more memory per allocation. When the same calculations are done for the memory application as for the basic application with formula $\frac{memoryNodeJS-noneNodeJS}{memoryLwLIoTR-noneLwLIoTR}$. RSS memory difference is 1.45 for the Laptop and 1.83 for Raspberry PI. PSS memory difference is 1.44 for the Laptop and 1.85 for Raspberry PI. Modified PSS memory difference is 1.46 for the Laptop and 1.84 for Raspberry PI. Significantly larger memory allocation size between Laptop and Raspberry PI might be the difference between the Node.js versions. However, LwLIoTR and Duktape still uses significantly less memory per allocation when compared to NodeJS runtime.

When *performance* application is measured memory wise, results are similar to the *basic* application. By using formula $\frac{performanceNodeJS-noneNodeJS}{performanceLwLIoTR-noneLwLIoTR}$, RSS memory difference is 5.1 times more in Laptop and 4.7 times more in Raspberry PI. PSS memory difference is 5 times more in Laptop and 6.9 times more Raspberry PI. Modified PSS memory difference is 5.4 times more in Laptop and 5.1 times more for Raspberry PI. The scale of the differences is roughly in the same scale, as with *basic* application.

Overall memory usage with LwLIoTR seems to be on a different level compared to NodeJS runtime. If we do not normalize against the *none* application the absolute differences between runtimes are up to 16 times more memory used if directly compared with *Modified PSS* memory. However, most interesting observed values are maximum memory used, which we can directly look at with RSS. This makes LwLIoTR to be in less than 16 MiB range with most ECMAScript applications, that do not intentionally allocate too much memory. Similarly NodeJS runtime has hard time of fitting into a range of 64 MiB as it was exceeded several times in Laptop environment.

Table 5.6. *Analysis of memory usage results. Relative usage is generally calculated by NodeJS runtime application as nominator and LwLIoTR application as denominator.*

	relative-none (KiB)	relative-basic (KiB)	relative-memory (KiB)	relative-performance (KiB)
Laptop-RSS	6.5	5.8	1.45	5.1
Laptop-PSS	14	6.4	1.44	5
Laptop-mod-PSS	16.6	6	1.46	5.4
PI-RSS	6.9	-6.1	1.83	4.7
PI-PSS	26.7	-10.9	1.85	6.9
PI-mod-PSS	20.2	-6.7	1.84	5.1

5.4.2 Analysis of Perfomance Results

Performance results from section 5.3.2 are fully presented in table 5.4 and configuration name is used to refer to each measurement vector. There are not many interesting or surprising things about the performance measurements, unless NodeJS runtime on Raspberry PI outperforming LwLIoTR on Laptop counts as a surprise.

Measurements were adjusted to be run in a reasonable time. Thus, making some of the measurements slightly too small to actually draw any other conclusions, than it is approximately instant as is the case with *Lap-Node-perf*. However, it still possible to see how much faster is the NodeJS runtime compared to LwLIoTR. On laptop the average speed difference is 27 times faster on average. On Raspberry PI the speed difference is on average 12 times, witch can be explained by the earlier version of Node.js and V8 engine at the same time so it is not that surprising.

Overall the NodeJS runtime will outperform LwLIoTR by a margin that is so significant, that there actually is no real comparison in performance. The massive difference can be fully contributed to the difference between Duktape and V8 engines as there is practically no overhead in LwLIoTR implementation of function execution.

5.4.3 Analysis of Heap Memory Results

Heap memory results from section 5.3.3 are fully presented in table 5.5 and the application name is used to refer to a measurement the same way as in section 5.4.1. Full graphs for the heap measurements for LwLIoTR are presented in figures 5.1, 5.2 and 5.3. Only *Total heap max* and *Total heap avg* fields are looked at during the analysis as they are the only fields that can be reasonably calculated from LwLIoTR measurements. There is still the question if these results are reliable. However, it seems like the results are realistic for NodeJS runtime, so *process.memoryUsage* function of Node.js seems to work approximately as expected. That said, most of the results analyzed are for the intrigue and are still not fully reliable.

