UiT The Arctic University of Norway

Faculty of Science and Technology, Department of Computer Science

Exploring the Challenges of a Flexible, Feature Rich IoT Testbed

A practical implementation, and lessons learned.

Alexander Jakobsen Hansen

Master’s thesis in Computer Science…INF-3990…May 2023

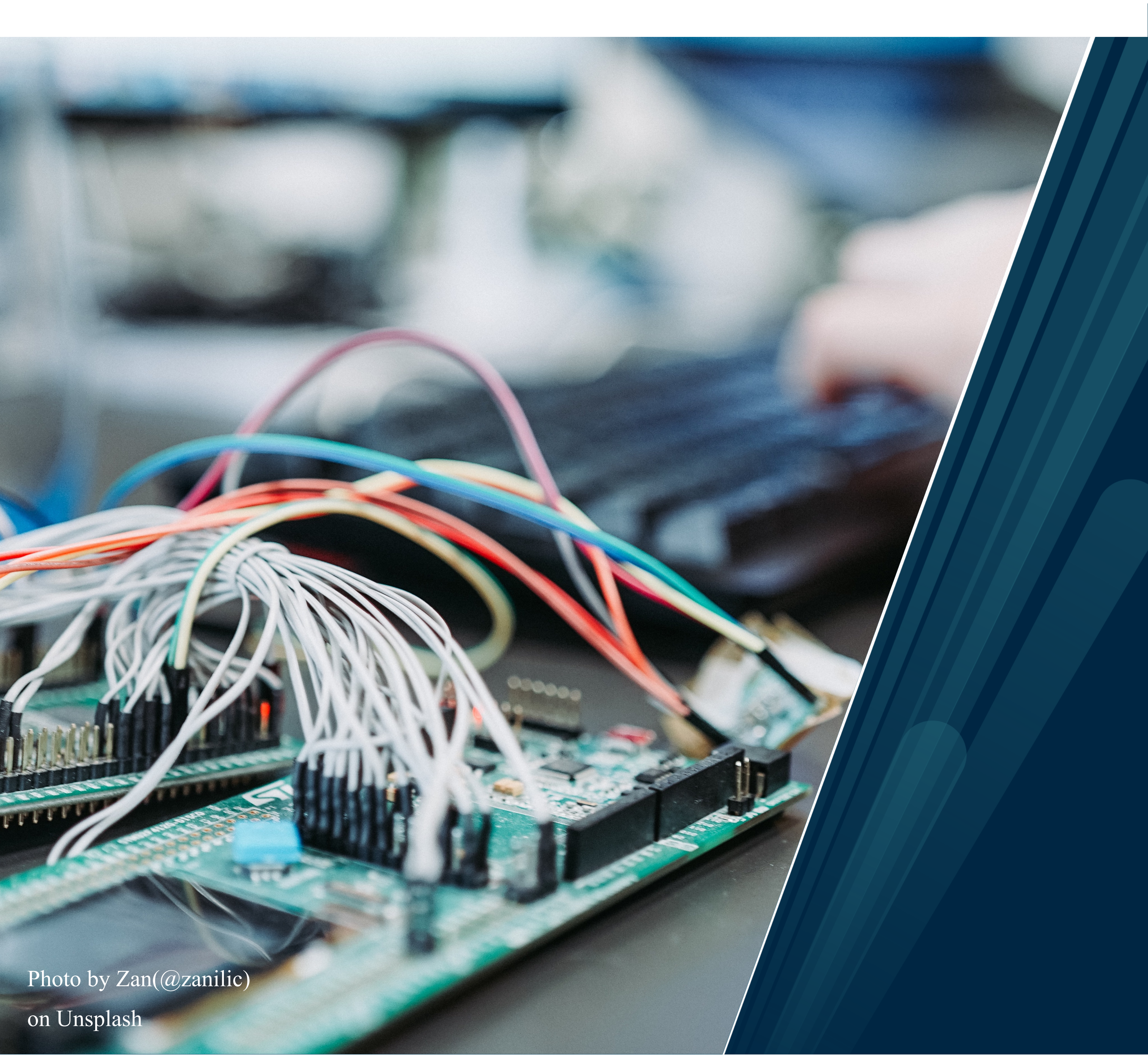


Photo by Zan(@zanilic)
on Unsplash

Contents

List of Figures

List of Tables

1 Abstract

IoT is a field of technology of ever growing importance in our daily lives. From smart cities, health devices, climate observations, appliances, and so much more, IoT surrounds us now more than ever. The types of devices being added to IoT networks is ever growing, and as this variety of hardware and software increases, so does the difficulty of working with them. Ensuring inter-compatibility between devices, testing new communication protocols, and writing software for emerging technologies becomes a complex challenge. To help solve this challenge are IoT Testbeds.

IoT Testbeds help developers, researchers, and many more groups of people explore and test their IoT solutions in contexts of real IoT Devices. These testbeds exist today, but as far as we know, no Jack of all trades testbed exists that supports all features one might want from a testbed. This thesis will introduce a first draft of a new testbed. Introducing a system design, architecture, and implementation that theoretically and practically implements all these features. Also highlighting issues with this design and ways to tackle them. In the end contributing a foundation onto which a powerful system could be built.

The challenge the thesis aims to tackle is, in short:

> What are the needed features that make up a good testbed? And how can we incorporate these features into a simple, flexible, unified system?

2 Introduction

The Internet of Things is an ever growing set of interconnected smart devices that help make our lives simpler, and easier. The first thing that comes to mind for a lot of us hearing the term ”IoT” is probably our smart light-bulbs, heating, fridges, and other home appliances. While these are all to be thought of as IoT devices there are so many other use cases that can be considered to fit under the IoT umbrella. IoT devices can be integrated into healthcare[3], arctic climate monitoring[10], agriculture[14], and so much more. This thesis will talk about IoT, but from the perspective of what goes into developing software for IoT devices. Specifically: connecting developers to IoT devices they can test their software with.

When developing software for an IoT device there are initially two ways of doing so. You either place the IoT device in the field and perform field tests, or you put it in a lab environment. While some devices can be field tested in your backyard, others might need to be field tested in far more remote, harsh environments. The positive side of this approach is the accuracy you get of how your device performs in it’s production environment, but the drawback comes in the form of difficulty of management. Updates can take a long time to transfer over poor connections, or might even require a full trek into some rough terrain. Testing in a lab on the other hand can significantly simplify the development process, but unless special care is taken to exactly match a production environment, it doesn’t give as good of a guarantee that your device will survive it’s production environment.

So it seems that either you get the accuracy of testing in the field, or the ease of development from testing in a lab. Though there is a way to experience the best of both worlds. The answer is an *IoT Testbed.* An IoT Testbed is a system that facilitates access to IoT devices to a group of end users, and offers features for running experiments on these IoT devices, as well as analyzing their execution. Such a testbed can integrate IoT devices of all kinds, and in all sorts of environments, from simulated to real environments, and can offer all sorts of arrays of hardware, sensors, and communication technologies for developers to test their software against. Furthermore it also means that developers and researchers don’t always need to be hands on with their hardware. This can help the development of projects where budgets are tight, or where hardware integration is a time consuming process that is not desirable. Opening IoT development to more people, and creating a good foundation for progress in the field.

Currently there are a few testbeds on the market. For example the FIT IoT Lab[1], the SmartSantander project[12], OpenCity[15], and more. Many of these testbeds are generic and multi-purpose, but looking at the sum of all their features, none, as far as we know, support all of them at once.

This thesis will present a new testbed system that will support this sum of all features. Both in the form of a theoretical design, and a simple practical implementation. This implementation will then be thoroughly discussed and critiqued so that future work can be done to improve upon it. Presenting a good foundation for a testbed which implements all needed, major features. As well as being flexible enough to support future features as well.

3 Technical Background

3.1 What is IoT?

The Internet of Things is an ever growing field of technology which encompasses a large group of digital devices that are connected to the internet. Things that normally won't fall under the ordinary definition of ”Personal Computer”, but instead *things* that have been given computing capabilities. Things like smart watches, glasses, health sensors, smart weights, smart traffic control, climate observations, and more. [9] The exact definition is purposely vague, as to encompass the ever increasing number and types of devices that the Internet of Things contains. These devices often utilize a combination of edge and cloud computing to offer powerful features to their end users.

3.2 What is an IoT Device

> ”IoT devices are pieces of hardware, such as sensors, actuators, gadgets, appliances, or machines, that are programmed for certain applications and can transmit data over the internet or other networks. They can be embedded into other mobile devices, industrial equipment, environmental sensors, medical devices, and more.” [2]

IoT devices are small compute units that usually sit closer to the user, on the edge of a network, and perform all sorts of tasks from monitoring climate, to controlling lights and heating, to managing traffic, and so much more. These devices are usually low power compute units with sensor arrays and often physically control some real world system. The coming together of a large amount of these devices is what forms the Internet of Things.

3.3 What is an IoT Testbed?

IoT testbeds can come in many shapes and sizes. Testbeds can be anything from rooms which explore how IoT devices and humans interact[11], city scale experimental platforms[12], to large scale compute clusters that expose generic IoT devices to researches and developers[1].

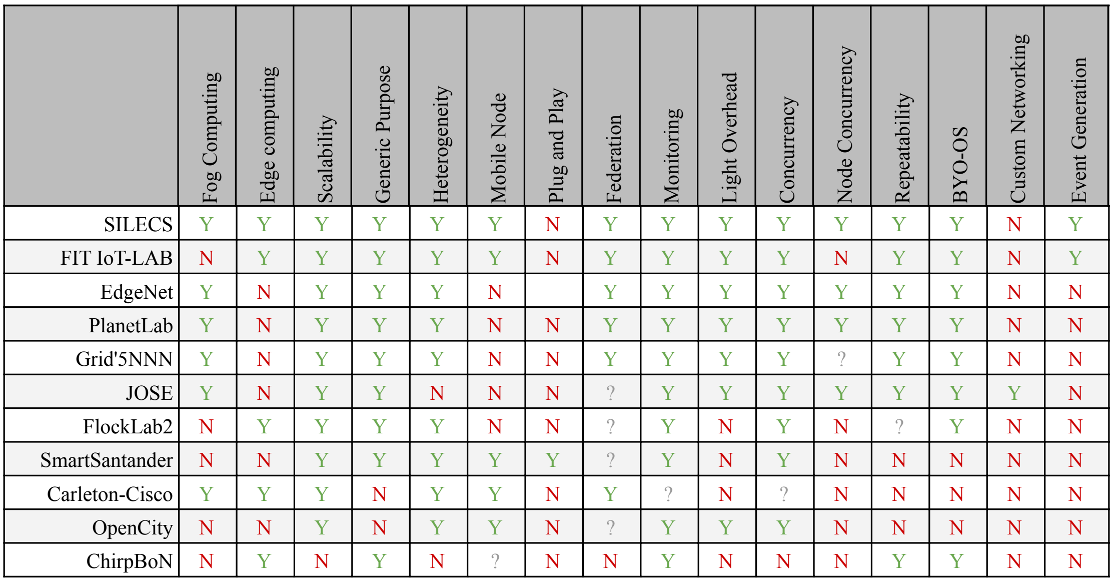
With the varying nature of what an IoT testbed is, we have devised a more general definition, and a definition which will fit the IoT Testbed described in this thesis: An

IoT Testbed is a system which unites IoT compute nodes with end users, whom use these nodes for all sorts of purposes. These purposes range from testing out new communication protocols, software, exploring the capabilities of IoT devices, and more.