In *none* application we can clearly see that LwLIoTR has really low overhead for the runtime itself. Where as, NodeJS runtime has quite large default heap size with approximately

36 MiB. From figure 5.1 we can see in the graph that Duktape heap only consumes 74 KiB in that instance as maximum heap memory usage for LwLIoTR is less than 1 MiB.

In *basic* application the Duktape heap usage of LwLIoTR only increases to 200 KiB by 126 KiB and the overall heap consumption increases by 237 KiB when *basic* application is added to the runtime and in practice such increases are practically unnoticeable. In NodeJS runtime, the mean total heap increased by approximately 2000 KiB. The figure 5.2 shows the heap fluctuations during the execution of the application. There might multiple reasons for these fluctuations, one is the activation of Duktape garbage collector, another possibility is opening and reading to log file, or it might be practically anything related to the *task* function that is being executed. Even the Duktape heap allocation is noticeable but still not overwhelming the overall memory consumption of LwLIoTR. Relative difference with formula $\frac{basicNodeJS-noneNodeJS}{basicLwLIoTR-noneLwLIoTR}$ is only 2.16 times more memory used when calculating *Total heap*.

In *memory* application the heap usages are overwhelmingly large, thus, it is not practical to think about the heap overhead too much in this case. In *mean Total heap* size, difference between runtimes is quite significant. However, probably in this case the max heap size tells more about the situation at hand as there might be differences that affect the mean too much, as we can seen in figure 5.3, there is a drop in the end that might affect the mean. The relative memory consumption with formula $\frac{memoryNodeJS-noneNodeJS}{memoryLwLIoTR-noneLwLIoTR}$ is 1.008, which makes the actual allocation of memory to be approximately equal.

The heap memory analysis gave us something to think about, if it is actually clear that the LwLIoTR will memorywise still significantly outperform NodeJS runtime on very memory heavy ECMAScript applications as there was some indication that this might not be the case. However, that is quite theoretical as the current *memory* application allocates an array thats size is $5 * 10^6$ and it is unlikely that larger arrays will be used in ECMAScript applications.

6. CONCLUSION

The goal with LwLIoTR was to make a LiquidIoT runtime environment that consumes only a small amount of memory, there were no exact numbers as a goal. As a little bit more concrete goal, a simple comparison to previous version of LiquidIoT runtime that was implemented with Node.js ECMAScript runtime was made. As the goal was to create a low memory version of LiquidIoT runtime, Duktape was the ECMAScript engine of choice. Thus, giving an ECMAScript engine that could compete in memory consumption against most ECMAScript engines.

LwLIoTR beats the memory consumption of NodeJS runtime in basically every measurement by a large margin. A large margin in this case means approximately a factor of 6 in most memory related measurements excluding some measurements related to Memory heavy application as that particular application allocated a lot of memory intentionally and there cannot really be several factors of difference in memory allocation. However, even if the memory consumption goal was met there still was the issue of performance. And performance speed wise LwLIoTR was not very successful and lost to NodeJS runtime by a factor of 10-30. So there was a trade off between memory consumption and performance and a significant one at that.

For actually running the LwLIoTR in small devices, LwLIoTR will run without any problems when there is approximately 16 MiB of memory available, maybe even less. In measurements, the maximum usage limit in a normal use case was 10716 KiB, which still has a lot of room to grow towards 16 MiB and this on a system that does not already natively use C/C++ libraries. With a different memory calculation method it is possible to get this amount to significantly less than 8 MiB. However, in this case, we probably should prepare for the worst and accept that LwLIoTR uses approximately 11 MiB of memory when there is one application deployed to it.

LwLIoTR fits well with all kinds of *nix based devices as long as some relatively modern libraries and a modern compiler is used. These libraries might be the actual bottleneck in integrating LwLIoTR into more exotic and smaller devices. Although it is possible that it is not a problem at all. There are multiple issues with current version of LwLIoTR that could be improved and those improvements could improve the overall efficiency of LwLIoTR in the future. One example of memory wise improvement could be slightly better usage of strings.

Possible future experiments could be the use of Google's V8 engine to have significant performance boost and not have the heaviness of Node.js. This, however would significantly increase the overall memory consumption of LwLIoTR and one would need to

carefully think on how much memory there is available on a target device. Some other improvements for the LwLIoTR could be improving the actual capabilities of the runtime by, for example adding support for external devices such as sound or temperature sensors. Another improvement could be adding of native C++ based libraries that could be loaded when desired onto an ECMAScript application as the current implementation only supports pure ECMAScript libraries.

Overall LwLIoTR succeeds in the original goal relatively well, even if there is a lot of room for enhancements. However, the absolute numbers in memory consumption are so small that these enhancements could be made without worrying about bloating LwLIoTR too much.

REFERENCES

[1] F. Ahmadighohandizi, Node.js LiquidIoT runtime. Available (accessed on 13.10.2018): https://github.com/farshadahmadi/liquidiot-server

[2] F. Ahmadighohandizi, K. Systä, Application development and deployment for IoT devices, in: European Conference on Service-Oriented and Cloud Computing, 2016, Springer, pp. 74–85.

[3] Boost C++ library. Available (accessed on 26.8.2018): https://www.boost.org/

[4] Duktape. Available (accessed on 18.7.2018): http://duktape.org/

[5] ECMAScript E5/E5.1. Available (accessed on 18.7.2018): http://www.ecma-international.org/ecma-262/5.1/index.html

[6] ECMAScript E6. Available (accessed on 18.7.2018): http://www.ecma-international.org/ecma-262/6.0/index.html

[7] ECMAScript E7. Available (accessed on 18.7.2018): http://www.ecma-international.org/ecma-262/7.0/index.html

[8] R.T. Fielding, Architectural styles and the design of network-based software architectures, Architectural styles and the design of network-based software architectures, 2000.

[9] Google V8. Available (accessed on 18.7.2018): https://developers.google.com/v8/

[10] Gzip. Available (accessed on 1.9.2018): https://www.gnu.org/software/gzip/

[11] Launching Ignition and TurboFan, May 15,, 2017. Available (accessed on 21.9.2018): https://v8project.blogspot.com/2017/05/launching-ignition-and-turbofan.html

[12] Libarchive. Available (accessed on 18.7.2018): http://www.libarchive.org/

[13] Libwebsockets. Available (accessed on 18.7.2018): https://libwebsockets.org/

[14] LiquidIoT Resource Registry. Available (accessed on 13.10.2018): https://github.com/ohylli/liquidiot-manager

[15] W. Milian, Massif-visualizer. Available (accessed on 28.8.2018): https://github.com/KDE/massif-visualizer

[16] NodeJS. Available (accessed on 24.7.2018): https://nodejs.org/

[17] Node.js modules. Available (accessed on 24.9.2018): https://nodejs.org/api/modules.html

[18] OpenAPI. Available (accessed on 27.9.2018): https://github.com/OAI/OpenAPI-Specification

[19] Performance interface. Available (accessed on 26.9.2018): https://developer.mozilla.org/en-US/docs/Web/API/Performance

[20] Web of Objects. Available (accessed on 10.7.2018): https://itea3.org/project/web-of-objects.html

[21] E.W. Weisstein, Erf. Available (accessed on 27.9.2018): http://mathworld.wolfram.com/Erf.html