This definition encompasses generic, multipurpose testbeds, such as the FIT IoT Lab[1]. Testbeds which are meant to flexibly solve many problems.

3.4 Requirements for an IoT Testbed

Due to the varying nature of what an IoT testbed is, there is no defined super-set of specific requirements for what a testbed should do. However, in an internship report written for the department of computer science at UiT The Arctic University of Norway, Bruno Maxime identified a number of requirements for a fully fledged testbed system by looking at existing testbeds on the market today. The testbeds analyzed, as well as their feature support can be found in the feature matrix of table 1. The report has identified a sum of features from all the listed testbeds, of which none of the testbeds support all of them.



	Fog Computing	Edge computing	Scalability	Generic Purpose	Heterogeneity	Mobile Node	Plug and Play	Federation	Monitoring	Light Overhead	Concurrency	Node Concurrency	Repeatability	BYO-OS	Custom Networking	Event Generation
SILECS	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	Y	N	Y
FIT IoT-LAB	N	Y	Y	Y	Y	Y	N	Y	Y	Y	Y	N	Y	Y	N	Y
EdgeNet	Y	N	Y	Y	Y	N		Y	Y	Y	Y	Y	Y	Y	N	N
PlanetLab	Y	N	Y	Y	Y	N	N	Y	Y	Y	Y	Y	Y	Y	N	N
Grid'5NNN	Y	N	Y	Y	Y	N	N	Y	Y	Y	Y	?	Y	Y	N	N
JOSE	Y	N	Y	Y	N	N	N	?	Y	Y	Y	Y	Y	Y	Y	N
FlockLab2	N	Y	Y	Y	Y	N	N	?	Y	N	Y	N	?	Y	N	N
SmartSantander	N	N	Y	Y	Y	Y	Y	?	Y	N	Y	N	N	N	N	N
Carleton-Cisco	Y	Y	Y	N	Y	Y	N	Y	?	N	?	N	N	N	N	N
OpenCity	N	N	Y	N	Y	Y	N	?	Y	Y	Y	N	N	N	N	N
ChirpBoN	N	Y	N	Y	N	?	N	N	Y	N	N	N	Y	Y	N	N

Table 1: Feature Matrix showing what features are implemented in what existing testbeds. [4]

To summarize, the features identified are as follows:

- Fog Computing

- Edge Computing
- Scalability
- Generic Purpose
- Heterogeneity
- Mobile Nodes
- Plug and Play
- Federation
- Monitoring
- Light Monitoring and Management Overhead
- Concurrency
- Node Concurrency
- Repeatability
- Experience as Fresh OS, or BYO-OS as we call it
- User Specified Network Architecture
- Event Generation

[4]

3.4.1 Requirement clarifications

Many of the above listed requirements should be quite self explanatory, but some are more vague and could do with some more thorough definitions. These definitions are our interpretations of the requirements, augmented with the observations made in the internship report [4].

Event generation Event generation pertains to the fact that in the real world IoT nodes might experience all sorts of events such as power loss, network connectivity issues, sensor failures, and so much more. These events should not happen randomly at the will of the node, as this could interfere with experiments. Instead these events should be simulated, generated, and configurable. The user should be able to say that power-loss happens on average every 10 hours, and a sensor failure every 2, for example.

This way the user can design their software around the environment they expect. From nicer, more forgiving ones, to the harshest environments they can imagine.

Concurrency vs. Node concurrency Concurrency in general refers to the ability to do more than one thing at a time. In the context of our testbed concurrency comes in two forms. Testbed concurrency, and node concurrency. Testbed concurrency refers to the ability for the testbed to run more than one experiment at the time. Multiple users should be able to utilize the system at the same time. Node concurrency refers to the ability for a single hardware IoT node to be able to run multiple experiments at the same time. If they have the hardware power to do so.

Monitoring Monitoring refers to keeping track of nodes. Both for the purpose of keeping track of the node's health, but also keeping track of the performance impacts, power usage, etc. of your experiment on the node. Offering statistics about your program at the end of it's execution for analysis purposes.

Fog & edge computing, & heterogeneity Fog and edge computing refers to having computers in a non centralized location. This means that the nodes of the testbed aren't necessarily devices in a computing rack in a central location, but could be IoT devices distributed across a large space, or on the edge of a network. Heterogeneity refers to these computing devices possibly being comprised of all sorts of hardware and software stacks.

Plug and play Plug and play refers to nodes in the system needing to be quick and easy to set up, preferably you just plug it in with a bit of configuration and they will work.

Repeatability Repeatability refers to the ability to start an experiment and repeat it multiple times over with as identical parameters as possible to be able to recreate the experiment equally every time. This will help with certain debugging purposes.

Federation Federation refers to the ability to take existing compute systems/testbeds and integrate them into this testbed. So an existing stack of IoT nodes should have some path for being integrated into this testbed without significantly large changes to the infrastructure.

4 System design

To implement this new testbed system we must first put some prior thought into how the system should be cobbled together. We must go through a theoretical design stage where we plan, in rough details, how this system should look. We will do so by looking at the requirements identified, user perspectives, and design philosophies.

4.1 Groundwork

When approaching the design of a system like this there is some groundwork that needs to be done. We need to take a closer look at the background of the system, and understand this background. We will take a look at the requirements of the system, and how we expect the system to work from a user perspective.

4.1.1 Requirements - Rapidly growing complexity

Testbeds have a plethora of different requirements depending on their use cases. Some testbeds are more specific in their design as they are intended to solve a specific problem, with a specific subset of hardware and software in mind. An example of such a testbed is ChirpBox. A testbed designed specifically for LoRa devices, with limited functionality [13]. Other testbeds offer a broader feature set, such as the FiT IoT Lab [1]. Though of the features identified in the technical background no testbed, at least as far as we know, exists that supports all of them. This is what we're here to change. To begin tackling this task we must first look at the requirements, and how they interweave with each other to shape our application.

The requirements listed in the technical background are plenty, but we have decided to narrow them down slightly to simplify both the implementation, and the rest of this thesis. Requirements that have been removed are considered to be a part of other requirements, are implied, or are considered of lesser importance without worrying that they can't be implemented later. The new list of requirements is as follows:

- Concurrency
- Heterogeneity
- Node management

- Experiment management
- Node concurrency
- Monitoring
- Logs
- BYO-OS
- Custom networking
- Mobility
- Event generation

While the now more limited number of requirements are not *that* many, each requirement alone is potentially a complex one, and necessitates many design decisions to be made surrounding them. Furthermore the intertwined web these requirements weave becomes rapidly complex. No feature is awfully complex to implement on it's own, but the combination of them creates a far more difficult problem.

Looking at the requirements the users must be able to run multiple experiments at once, possibly even on each node, in the context of ”experiments”. Not necessarily complicated on it's own, but, there needs to be support for many different types of hardware, which complicates things. Made even more so by the requirement for logging and monitoring as how this is done varies from system to system. Furthermore users must be able to bring their own operating system, and will be expecting support for generating events, both of which need to support all sorts of different hardware configurations. Not made easier by the fact that they must support mobility, and varying network topology. The hardware also needs to be managed in some way.

Any of the above requirements are not complex on their own, but the combination of supporting all of them creates a massively complex solution space. The number of combinations of hardware, software, and use cases becomes so large that no single developer can possibly hope to implement support for all of it. As mentioned before, there aren't any testbeds on the market that support all features (as far as we know), and this is a reason as to why no do-it-all testbed currently exists. Still this thesis aims to bring to the table a testbed which has all these features. We will come back to the how and why. What matters now is that you understand the scale of the problem, and why the requirements pose such a large issue.

4.1.2 Understanding the system requirements from a user's perspective

Understanding that the requirements of the system are complex together is only a small fraction of the groundwork. To truly understand the requirements and what they mean in the context of an IoT Testbed, we must also analyze how we imagine these requirements and the following features to be used by the end users. As such we must also understand our end users.

The intended users of this system would be a mix of developers, hardware integrators, researchers, and more. Generally a group of people who should have slightly above average skills in the context of computers. This however is not an excuse for developing a system that is hard and complex to use. A user unfriendly system is user unfriendly regardless of the level of the end user. However we are given some leeway as to just how the user is exposed to the system. We are fine giving them a nice, albeit rudimentary command line interface, rather than a fully fledged, modern, sleek web interface.

This also helps influence the imagined use patterns of the system. We expect users to have the technical knowledge required to manage simple APIs, configuration files, and some more low level technical terminology. We also expect users to be, for the most part, self helped, and we don't need to offer a lot in terms of automatic assistance. This is not something that should be taken too far either however. Automation is good, but we don't need one click solutions either.

Furthermore we need to take a stance as to which environment this system is going to be deployed into. We want the system to be possible to deploy in all sorts of contexts. A small local university lab with a handful of devices, to a large, global system. In a distributed computing cluster, or on a low power Raspberry Pi with some Arduinos. We need the system to run in almost every situation, again expanding the space in which we need to operate, further increasing complexity.

4.1.3 Imagining a user interaction

With the above knowledge we perform a mental exercise. We imagine ourselves a number of nodes, and a user. Knowing what we know now, how can we, as best as we can, facilitate the features we've mentioned for the given user? Knowing that this will be a multi user system we can identify that managing everything on the user's computer is probably a bad idea, and that we'd probably want some more central, coordinating component. We won't dive deeper into that, as that is all we require for a functioning interaction on a more abstract level. So now we imagine a typical interaction. A user wants to reserve some nodes for an experiment, launch an experiment on these nodes, and get the results of this experiment. Figure 1 is a sequence diagram illustrating such an interaction with the central, coordinating server. Also showing what we imagine to happen behind the scenes to begin getting a grasp of just how this system might be shaped.

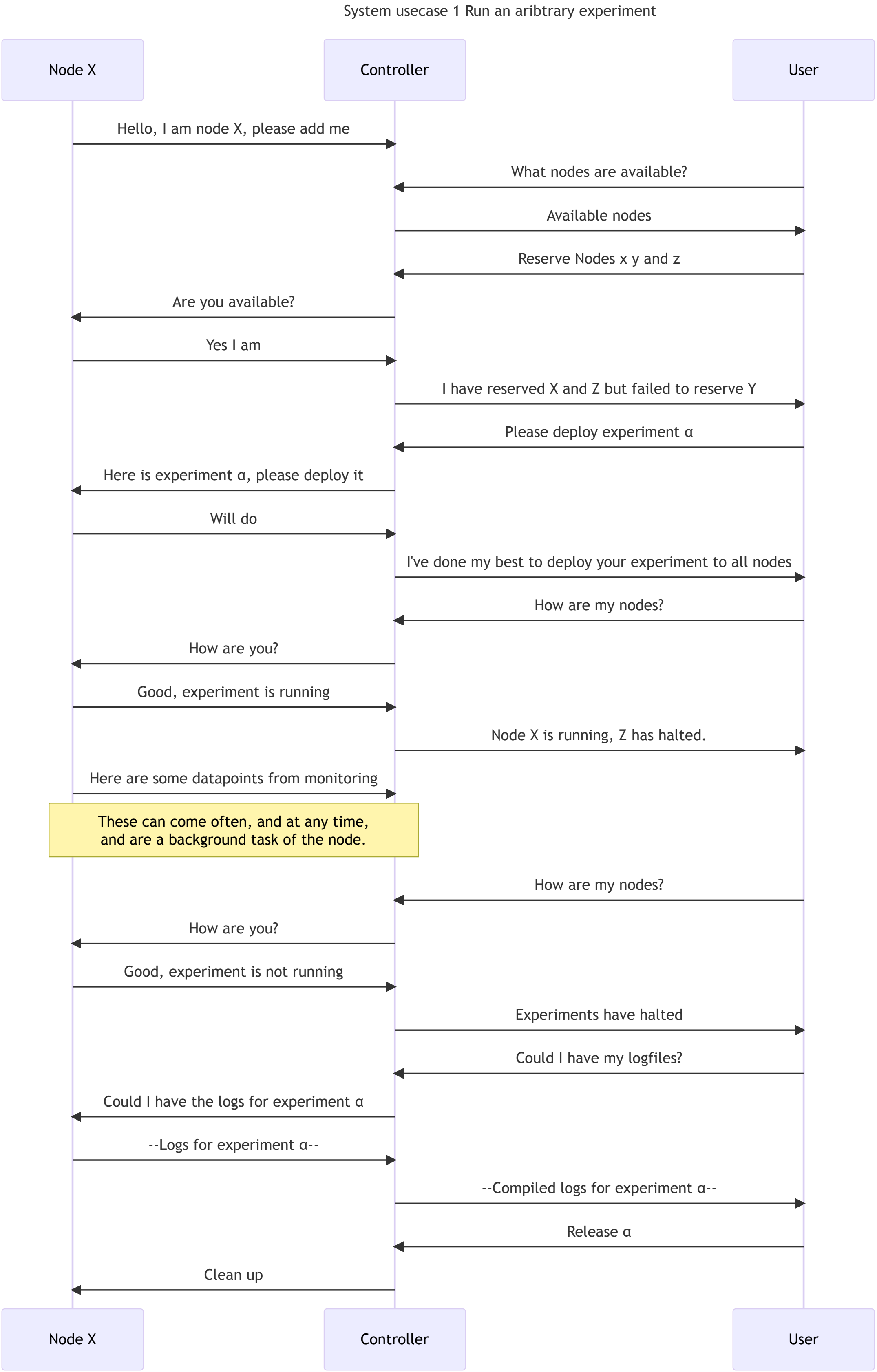

System usecase 1 Run an aribtrary experiment
Node X
Controller
User
Hello, I am node X, please add me
What nodes are available?
Available nodes
Reserve Nodes x y and z
Are you available?
Yes I am
I have reserved X and Z but failed to reserve Y
Please deploy experiment α
Here is experiment α, please deploy it
Will do
I've done my best to deploy your experiment to all nodes
How are my nodes?
How are you?
Good, experiment is running
Node X is running, Z has halted.
Here are some datapoints from monitoring
These can come often, and at any time, and are a background task of the node.
How are my nodes?
How are you?
Good, experiment is not running
Experiments have halted
Could I have my logfiles?
Could I have the logs for experiment α
--Logs for experiment α--
--Compiled logs for experiment α--
Release α
Clean up
Node X
Controller
User

Figure 1: Sequence diagram showing the imagined usage pattern running an experiment

Studying figure 1 we see the imagined usage pattern. We'll study this pattern simply. Starting the procedure the nodes initiate a call to a controller, announcing their existence, registering themselves. The user then proceeds to reserve the nodes they want for their experiment. The user, once they know their reservation is in place, deploys an experiment to the nodes. As the nodes run they send back statistical data to the controller, and the user can ask the controller about the status of their experiment. Once the user identifies that the experiment has finished running they can gather log-files and statistics from their experiment, and finish off by releasing the reservation on the nodes.

This design already satisfies some core requirements of the testbed, and doesn't immediately raise any red flags as to issues with other requirements, so we continue work using this idea as part of our foundation.

4.1.4 Analyzing it all together - Identifying a design philosophy

The above sections provide a rough overview of the challenge at hand, but do not provide us the tools we need to begin tackling this challenge. Though with this background we now have what we require to begin identifying a design philosophy. It will be the tool we need to get started breaking down our problem.

We know that implementing every single feature specifically will become a nightmare. Every new combination of hardware and software needing lots of code to be written means that it will become an unmaintainable mess. This raises the question; How can one avoid this problem? The answer is indeed quite simple: Abstraction. Specificity in how our system works with every component will be the bane of the testbed, so instead of being specific, we are vague. Instead of trying to specifically support every feature, we need to support a superset of features that can be used to build a more specific subset of these features.

This idea of building an abstract set of layers unlocks the true superpower of this architecture: Reallocation of responsibility. By creating these abstractions we no longer worry about what happens underneath the layers we create. We focus only on creating a higher level of abstraction that facilitates the logic needed for our requirements. How these requirements are satisfied in the below layer of abstraction is not something we care about. This in turn gives us what we call the "black box" design philosophy.

4.2 Design philosophy

In the previous section we've identified what we call a *black box* design philosophy. But what is it, and how is it used? Here we'll cover, in coarse detail, how it is used, and lay the foundation for how it can be integrated into a functional IoT Testbed.

4.2.1 What is a black box?

The black box architecture is our first step out of the groundwork, and into the work of properly designing the testbed system. So what is a black box? Merriam Webster defines a black box as:

> Black Box - Noun
>
> 1 : a usually complicated electronic device whose internal mechanism is usually hidden from or mysterious to the user
>
> broadly : anything that has mysterious or unknown internal functions or mechanisms [5]
>
> [...]

In context of our computing system we can consider it to be a collection of completely separated components. The inner workings of these components do not matter to anyone outside of the black box. All that matters is that the inputs, and outputs of the black box follow a certain predictable pattern. We do not enforce how they should work on the inside, we only set expectations for how they should work externally. This allows us to completely distance ourselves from the worries of specific implementation details of the internals of a system. This does not reduce the size of the problem, but it breaks the solution to it into many bite sized chunks which can be tackled efficiently. Also reducing the amount of intertwining between components, helping keep the solution cleaner, and less complex.

In the design of this testbed we imagine the client, controller, and nodes as black boxes. None of them should need any understanding, or make any assumptions about the inner workings of any other component. This means that the controller no longer needs to take a stance to a thousand different node types, it only needs to take a stance to a large set of abstract nodes, which behave the same externally, but can have completely different implementations internally. Effectively highly reducing the complexity of the system from all sides.

4.2.2 The problem of specificity

While this design choice appears to be a good solution to the major problem we face, it also introduces new problems. Not necessarily problems that makes this design choice unfeasible, but problems that we must identify early so we can take them into account in our design. That way we can proactively work towards using this design choice to make a good architecture, even with possible limitations that it might impose. The largest issue of this design choice is that of specificity. Just how specific should the interfaces between our components be? Let's illustrate the problem with an Analogy.

Cars offer a high level of abstraction in that they provide you with a steering wheel, pedals, and a gear shift mechanism. Abstracting away the complex mechanics underneath. Creating something that you can universally interface with. This high level of abstraction being just what we are looking for, as it allows our drivers and cars to be highly intercompatible. However as developers we not only want to be able to drive the car, we also want to control the more specific features of the car. ABS, ASC, Climate Control, etc., and this is where the higher level of abstraction becomes a problem. The solution is to simply add more buttons and controls to the dashboard that allow you to interface with, and control the more granular, specific parts of the car. However if these controls become too specific, all of the sudden the controls will cause issues when a user switches from one car to another. The specific interfaces of the prior car suddenly make them no longer fully compatible with any type of new car, without retraining.

If we look at the car as our node, and the driver as the controller, all of the sudden we loose the entire point of having a black box architecture if we become too granular with our interface. If we give our black box too many node specific inputs and outputs. We can solve this by creating a wide array of inputs and outputs that are standard for all nodes, and all controllers. However this offers issues too. Now suddenly we've made it very difficult for whomever is designing the nodes to follow all these specifications for their, possibly very different, hardware. As such it is important that we balance on a very fine line of giving the system enough inputs and outputs, while not enforcing too many.

4.2.3 Generic is good

To tackle the problem discovered in the previous section we need to keep to a lesser set of interfaces. Fewer things for everyone to worry about. But how do we still give our developers the powerful set of tools that they require? We take a generic approach to our problems. Instead of enforcing hundreds of different, specific mechanisms, we enforce a few large, generic mechanisms. Creating a larger super-set of features that we can flexibly reuse to implement more granular controls. This is the second superpower of this testbed. It enables us to create not only powerful tools in the first place, but also means that we have a good foundation for further expansion on top of the already present mechanisms. Meaning that this testbed not only will support the current identified features, but has a great potential to support any future, upcoming features.

4.3 Architecture

Knowing our general design philosophy we can now utilize it to create a coarse grained system architecture. This architecture design will be the cornerstone of the testbed's further implementation.

4.3.1 Macro organization

As mentioned shortly before, the system is intended to have multiple end users, and as such, coordinating the system through client machines is going to be difficult. As such we need some form of centralized control layer. Centralized here not necessarily meaning not distributed. This centralized control layer is what we'll call the Controller. The controller is a black box component of the system responsible for facilitating the required functional mechanisms for the client to interact with as to meet the feature requirements. The controller node will be the hub of communication for a set of nodes. These nodes are again black boxes of varying types of hardware, software, network topology, etc. The clients talk to the controller, and the controller mediates their requests to the relevant nodes in their reservation.