APPENDIX A: BASIC APPLICATION

```
1  var repeat = true;
2  var interval = 3000;
3  var counter = 0;
4
5  var memory_measurement_values = {
6    "max_rss" : 0, "avg_rss" : 0, "max_heap_total" : 0,
7    "avg_heap_total" : 0, "max_heap_used" : 0, "avg_heap_used" : 0,
8    "max_external" : 0, "avg_external" : 0
9  };
10
11 $app.$configureInterval(repeat, interval);
12
13 $app.$initialize = function(initCompleted) {
14   counter = 0;
15   console.log("Counter␣initialized␣at␣" + counter);
16   initCompleted();
17 };
18
19 $app.$task = function(taskCompleted) {
20   counter = counter + 1;
21   console.log("Counter␣is␣at␣" + counter);
22
23   if(typeof process != 'undefined') {
24     var current_mem = process.memoryUsage();
25
26     if(memory_measurement_values.max_rss < current_mem.rss) {
27       memory_measurement_values.max_rss = current_mem.rss;
28     }
29
30     if(memory_measurement_values.avg_rss === 0) {
31       memory_measurement_values.avg_rss = current_mem.rss;
32     } else {
33       memory_measurement_values.avg_rss =
34         memory_measurement_values.avg_rss +
35         (1 / (counter + 1)) *
36         (current_mem.rss - memory_measurement_values.avg_rss);
37     }
38
39     if(memory_measurement_values.max_heap_total <
40                 current_mem.heapTotal)
41     {
42       memory_measurement_values.max_heap_total = current_mem.heapTotal;
43     }
44
```

```
45      if(memory_measurement_values.avg_heap_total === 0) {
46        memory_measurement_values.avg_heap_total = current_mem.heapTotal;
47      } else {
48        memory_measurement_values.avg_heap_total =
49          memory_measurement_values.avg_heap_total +
50          (1 / (counter + 1)) * (current_mem.heapTotal -
51          memory_measurement_values.avg_heap_total);
52      }
53
54      if(memory_measurement_values.max_heap_used < current_mem.heapUsed)
55      {
56        memory_measurement_values.max_heap_used = current_mem.heapUsed;
57      }
58
59      if(memory_measurement_values.avg_heap_used === 0) {
60        memory_measurement_values.avg_heap_used = current_mem.heapUsed;
61      } else {
62        memory_measurement_values.avg_heap_used =
63          memory_measurement_values.avg_heap_used +
64          (1 / (counter + 1)) * (current_mem.heapUsed -
65          memory_measurement_values.avg_heap_used);
66      }
67
68      if(memory_measurement_values.max_external < current_mem.external) {
69        memory_measurement_values.max_external = current_mem.external;
70      }
71
72      if(memory_measurement_values.avg_external === 0) {
73        memory_measurement_values.avg_external = current_mem.external;
74      } else {
75        memory_measurement_values.avg_external =
76          memory_measurement_values.avg_external +
77          (1 / (counter + 1)) * (current_mem.external -
78          memory_measurement_values.avg_external);
79      }
80
81      console.log("Iterations␣" +  counter);
82      console.log("Max␣rss␣" +
83        Math.round(memory_measurement_values.max_rss / 1024));
84      console.log("Avg␣rss␣" +
85        Math.round(memory_measurement_values.avg_rss / 1024));
86      console.log("Max␣heapTotal␣" +
87        Math.round(memory_measurement_values.max_heap_total / 1024));
88      console.log("Avg␣heapTotal␣" +
89        Math.round(memory_measurement_values.avg_heap_total / 1024));
90      console.log("Max␣heapUsed␣" +
91        Math.round(memory_measurement_values.max_heap_used / 1024));
92      console.log("Avg␣heapUsed␣" +
93        Math.round(memory_measurement_values.avg_heap_used / 1024));
94      console.log("Max␣external␣" +
95        Math.round(memory_measurement_values.max_external / 1024));
```

```
96      console.log("Avg external " +
97        Math.round(memory_measurement_values.avg_external / 1024))
98    }
99
100   taskCompleted();
101 };
102
103 $app.$terminate = function(terminateCompleted){
104   console.log("Counting finished");
105   terminateCompleted();
106 };
107 /**
108  * Application Interface: normal_app_api
109  */
110 $router.get("/counter", function(req, res){
111   res.setHeader("Content-type", "application/json");
112   res.statusCode = 200;
113
114   var resp_obj = { "counter" : counter };
115   res.write(JSON.stringify(resp_obj));
116   res.end();
117 });
118 // normal_app_api - end
```