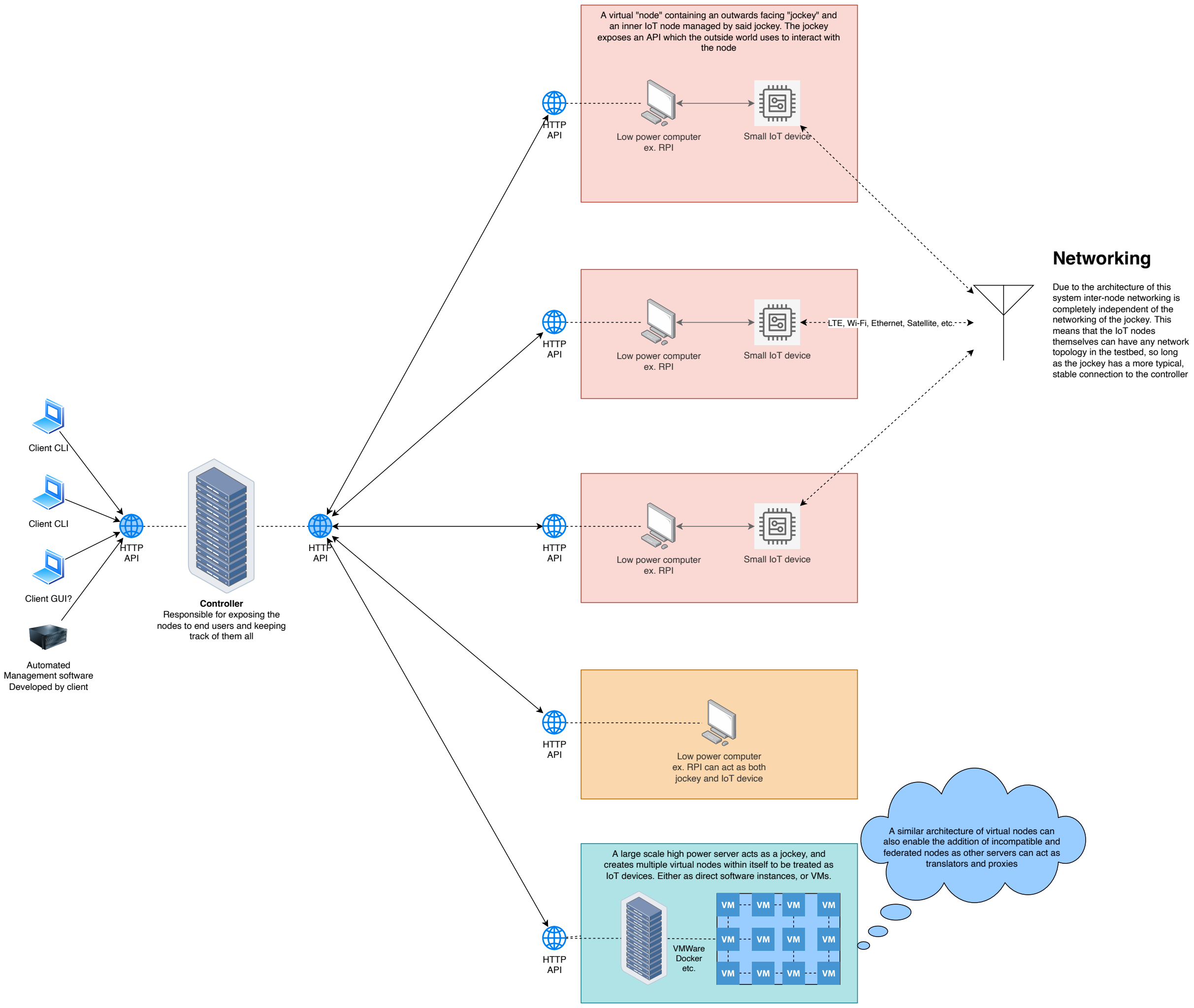

A virtual "node" containing an outwards facing "jockey" and an inner IoT node managed by said jockey. The jockey exposes an API which the outside world uses to interact with the node
HTTP API
Low power computer ex. RPI
Small IoT device
LTE, Wi-Fi, Ethernet, Satellite, etc.
Networking
Due to the architecture of this system inter-node networking is completely independent of the networking of the jockey. This means that the IoT nodes themselves can have any network topology in the testbed, so long as the jockey has a more typical, stable connection to the controller
Client CLI
Client CLI
Client GUI?
Automated Management software Developed by client
Controller
Responsible for exposing the nodes to end users and keeping track of them all
Low power computer ex. RPI can act as both jockey and IoT device
A large scale high power server acts as a jockey, and creates multiple virtual nodes within itself to be treated as IoT devices. Either as direct software instances, or VMs.
VMWare Docker etc.
VM
A similar architecture of virtual nodes can also enable the addition of incompatible and federated nodes as other servers can act as translators and proxies

Figure 2: System Architecture

Figure 2 shows the system architecture in rather fine grained detail. We'll explain the figure more later, but what matters for now is that, from left to right, you see the rough structure of the system. Client, controller, and nodes.

4.3.2 Controller

The controller serves an important role in the system in that it enables the user to actually make use of the nodes in the system in a coordinated manner. It needs to expose a set of endpoints that allows the client to efficiently communicate actions they wish to perform against the nodes. The controller then needs to mediate these requests to the nodes and compile the results for the user to receive.

The controller's internal structure is not something we care too closely for in the design phase. It can be a large distributed computing cluster that scales up and down to manage thousands of nodes, or it can be a single small computer managing a dozen nodes in some lab.

The common component of all controllers is that it needs to consider the system in terms of experiments and reservations. These two terms are tightly intertwined. The thought is that a user marks nodes down for reservation, and the nodes they successfully reserve they can launch experiments on. The experiment being deployed to every node in that reservation.

The node also needs to provide some generic API for managing nodes in a reservation. This would involve getting logs, monitoring info, and doing administrative things like rebooting the nodes, or configuring them.

4.3.3 Nodes

In this system nodes come in two variants. They come in the virtual and real variant. Real nodes, or just Nodes are referring to the hardware unit that the experiment is running on. Virtual nodes are the node units that the controller sees. Some nodes expose only one virtual node, and other nodes can expose multiple virtual nodes. The machine that exposes virtual nodes might not even be the same as that which hosts the real nodes.

4.4 Feature Overview - How can it be implemented?

Now knowing the rough design of our architecture, and our design philosophy, we can do a final circling back to our requirements. Taking a look at them and studying just how the architecture facilitates them from a initial design standpoint.

4.4.1 Heterogeneity

To enable hardware and software heterogeneity we utilize the black box architecture. What is on the inside of the black box does not matter so long as it can be programmed to obey the interfaces that are required for it to be integrated with the testbed. This can either be done directly on the IoT Node or through some kind of middleware. Meaning that the node which you are actually running your experiments on doesn't have to be the node you are also talking to. Looking at figure 2, the red box can be seen as a black box. This is what you're talking to, without caring for the fact that there's two computers inside. One higher power computer which talks to the testbed, and a lower power computer that runs the actual IoT hardware and software. Being programmed by the higher power computer. This way even the most low powered devices can be integrated into the testbed, theoretically making it support all software and hardware.

4.4.2 Concurrency - Testbed & Node

The testbed operates in the context of experiments and reservations. What this means is that the controller will have concurrency in that it can have multiple reservations and experiments going at the same time, since it is keeping track of them. The nodes can be concurrent in the sense that they can offer multiple virtual nodes that run on their real node hardware. That way they can offer multiple ”experiment slots” for the users to make use of.

4.4.3 Experiment & Node management

Through the black box architecture we can implement a generic interface which allows us to perform the most basic, and possibly even more complex commands against our nodes. Allowing us to manage both their running and future experiments, as well as managing the nodes themselves.

4.4.4 Monitoring & Logs

By using a generalized interface we can create simple, system agnostic mechanisms for retrieving logs and for gathering monitoring data. The origins of this data is not important. It can be a datastream from the com port of some IoT device, captured by a computer, it can be network relayed messages, etc. As long as the server implements the standardized endpoint for gathering these logs.

4.4.5 BYO-OS

This is a more complicated challenge, but can once again be solved through the black box architecture. Nodes can offer certain mechanisms for installing OSes that are standardized in some way. That way the user can provide their own operating system. This is more of a technical challenge however.

4.4.6 Custom networking

Through the black box architecture we imagine that a node with custom networking has two networking layers. The public layer which faces the testbed, and an internal layer. The public layer is a standard connection which is hopefully hard to mess with or interrupt, and the internal layer can be any form of communication technology. The experiments run by the user should be exposed to this internal layer of communication. This internal layer can be anything that the node provider decides. It could be a simple star Ethernet network, it could be a mesh Bluetooth network, or more IoT centred protocols like Zigbee, and more. If you observe figure 2, the red nodes have an interface to their left which communicates with the controller, and another networking interface to the right which the IoT nodes actually talk to.

4.4.7 Mobility

Mobility is less of a design challenge and more of a technical challenge as the coarse grained design of the testbed doesn't necessarily care if a node is mobile or not. What does care however is the actually implemented software. We will get back to discussing mobility later.

4.4.8 Event generation

Event generation is a larger design issue as this is a complex feature which has a lot of different sub requirements. Event generation might have to run on the node itself, or if the node is too low power, on another machine. The event generation has to be configurable in a sensible manner, while also being largely intercompatible. To solve the problem we tackle it, once again, with our black box design. All nodes that support event generation implement an event endpoint. A generic endpoint to which an event generation server can send information about an event. These events can be highly customizeable, or standardized.

With this layout the event generation server can run anywhere that it can reach the node. It also means that the user can configure it, and that the components of the system, in theory, should be highly reusable.

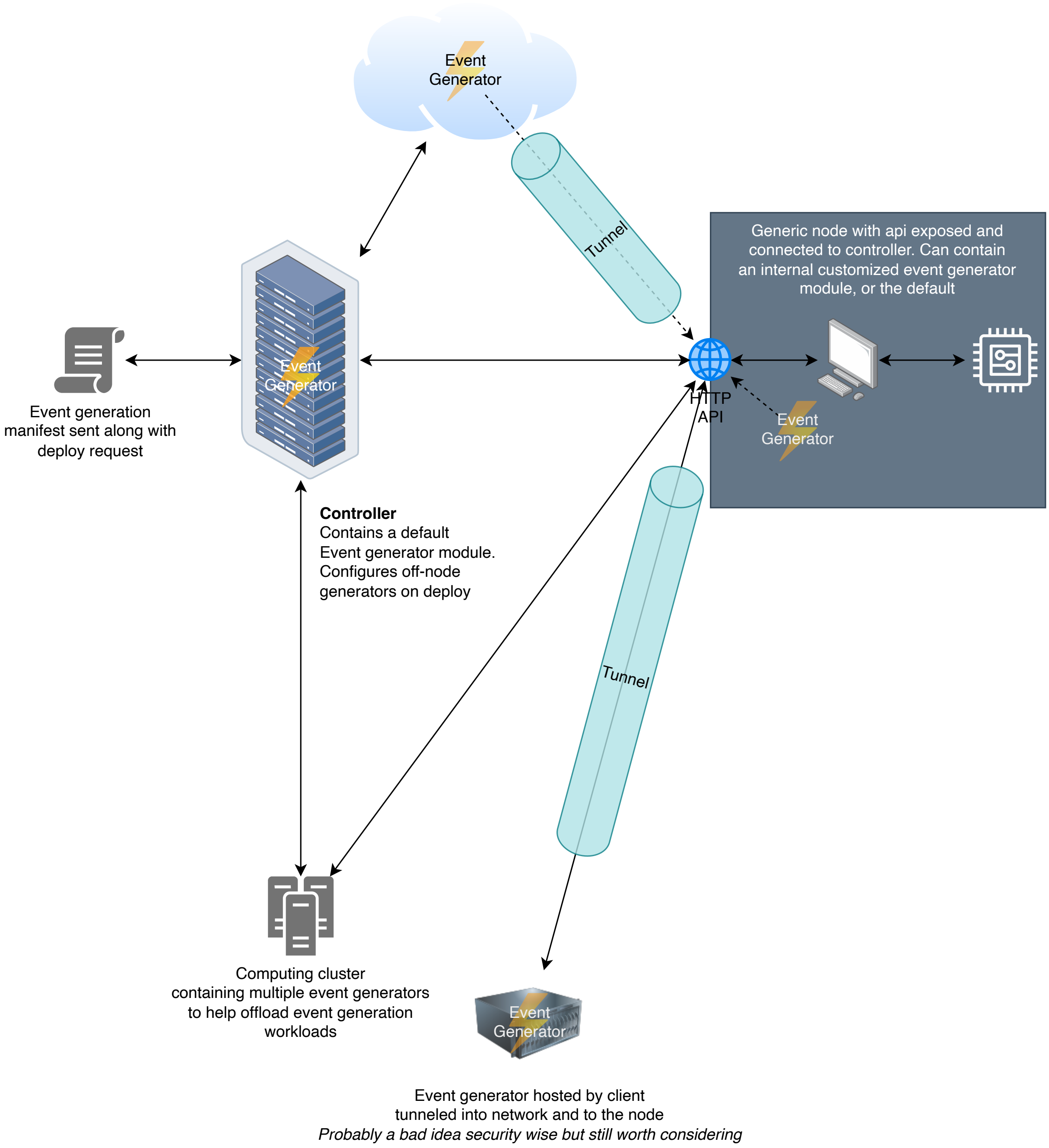

Event
Generator
Tunnel
Generic node with api exposed and connected to controller. Can contain an internal customized event generator module, or the default
Event
Generator
HTTP
API
Event
Generator
Event generation manifest sent along with deploy request
Controller
Contains a default Event generator module.
Configures off-node generators on deploy
Tunnel
Computing cluster containing multiple event generators to help offload event generation workloads
Event
Generator
Event generator hosted by client tunneled into network and to the node
Probably a bad idea security wise but still worth considering

Figure 3: A closer look at event generation

Figure 3 shows the intended design of this feature. Showing the different locations of event generation servers, and how they connect in the context of our testbed. We'll get more into detail about event generation from a technical standpoint later. Note that the event generator in the node doesn't have to be *on* the node itself. It can run on a machine parallel to the node, but which is managed by the provider of the nodes, and configured by the "nodes".

5 System implementation

5.1 Technologies

The choice of technologies of the testbed could simply be one of personal preference, but should also be an educated choice. The testbed should aim to reuse as many existing technologies as possible. Technologies that are widely adopted, and very easy to use and interface with. It should also aim to be implemented in a language that facilitates ease of development, and flexibility. As such the language of choice became Python 3. It's easy cross platform programming allows for fast development of the system across different devices, and an easy push from development to production environments. It also offers a plethora of existing packages to help make development extra speedy. For technology usage Click was used to make the CLI, which is a very handy library that enables the fast development of CLI's. Most importantly, for communication, the HTTP Protocol was decided upon. The reason for this being that the protocol offers a large degree of flexibility. A wide array of builtin features and flexibility meaning the testbed can be created faster. The HTTP protocol was also chosen for it's ubiquity. HTTP has libraries for creating servers and clients for all major languages, and even has tools like CURL that comes with most operating systems as standard software. That way the testbed should be both quick to develop, and effortless to interface with. To facilitate HTTP communication Flask was used on the server side for most components, and Requests was used for the client side components. Most components implement both a client and server side to be able to both send and receive requests.

5.2 The API

The (HTTP) API will be the interface between the black boxes in our system. As we mentioned in the design section this API needs to be generic. We need fewer, more versatile endpoints that can get the job done rather than many specific endpoints. For this we analyzed the usage patterns in figure 1, as well as our feature list and some other use cases, and devised a common denominator API structure that would facilitate all these features in as few HTTP endpoints as possible. The endpoints are listed in rough detail in appendix A.

These endpoints in large part facilitate their communication through JSON datastructures. That way we can have a great deal of flexibility and rigidity. We can devise JSON schemas that can offer validation, while still keeping the data-structures flexible. Endpoints can also vary their datatypes from binary data, text data, specific file formats, etc; and all this is done in a way that is built into the protocol. The different datastructures are described in appendix B.

5.3 System components

5.3.1 Node

The node is a self contained component that needs to implement a set of endpoints required for managing the node. The basic outwards behaviour of the node needs to be the same, and the internal behaviour can vary from node to node. This part primarily describes external behaviours, but will also mention some internal behaviour specific to this implementation.

Startup When starting up, the node needs to announce itself to the controller. Doing so it will provide some important information about itself. This information is provided in a node manifest. See appendix B.1 for an example. In short this datastructure contains the unique identifier of this node, preferably a GUID (Globally Unique Identifier), as well as a hostname and port combination. If the hostname is not provided you should address it by the IP address instead. It'll also send some information about the node's hardware, such as CPU architecture, networking, OS, etc. From here it needs to verify that the controller accepted it's request to join. It is possible that the controller reports a collision, in which this ID is already consumed. If the ID is unique to the node, such as a MAC address, or a GUID, then it can safely assume that it's colliding with itself, and can consider this a theoretical success. If not it should try to re-register with a new ID.

In the case of this node implementation it only sends one node manifest, but if it wanted to present itself as multiple virtual nodes then all it has to do is send multiple manifests with a different identifier.

Experiments The nodes, much like the controller, need to operate in the context of experiments. Each virtual node that it provides should have one experiment slot. When an experiment is started this experiment is started with the context of the reservation. The node should take the supported file type and perform the appropriate action. For the nodes in this testbed an executable file is simply launched, and let run. Though some nodes could accept image files, python files, shell scripts, C files, and anything you could really dream of that the node provider decides to support.

When starting an experiment the controller should forward, from the user, a file called an experiment manifest. This JSON datastructure, an example of which can be found at appendix B.2, should contain information about the experiment. Specifically how to configure the event generation and monitoring for the experiment, as well as other important parameters that can be custom defined by the node.

Reporting In the experiment manifest the user configures how monitoring should be performed. The node specifies which types of monitoring it performs, and the user configures just what monitoring they want performed, on what frequency. The node then performs this monitoring and submits a monitoring datapoint, detailed in appendix B.4. Either individual datapoints, or a set of datapoints batched together. This datapoint is then stored with the controller and is given to the user upon request.

Logfiles on the other hand potentially generate data very frequently and are continuous datastreams. As such sending these as streams continuously would be a troublesome affair. That's why the logsfiles are retrieved upon request from the controller, presumably upon request from the user.

Event generation Each node implements an event endpoint. This event endpoint is called upon with an event descriptor, see appendix B.3.2 for the JSON structure. This datastructure contains information about the type of event, and the parameters of the event. This can be either a standardized event such as powerloss, or can, in certain cases, be a completely custom event. In the case of this implementation the event endpoint is present but does nothing as simulating events right on the root node brings along a lot of problems that could interrupt normal operation of not only the experiment, but the node as a whole. Though the framework required for event generation is fully present.

5.3.2 Controller

The controller software is the mediator of communications between the node and the client. This software needs to keep track of reservations, and needs to facilitate the necessary endpoints to help the user manage the nodes in the system. The controller is interesting because it must only maintain correctness for the API's that the nodes will be calling upon. The outwards facing APIs that the user will use can be adjusted and made more powerful or more broad, but should probably be kept quite standard so that CLIs and scripts will work with the testbed regardless of controller. Though added power-functions that could be nice to have in the context of this testbed could be added. In this implementation however only the required endpoints for the needed features have been implemented.

Keeping track of nodes To keep track of nodes the controller simply keeps an in memory list of the nodes that have reported themselves to the controller. The controller periodically asks the nodes for their statuses to keep a somewhat up to date state on the nodes. This is primarily done to make listings faster as we can cache statuses instead of asking for them on the request, however this is not strictly necessary. The nodes themselves do not need to be aware of their status with the controller. They are simple workers who are simply told what to do, and the state that they are in is largely managed by the controller. This state, in truth, is rather simple as a node is either reserved, or it is not. Experiments are run on the node, but these are kept track of on demand from the user. The node knows if it's running an experiment, but doesn't need to know much more than that.

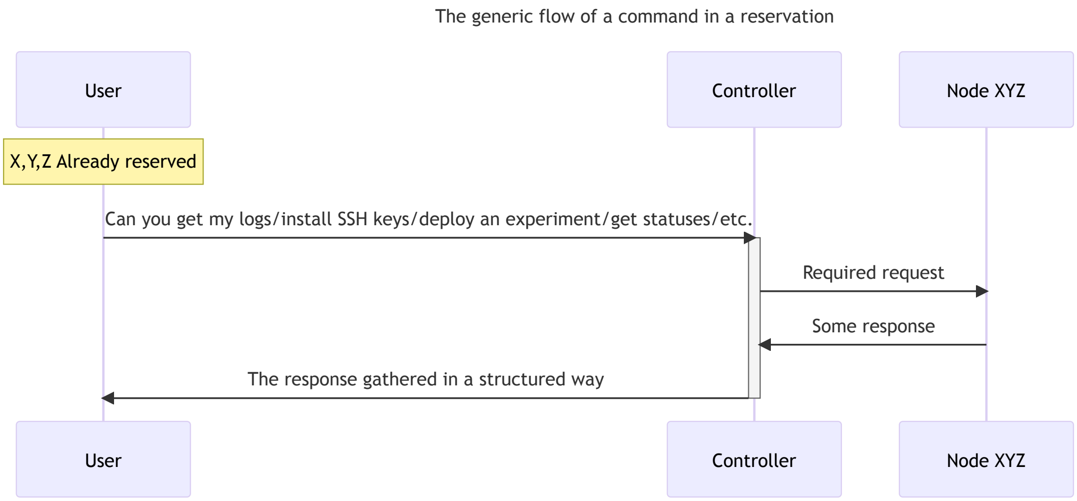

The generic flow of a command in a reservation
User
Controller
Node XYZ
X,Y,Z Already reserved
Can you get my logs/install SSH keys/deploy an experiment/get statuses/etc.
Required request
Some response
The response gathered in a structured way
User
Controller
Node XYZ

Figure 4: Sequence diagram showing the generic flow of a number of commands in the system

Facilitating node requests Figure 4 shows the typical flow of facilitating a node request on behalf of the user. The user performs a request towards the controller which is then simply forwarded, fanned out, to all the nodes in the reservation. The answers from these nodes are then aggregated, and sent back to the user. This is how almost every request in the system is performed, since most types of requests are not asynchronous. Asynchronous requests could have the controller return when it considers the request valid, and has successfully queued up the command. There will probably be more asynchronous requests in future, but at the moment the response from the nodes is what the user is after in almost all scenarios.

All of these requests are tagged with the header ”Request-Is-For”, which includes the unique ID of the node. This means that if a node runs multiple virtual nodes on the same hostname and port, the node can still separate which virtual node this request was actually for.

5.3.3 Event generators

When the user sends an experiment manifest they specify the event generation configuration. This configuration contains controls for performing the event generation in one of two places. On the controller, or on the node. Looking back to figure 3, the lightning bolts signify event generators. All of the generators are either controller, or on-node. The controller simply decides where to keep the event generation servers, and the node too can have external servers for event generation. The controller should support a standard subset of events with a certain configuration, whereas the nodes event generators can support custom events.

In the case of this testbed the event generation is implemented through two components. The event generation server dispatcher, and the event generation server. When an experiment is started the controller checks if generation happens on node or on controller. If on controller then it launches a new generation server on the controller via a request to the dispatcher. Giving it the manifest and the target node. If it is set to on node the controller expects the node to start the server based on the experiment manifest. At the moment nothing happens since there is currently no event generation server on the node. This would be easy to change, as one could simply reuse the generation server, and simply write a new dispatcher to run alongside the node software, or even be integrated directly. This has not been implemented however since event handling would need to be properly implemented on the node first.

The event generation server is a simple component which consumes a event generation manifest and configures the appropriate generators. These generators then pseudo-randomly dispatch the events with the parameters provided. These events, when triggered, are sent as an event descriptor to the target node.

5.3.4 Command line interface

The command line interface for the testbed was implemented in *Click* and was designed to be a helpful tool for the developer of the testbed to test common usage patterns, as well as be a helpful entry tool to those who might use it later. It's in need of some care and attention, but serves as a good entrypoint. The CLI offering multiple modules with almost all the features of the testbed available behind a simple command.

5.4 The Hardware

Here we will detail how the physical testbed was actually set up in our lab. While this is a functioning setup it is important to note that there are many possible ways to solve this depending on the needs of the individual testbed. If you are a large organization like a university with an open testbed that allows devices from other locations to be set up on your testbed then the topology of the network might look completely different compared to what the topology of this testbed is. The hardware is also completely the dealer's choice.

5.4.1 Nodes

The nodes in the current testbed consist of 20 Raspberry PI 4 Model B's. These nodes are running Raspbian OS Lite 64-Bit. They are managed through Ansible, and run the testbed node software through a Python package installed from a local package repository. This package is installed by Ansible, and then configured as a Systemd job which starts at boot. The nodes are powered by a Pi Supply POE (Power Over Ethernet) board.

5.4.2 Networking

The node's networking is facilitated by a Fortinet POE switch. The ports are 1gig each, but due to limitations of the POE board the Raspberry PIs are given 100Mb/s each. This is configured on the switch itself as a limit, and not on the PIs.

DHCP (Dynamic Host Configuration Protocol), DNS (Domain Name System), NAT (Network Address Translation) and Routing was facilitated by an ASUS GT-AC2900. The network layout can be seen in figure 5.