Program A.1. *Basic ECMAScript application used in measurments.*

APPENDIX B: MEMORY CONSUMING APPLICATION

```
1  var max_reserve = 5e6;
2  var counter = 0;
3  var arr = [];
4
5  var memory_measurement_values = {
6    "max_rss" : 0, "avg_rss" : 0, "max_heap_total" : 0,
7    "avg_heap_total" : 0, "max_heap_used" : 0, "avg_heap_used" : 0,
8    "max_external" : 0,
9    "avg_external" : 0
10 };
11
12 $app.$configureInterval(true, 10000);
13
14 $app.$initialize = function(initCompleted){
15   for(var i = 0; i < max_reserve; ++i) {
16     arr.push("Some␣random␣text");
17   }
18   initCompleted();
19 };
20
21 $app.$task = function(taskCompleted) {
22   if(typeof process != 'undefined') {
23     counter = counter + 1;
24     var current_mem = process.memoryUsage();
25
26     if(memory_measurement_values.max_rss < current_mem.rss) {
27       memory_measurement_values.max_rss = current_mem.rss;
28     }
29
30     if(memory_measurement_values.avg_rss === 0) {
31       memory_measurement_values.avg_rss = current_mem.rss;
32     } else {
33       memory_measurement_values.avg_rss =
34         memory_measurement_values.avg_rss + (1 / (counter + 1)) *
35         (current_mem.rss - memory_measurement_values.avg_rss);
36     }
37
38     if(memory_measurement_values.max_heap_total <
39                 current_mem.heapTotal)
40     {
41       memory_measurement_values.max_heap_total = current_mem.heapTotal;
42     }
```

```
43
44      if(memory_measurement_values.avg_heap_total === 0) {
45        memory_measurement_values.avg_heap_total = current_mem.heapTotal;
46      } else {
47        memory_measurement_values.avg_heap_total =
48          memory_measurement_values.avg_heap_total +
49          (1 / (counter + 1)) *
50          (current_mem.heapTotal -
51          memory_measurement_values.avg_heap_total);
52      }
53
54      if(memory_measurement_values.max_heap_used < current_mem.heapUsed)
55      {
56        memory_measurement_values.max_heap_used = current_mem.heapUsed;
57      }
58
59      if(memory_measurement_values.avg_heap_used === 0) {
60        memory_measurement_values.avg_heap_used = current_mem.heapUsed;
61      } else {
62        memory_measurement_values.avg_heap_used =
63          memory_measurement_values.avg_heap_used +
64          (1 / (counter + 1)) *
65          (current_mem.heapUsed -
66          memory_measurement_values.avg_heap_used);
67      }
68
69      if(memory_measurement_values.max_external < current_mem.external) {
70        memory_measurement_values.max_external = current_mem.external;
71      }
72
73      if(memory_measurement_values.avg_external === 0) {
74        memory_measurement_values.avg_external = current_mem.external;
75      } else {
76        memory_measurement_values.avg_external =
77          memory_measurement_values.avg_external +
78          (1 / (counter + 1)) *
79          (current_mem.external -
80          memory_measurement_values.avg_external);
81      }
82
83      console.log("Iterations␣" +  counter);
84      console.log("Max␣rss␣" +
85        Math.round(memory_measurement_values.max_rss / 1024));
86      console.log("Avg␣rss␣" +
87        Math.round(memory_measurement_values.avg_rss / 1024));
88      console.log("Max␣heapTotal␣" +
89        Math.round(memory_measurement_values.max_heap_total / 1024));
90      console.log("Avg␣heapTotal␣" +
91        Math.round(memory_measurement_values.avg_heap_total / 1024));
92      console.log("Max␣heapUsed␣" +
93        Math.round(memory_measurement_values.max_heap_used / 1024));
```

```
94      console.log("Avg␣heapUsed␣" +
95        Math.round(memory_measurement_values.avg_heap_used / 1024));
96      console.log("Max␣external␣" +
97        Math.round(memory_measurement_values.max_external / 1024));
98      console.log("Avg␣external␣" +
99        Math.round(memory_measurement_values.avg_external / 1024));
100   }
101
102   taskCompleted();
103 };
104
105 $app.$terminate = function(terminateCompleted){
106     terminateCompleted();
107 };
```