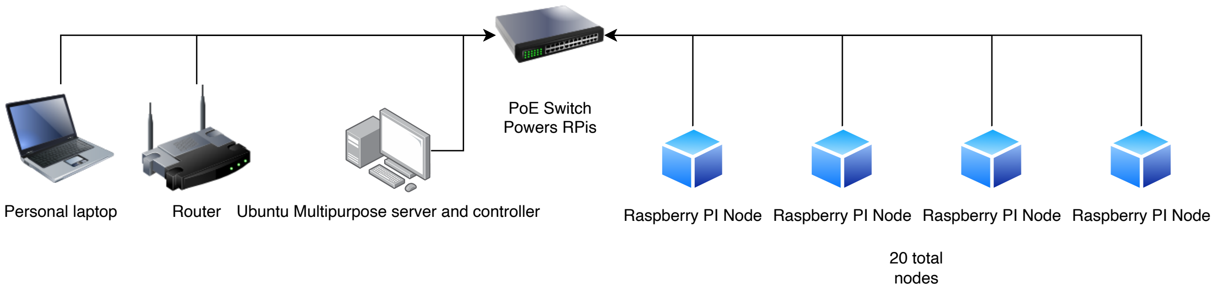


Figure 5: The testbed setup

5.4.3 Controller

The controller is a Generic ThinkCentre M920q Tiny. Running an Intel I5-8500T 6C/6T @ 2.1(3.5 boost)GHz, paired with 16GB of ram. This machine runs the controller software, as well as acting as the Ansible controller which can issue Ansible commands to all the nodes. The controller also runs a pip repository server called DevPi. This repository hosts all the python packages of the testbed as well as mirroring PyPi (Python Package Index). That way all the nodes can reference to this as their package repository, and install the testbed software from it. The controller software on the testbed was similarly configured to the software on the node, added as a service in Systemd.

5.4.4 An experiment in OS management

Prior to using Ansible an experimental approach to administrating the testbed was tested, though not successfully implemented. Though it's a promising concept so it is still detailed here. The process included using PXE (Preboot Execution Environment) to boot the Raspberry PIs off of some central server's Network File System (NFS). Combined with an Overlay File System one could use a simple root file system acting as the basis of a fresh node, and have any work performed by the nodes be written to the overlaying file system. This would make wiping nodes after experiments incredibly easy as you simply wipe the Overlay File System and reboot the node, making it effectively boot exclusively from the fresh underlying file system. However compatibility issues between NFS and Overlay FS regarding file permissions meant that while one could boot into the operating system, the system was effectively rendered useless by permission issues. If one were to have more time to dedicate to this approach it might be a good, viable way of both implementing good features for wiping nodes after experiments, but also Bring Your Own OS features. Although this setup does come at a cost with the file system naturally being rather slow since all disk operations now have to go over the network. Another approach would be to use PXE to start a bootloader and ”OS” which could retrieve OS files from the central server and write them directly to the SD card. The bootloader installed could then simply be deleted and the node would fall back to reinstalling over PXE as a wiping mechanism. However due to time constraints this approach was not tested, but is also one that is likely to help make the testbed more powerful and flexible. These approaches are not a must however. Each node provider can implement their node wiping behavior however they desire. And this is largely considered a feature that should be abstracted away behind the node's black box.

The above mentioned approaches of wiping are only really needed in situations where the user has direct access to the root node. Either through running experiments directly on the root or by having SSH access to it. In those cases such nodes will need to be wiped to remove any possible user tampering after the reservation has been released. Preferably the system's nodes would consist of a two part architecture. A root node and a real IoT Node which the root node controls. Starting experiments, gathering logs and statistics, as well as wiping once it's done. With the user only being given access to the underlying node itself.

6 Experiments

To attempt to measure the viability of the testbed a set of performance and usability tests have been done. These experiments are not meant to be an outwards proof of the testbed's superiority or usefulness, but instead mostly acted as an internal tool to both validate that the implementation was worth developing further, and to help uncover possible issues with the current implementation. Allowing us to identify problems early and underway, and devise ideas as to how to tackle these issues, were they to arise.

6.1 Experiment 1 - Preliminary tests

Initially the testbed's early implementation was tested in a local Docker environment on a generic laptop. This was to simplify and speed up development as much as possible. To validate that the developed solution in this environment was worthy of promoting to the real testbed, and developing further, a set of tests were done.

The tests performed were simple stress tests that hit the most important endpoints of the testbed hard from a client perspective. The endpoints tested were those that performed listings, reservations, and deployments. For the purpose of these tests the nodes were simple, hollowed out HTTP servers that replied with a simple response, without performing actions. This is to isolate the controller as the primary tested component, since node implementations can vary a lot from system to system. To make up for this however the experimental nodes were given the ability to introduce artificial delays to all requests to help uncover weaknesses in the architecture.

The tests were performed for a set of node numbers, being: 10, 50, 100, 150, 200, 300, 400, 500, 850, 1000. For each of these node numbers the test was performed five times to provide an average and standard deviation. The tests were simple requests to the listings, deploy, and reservations endpoints. Measuring how much time it took to complete the requests.

The tests were performed for four different configurations, being combinations of two factors. With/without an added network latency of 50ms, and before/after adding concurrency to the HTTP requests.

6.2 Experiment 2 - Performance tests on the real hardware

This second set of tests were performed after moving from the dockerized environment to the real environment. This test was to see how the system compares to the virtual environment, as well as adding a new test metric to see how the controller performed under high stress from the nodes.

The experiment consists of two tests, and two resulting metrics. The nodes would register themselves with the controller as a given number of virtual nodes. This meant that if the number of nodes was set to 100, across 10 physical nodes, they would present 1000 virtual nodes for the controller to keep track of. This registration of virtual nodes was timed by the nodes. Taking the average time to perform this request. Note that this was done in parallel, so 500 requests could be running at the same time. This average, with it's standard deviation, is one metric.

The second metric was gathered after registrations were complete. Then the client would perform listings of all these nodes. In this case the listings were performed 5 times, but due to a bit of oversight only the average of the values were taken, and the standard deviation was not calculated.

These tests were done for a range of nodes being 10, 50, 100, 200, and 500. This means that, with 19 active nodes (one was prone to failure so was left out) the experiments would scale from 190 to 9500 virtual nodes.

6.3 Experiment 3 - System overhead

Low monitoring and management overhead is one of the requirements identified in the background, so getting some metrics on this to see where we were standing in this regard was interesting. This was done via a set of tests, which can be broken down into two parts.

The first part was a baseline test of the system's overall CPU usage. This CPU usage was derived by a shell script running a periodic TOP command on the process ID of the node software to obtain it's consumed CPU percentage. This is measured as $\frac{cputime}{runtime}$. So the amount of time the process has used the CPU for, divided by the time it's been alive. 100% would mean that the process has utilized the CPU to it's fullest the entire time it's been running. 50% would mean that it has only spent about half of it's lifetime actually computing, etc. This test would give us the knowledge of the system's usage in a normal use case.

The second test was performed with monitoring configured to be very aggressive. It was configured to gather 100 datapoints every second. In this test we performed two measurements. We looked again at the same CPU usage metric, but we also used a program called powertop to get an estimated power usage in watts. Here we also performed multiple experiments with the monitoring to see what difference it would make batching the datapoints into 10 points per request to the controller, vs sending everything, vs. not actually sending anything.

All these experiments were run across all the nodes, and the values gathered by these nodes were averaged to give us as accurate of a result as possible. Since this was a running testbed experiment we also had to use some kind of an experiment binary to run on the node. This binary was a simple blocking binary that halted for most of it's 600 seconds of operation. That way it would not affect our results in any way, showing us only the overhead of the testbed system itself. This binary is largely the same across all experiments where an experiment was run in any capacity.

6.4 "Experiment" 4 - Testing system flexibility by implementing something new

To know how flexible the testbed actually is feature wise we decided to do another "experiment". We decided to implement a feature requested by a PHD student and identified as a feature which would be lacking in a production system: SSH support. Being able to implement this as an afterthought without much effort would prove that the system is, indeed, flexible enough to handle future needs as they arise.

6.5 Experiment 5 - User tests

Lastly to get insight into the usability of the system we had a PhD student test the system's commands and usability in a somewhat realistic manner. The student was given a quick demo of the system, and a markdown document documenting the command line interface, the testbed, and how to use it in theory. This offered lots of useful feedback and uncovered a plethora of bugs. The results of which will be detailed in the results section.

7 Results and Observations

7.1 Product

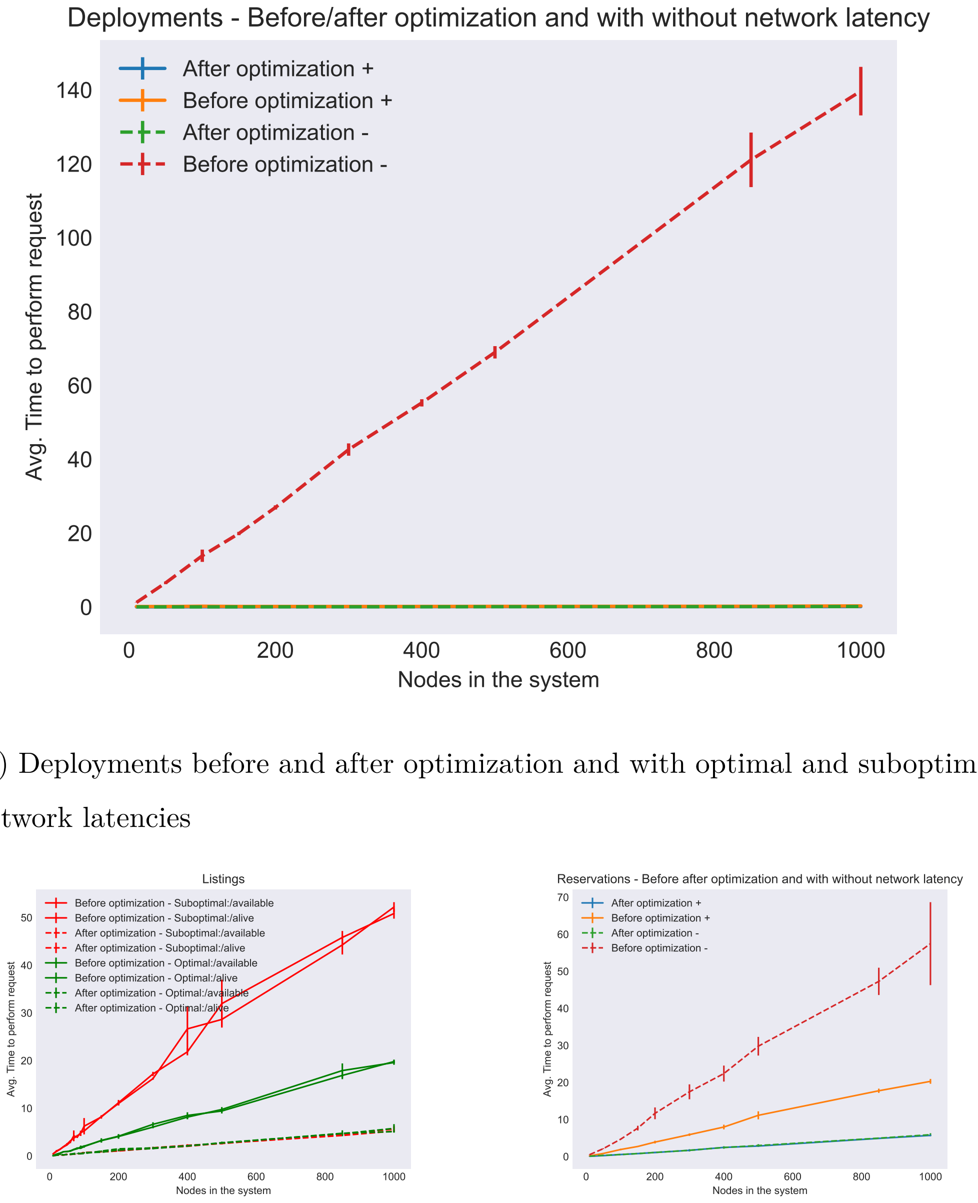
The resulting product of the project is a largely functional testbed, and a seemingly solid architecture for it. The biggest contribution here not being the software itself, but the ground work that has been done in designing a testbed architecture, and the lessons that have been learned both designing this architecture, and through the implementation of it. In the end we are left with a system that theoretically supports every feature identified as a requirement. Built on top of an architecture designed for flexibility. Meaning we have a system that not only supports every feature now, but hopefully also will be able to support every future feature with some degree of tinkering. Table 2 details the features and their statuses. Fully implemented meaning that it is present and working in the software today. Partially supported meaning that it isn't fully developed, but has important parts in place. And theoretically supported meaning that the architecture takes it into account, and that it can be implemented, but isn't in place currently.