Program B.1. *Memory consuming ECMAScript application used in measurements*

APPENDIX C: PERFORMANCE HEAVY APPLICATION

```
1  var time_0;
2  var time_1;
3  var fin_time = 0;
4
5  var counter = 0;
6
7  var performance_measurement_values = {
8    "min_exec" : -1,
9    "max_exec" : -1,
10   "avg_exec" : -1
11 };
12
13 var performance_set = [];
14
15 $app.$configureInterval(true, 1000);
16
17 $app.$initialize = function(initCompleted){
18   initCompleted();
19 };
20
21 $app.$task = function(taskCompleted){
22   console.log("Task␣started");
23   if(typeof performance != 'undefined')
24     time_0 = performance.now();
25   else
26     time_0 = process.hrtime()
27
28   var arr = [];
29   for(var i = 0; i < 100; ++i) {
30     arr.push(i);
31   }
32
33   for(i = 0; i < 1e5; ++i) {
34     arr.reverse();
35   }
36
37   if(typeof performance != 'undefined') {
38     time_1 = performance.now();
39     fin_time = time_1 - time_0;
40   } else {
41     time_1 = process.hrtime();
42     fin_time = (time_1[1] - time_0[1]) / 1e6;
```

```
43    }
44
45    console.log("Task␣ended");
46    console.log("Execution␣took␣" + fin_time + "␣milliseconds.");
47    if(fin_time < 0) {
48      fin_time = performance_measurement_values.avg_exec;
49    }
50    performance_set.push(fin_time);
51    counter = counter + 1;
52    console.log("Iterations␣" +  counter);
53
54    if(performance_measurement_values.max_exec < 0 ||
55      performance_measurement_values.max_exec < fin_time) {
56      performance_measurement_values.max_exec = fin_time;
57    }
58
59    if(performance_measurement_values.min_exec < 0 ||
60      performance_measurement_values.min_exec > fin_time) {
61      performance_measurement_values.min_exec = fin_time;
62    }
63
64    if(performance_measurement_values.avg_exec < 0) {
65      performance_measurement_values.avg_exec = fin_time;
66    } else {
67      performance_measurement_values.avg_exec =
68        performance_measurement_values.avg_exec + (1 / (counter + 1)) *
69        (fin_time - performance_measurement_values.avg_exec);
70    }
71
72    console.log("Max␣exec␣time␣" +
73      Math.round(performance_measurement_values.max_exec));
74    console.log("Min␣exec␣time␣" +
75      Math.round(performance_measurement_values.min_exec));
76    console.log("Avg␣exec␣time␣" +
77      Math.round(performance_measurement_values.avg_exec));
78    var variance = getVariance(performance_set,
79      performance_measurement_values.avg_exec);
80    console.log("Variance␣" + variance);
81    console.log("Confidence␣interval␣[" +
82      (performance_measurement_values.avg_exec -
83      (Math.sqrt(variance) / Math.sqrt(counter))) +
84      ",␣" + (performance_measurement_values.avg_exec +
85      (Math.sqrt(variance) / Math.sqrt(counter))) + "]");
86
87    taskCompleted();
88  };
89
90  $app.$terminate = function(terminateCompleted){
91    terminateCompleted();
92  };
93
```

```
94 function getVariance(arr, mean) {
95   return arr.reduce(function(pre, cur) {
96     pre = pre + Math.pow((cur - mean), 2);
97     return pre;
98   }, 0)
99 }
```

Program C.1. *Performance heavy ECMAScript application used in measurements*