Feature	Status	Comment
Concurrency	Fully supported	
Heterogeneity	Fully supported	Node providers must take into account how they integrate their hardware with the system
Node Management	Fully supported	Possibly limited but expandable
Experiment Management	Fully supported	Might require more features down the line. Separate frontend?
Node Concurrency	Fully supported	Virtual nodes
Monitoring	Fully supported	Highly flexible and configurable
Logs	Fully supported	Possibly inefficient
BYO-OS	Theoretically supported	Experiment deploy could point to image through URL, or it could be uploaded, though that could be slow.
Custom Networking	Partially supported	Providers can add nodes that have custom network architectures. This could be expanded to be customizeable but should be explored further
Mobility	Theoretically supported	Use middleware
Event Generation	Fully supported	Customizable to a large degree

Table 2: Features, and their statuses

7.2 Experimental Data

7.2.1 Experiment 1 - System performance in virtual environment



(a) Deployments before and after optimization and with optimal and suboptimal network latencies

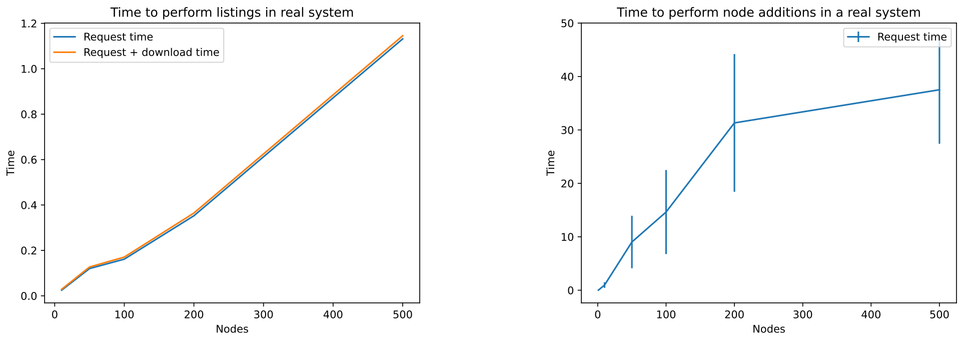
(b) Listing endpoints alive, available, and all, before and after optimizations, with and without network latencies.

(c) Reservations before and after optimizations, with and without network latencies

Figure 6: Tests performed in the virtual, pre-production system.

The figures above 6 were produced in the virtual Docker environment as a preliminary architectural test to see if the base architecture was something that would be worth developing further. The optimization refers to a threading of requests performed by the server to reduce the time wasted by waiting for blocking web requests. Optimal and sub-optimal networking refers to the virtual nodes adding an artificial network latency of a few milliseconds. The small addition of latency can, in some cases, have significant impacts due to the sub-optimal implementation's blocking nature.

7.2.2 Experiment 2 - System performance on the real hardware



(a) Time to perform listings in the real testbed with virtual nodes

(b) Registration latencies for virtual nodes in the production system

Figure 7: Tests performed on the "production" testbed system.

The two graphs 7 were produced by running tests on the real production testbed. 7a tests the availability listing call, as this is the most intensive, and refrains from performing the other calls for simplicity. The graph has two plots, one timing only the request itself, and the other timing request plus download. 7b shows the registration latencies of a large number of virtual nodes. The nodes of the testbed all attempt to register a number of virtual nodes from 1 to 500. The time it took for each of these registration requests to complete on average is on the Y Axis. The number of virtual nodes per node is the X axis. The standard deviations are also marked. These are very high for reasons we'll come back to in the discussion section.

7.2.3 Experiment 3 - System overhead

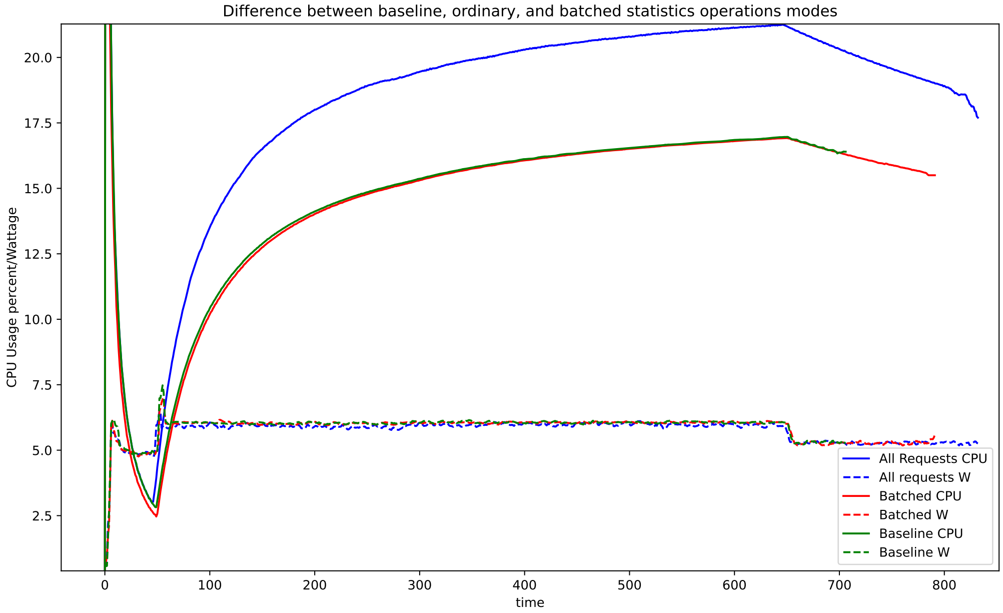


Figure 8: Measurements of power and CPU utilization during an experiment with high monitoring requirements. Dashed lines are watts, solid are CPU usage in percent

The above figure 8 is derived from observing the estimated power utilization of the Raspberry PI, and CPU usage of the node software running on it. The curve shows the initial startup being quite CPU intensive, before relaxing to not utilize a lot of CPU. An experiment is then started. The experiment itself is a blocking executable that should not affect results. The experiment on the other hand demands a high degree of monitoring. Attempting to get 100 samples every second. As a result the CPU and Power utilization can be seen spiking upwards. The CPU utilization converging on approximately 20% before the experiment ends.

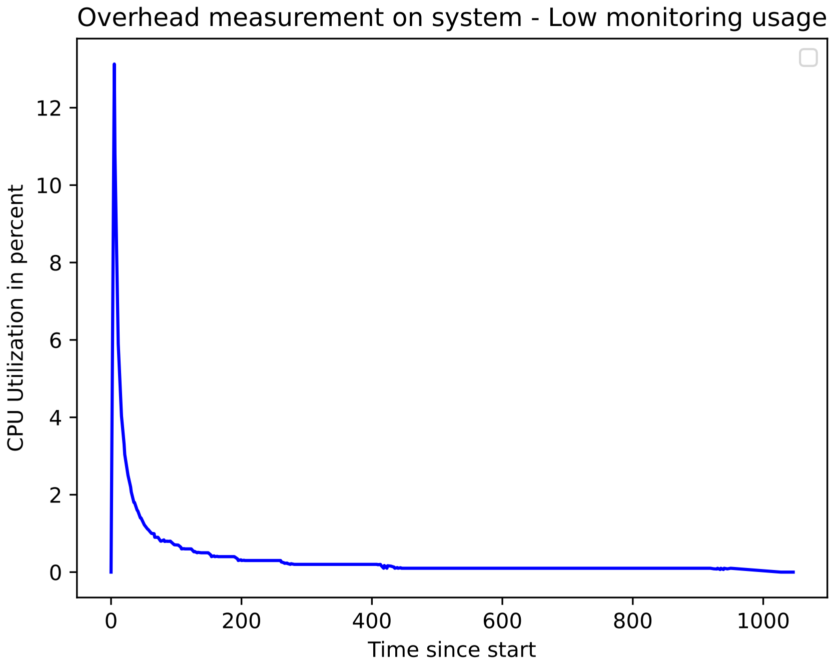


Figure 9: CPU utilization of the system - Baseline

The above figure 9 shows the baseline CPU usage of the node application. This is effectively the same setup as in 8, but without the heavy power monitoring of powertop, and without a very high polling rate of the monitoring system. In this case it only performs a poll every second. This graph shows a high initial spike of CPU usage, similar to figure 8 but even as the experiment starts the utilization of the system converges on about 0.1% usage.

7.3 Experiment 4 - Testing system flexibility

Testing the flexibility of the testbed we implemented SSH after the architecture had been designed and implemented. SSH unfortunately was a bit of an afterthought, but turned out to be a great way of testing the flexibility of the system. The implementation took no more than a couple of days and ended up being rather simple. All components in the system had to undergo changes as this was an end to end change, but these changes were made by only modifying the components implementation of the architecture, and not the architecture itself. The node gained a new endpoint for installing SSH keys, the controller gained a similar endpoint forwarding such keys to nodes in a user's reservation, and the CLI gained a command to find and send their public key to the controller. The node manifest also got a new field informing about SSH compatibility. An optional field that is defaulted to false. That way the system maintains complete backwards compatibility with the previous versions of the software, and offers the ability to SSH into nodes so long as the user is on the same network as the nodes. Something they can be through either a VPN or a proxy of some kind.

7.4 Experiment 5 - User tests

The user tests proved very valuable in uncovering the usability of the software. The user, overall, did have a positive experience with the testbed. The user managed to reserve nodes, deploy experiments in the form of shell scripts, install their SSH key on the nodes, and SSH in. The user could also get their logs and statistical data, and once they were done, they could release their reservation without issue. Meaning that, end to end, the system does manage to do it's job in accordance to the imagined use case 1. Though with this system being an initial prototype, it also follows that there are some issues that need to be tackled.

Initially the software proved itself hard to use due to the simple fact that Python 3.10 was used for development, but Ubuntu, the operating system of the choice for the user, only had 3.8 as it's latest distributed package. As a result the user had to jump through a number of hoops to be able to install the correct version of Python to get the system up and running. This should be a simple pip install, and should have the system ready to go within seconds, but turned out to be an hour long endeavour.

Once the CLI was installed we met another issue. The users's machine refused to use the DNS of the router. This meant that local namespace host resolution refused to work, and the user couldn't access neither the controller or the nodes through their local hostnames. This caused another hour long problem solving session in which the user had to manually configure *resolved* to use the appropriate DNS address for the network. Once this was finally complete the user was given completely free reigns to the testbed to test it out.

This however did not go without a hitch either. The user ended up needing help with the system on multiple occasions, to no fault of their own, once again. This time most of the issues were related to lacking, erroneous documentation, or misleading command names and outputs. In general the developer appears to have become blind to several issues, and having end users utilize the system with little to no context on how things work under the hood has uncovered a lot of areas of improvement where the programs can become better.

7.4.1 Confusion surrounding experiments

The user noted that experiments were difficult to understand intuitively. A large part of this was due to confusion as to just what the term ”experiment” referred to. It could reference the experiment and reservation from the context of the current project. Or it could refer to the active execution of an experiment. This poor wording caused a lot of further confusion when the CLI also had a poor use of the terms, and referred to reservations as ”running”. Confusing the user as to just how the system architecture actually worked.

7.4.2 System design concerns

The user raised some concerns related to the system’s design. Primarily these concerns were related to how the user interacts with the system in the context of experiments. Partially with if manifests should be transferred at the reservation. Though primarily they were concerned about experiment IDs being managed by the client and not by the controller. An understandable concern that should be debated.

7.4.3 System feedback

Another thing the user commented on was the lack of, poor, or misleading feedback of the system. Performing certain requests would give back poor responses such as ”Successfully deployed to 0 nodes.” which could be worded better to be an outright error. In some cases the CLI would be misleading by saying ”Running experiment GUID” even if that was just the output of the experiment ID, and not a status check. In other cases the API just didn’t return proper feedback at all. Failing to launch an experiment due to an invalid JSON experiment manifest file would return ”Successfully deployed to 0 nodes” without any mention that the JSON data had a slight formatting error. This sort of proper error reporting is something that is repeated many times throughout the application.

7.4.4 Usability issues

The user also reported having significant issues with the usability of the system. Things like returning more useful result data, making the listings functions more useful, making reservations take comma separated lists and not newline ones, having the ability to manage multiple experiments at the same time, having the controller remember it's state between restarts, using more human readable formats than GUIDs, or avoiding exposing GUIDs to the users, and more. The user also noted that when starting an experiment, if it fails, should give the user some immediate feedback to the failure, and should give them the ability to roll back the execution of the experiment on the other nodes. The user also noted that the management commands were not properly implemented. Further the user describes a lack of user accounts and workspace management.

8 Discussion

8.1 Graph Analysis

Starting by analyzing figure 6 the most important takeaways of the graphs are that the differences between linear and concurrent execution are massive, and that after added concurrency the difference between low and high network latency is negligible for even a large system of nodes. What this implies is that the system is heavily I/O bound, and that the system is spending a lot of it's time blocking and waiting for requests to complete. Added concurrency allows the system to be far more efficient. This is natural because the many listings the system does is just sending HTTP requests to all the nodes to check if they are alive and returning the results to the end user. As such this is not a CPU intensive task, and neither is it very network intensive, but the fact that the program blocks and awaits an answer can choke the performance heavily.

Looking at figure 7a, it tells a more positive story compared to the virtual system. With 500 nodes **per** node, meaning a total of **9500** nodes (only 19 were functional during this test) the listings take only about 1.2 seconds. Hence, with a number of nodes 9.5 times larger than before, listings were 8 times faster, with similar listings taking 10 seconds before. How come? Most likely this is due to the distributed workload. Rather than a virtual environment with limited networking resources on one node, we now distribute the virtual node workload over many nodes. There is no reason to suspect that the virtual workload was ever CPU bound, but rather I/O bound, probably due to limited packet rates/sockets/etc. in the network interfaces. This proves that listings are potentially quite rapid, especially if we were to scale to an even larger cluster of physical nodes. This is also with caching turned on, and with rotary updates continuously running, so the cache is never too stale. All in all this performance is quite acceptable.

On the other hand 7b tells a more worrying story. It displays the average time taken to perform a node addition. For 9500 virtual nodes we see that the average time taken to add a node is 40 seconds! (This is not linear, but parallel). Such a long period of time to perform a simple addition action is an unacceptable amount of latency. However, luckily, it's not all doom and gloom. Firstly, the chance that 9500 nodes are going to want to be added to the system at the very same time is very low. Secondly we know that the controller server isn't very performance optimized when it comes to additions. While the Waitress WSGI server offered a good number of threads for concurrency, this does not help the fact that all the requests relied on the same in memory list. The list is thread safe, but is so through a mutex lock. As a result the performance remains close to linear. This will make the average time required to perform an addition request very high, not because the server is spending a lot of time per request, but rather because each request has to queue for a long time to be processed. Not only that, but each raspberry pi might meet some issues with parallel processing 500 HTTP requests. We might not just be running into performance bottlenecks on the controller, but also the node. To test this further the system should be scaled up to a larger amount of hardware nodes. Regardless the system can be made more distributed to handle large workloads like this, even though it should rarely be necessary.

Next we can look at figure 8. This figure illustrates how heavy the node is on the system with a very high monitoring workload. It unfortunately appears that when under this high load the CPU converges on just above 20% CPU utilization. For comparison we have figure 9. The CPU utilization of the system with polling once per second was almost negligible. This means the overhead of the rest of the system is in fact quite low, and that the monitoring is the heaviest part. Due to the design of the monitoring, this is no surprise. Observations during testing showed that powertop, the tool used for monitoring estimated power usage, would use very significant amounts of CPU alone. This means that a lot of performance is to be gained by making a more efficient monitoring system. Powertop also shouldn't be used in general due to the inaccurate, estimated measurements it takes.

Further, looking at the graph with no callbacks, vs. batched callbacks, vs. all callbacks we can see that simply batching the callbacks effectively is equivalent to no callbacks to the controller. This means that if we separate the concerns into "gathering data" and "sending data" the gathering part is clearly the most taxing component of this system. As such implementing a far more efficient monitor would be the most effective way to reduce stress on the node.

With the above observations we can also conclude that if it weren't for the very heavy monitoring of the software, the software would be practically without any significant overhead on the node itself, which is very promising, and due to the architecture of the system the monitoring can be implemented in ways that are as high or as low performance as the provider desires. A user can request certain configurations, but a provider can, in their documentation, describe what configurations are in spec, and simply ignore requests for monitoring a thousand times a second, if that is something they don't support due to high CPU utilization. As such what truly matters here is the overhead of the system on the node itself. Which we have proven to be negligible.

8.2 Architecture Viability

While the above analysis of the experimental data might seem to offer a mixed bag in terms of positive and negative outlook on the testbed, this is to be expected. It is a preliminary architecture and implementation with little thought made towards performance. In that sense the testbed in fact appears to be performing rather admirably for a version 1.0. It seems to be able to scale to a large number of nodes before slowing down, and the large scale experiments seem to have bottlenecks which come more down to realistic use of hardware, rather than software limits. Though it's hard to tell how the system will scale with a large amount of users. In the end the important takeaway of the data is that while we haven't proven the testbed to be perfect, we also have not managed to uncover any fatal flaw. As such the architecture is, for all intents and purposes, still worth exploring and developing further.

8.3 Room for improvement

While the testbed appears to be functional as it is there are some things we have learned through both developing and testing the testbed that should be taken into account for further development. There are also a lot of identified design flaws which are simply artefacts of learning along the way. Identifying new requirements as you go. Design flaws which, in some cases, might require significant changes to the design of the testbed. As such it's important to highlight these, and iron them out now, to give the testbed a good chance of making it once it becomes a more fully fleshed out product.

8.3.1 User feedback

Firstly we'll take a look at some user feedback to see where the testbed has more room for improvement in the sense of usability.

A large part of the feedback we received was as a result of poorly adjusted expectations. This is a mistake on our part as we had presented the system as a testbed, without explaining the limitations and current design choices. The user was met with the expectation that his would be a truly user facing testbed like the FiT IoT Lab. However the truth is that this system is more like a backend. Meant to have a frontend built on top of it at a later date. This meant that a lot of the user's expectations regarding managing nodes, experiment IDs, logs, statistics, and having a user account, was more in line with that of a fully fledged, public testbed, instead of this internal prototype, meant to serve as a form of a backend to such a system.

This topic should be explored more in the future. Should the testbed be implemented as a layered architecture when providing it to external users, or should it be a monolithic application? The current intention of this implementation is to be a kind of core to a layered system. A backend which offers APIs for managing nodes and experiments, onto which automated systems can be built. Systems like user centric frontends, CI/CD pipelines, test environments, etc.

Most of the rest of the feedback was related to usability. A lot of feedback and issues were caused simply by how the term "experiment" was too vague, and reused for two different concepts. As such we have decided that perhaps the best course of action would be to rename experiments (the kind which keeps track of reservations etc.) to "projects/reservations". That way you would run an executable in an experiment tied to your project/reservation.

Further the user gave feedback on the general usability of the system. This comes in multiple forms but in general boils down to improving feedback to the user, error reporting, and command structure and naming. This is definitely something we need to take a closer look at in future. There should be some effort before the system is made publicly available to make sure that the User Experience of the system is good. Ensuring that the system behaves in a way that users expect, with the feedback it gives being actually helpful. And that it offers the tools needed by user to make good use of the system.

It's important to keep in mind that this is a prototype implementation. As such we were expecting it to have a lot of issues. We have taken the feedback given, and we have highlighted the problems so that future efforts can work towards fixing them and making them better. Perhaps making the system look bad, but in truth, as we mentioned in the results, the user did manage to perform their tasks according to our design. And this should be considered a large victory.

With this user feedback in mind we can now take a look at the more technical design flaws of the testbed, as well as lacking features.

8.3.2 Security

Authentication The controller is not responsible for authentication, at least not at the moment. Perhaps it can support something like JWT based authentication. Though the system is mostly designed with the idea of having another public facing system, a frontend to the testbed, which should handle users, timed reservations, node listing search and caching, etc. The testbed controller should only be directly interfaced in a trusted system. At least for now. Though since the controller is a web server implementing proper authentication, probably with an external authentication provider, should not be too big a problem. As long as the APIs remain the same the system should be completely interoperable.

Running Arbitrary Code The system runs arbitrary code. As a result it's important for us to take into account the code's authenticity, and think about what we do with the code to prevent security happenings. Code should not be run as trusted users on end nodes, and should preferably be run completely separately from the root node. If the Root node and the worker node are the same, then the code should be run in a virtualized, or otherwise separated and secured environment.

Node trust A system in which many providers can contribute their hardware poses challenges related to authenticity. What nodes are actually real, and which ones are not? In this case we propose using public key cryptography and trust chains to verify the legitimacy of a node. Example: The University of Mars decides to provide a dozen new nodes to our system. The university of Mars, UoM, has securely provided us with their public signing key some time prior. We have this key as a trusted source in our system. All the new nodes get subkeys, signed for their unique ID. When adding themselves to the testbed the node proves its legitimacy by signing it's requests with it's public key. We then validate that this public key is in fact signed by UoM, and can then be sure that the request is valid.

This security architecture is a rough idea and should be validated for attack vectors. Admittedly this feature is not the most important to implement right away either as this would only be a problem in a public contribution system.

8.3.3 Performance

While the system appears performant and is snappy with a small selection of nodes, we would also like to create a system that would be able to scale up to thousands of nodes without significant issue. As a result some suggestions are already on the table for how to possibly improve the architecture of the system for later.

Status management - Filtering and Status on Demand The current implementation of the system continuously updates the status of all nodes, and stores this information in a cache. This is to reduce time to load when the user queries all available nodes. However this isn't strictly speaking necessary. The system is largely event driven, and as such, only needs data in a reactionary way. Though spending a long time querying many nodes for their availability, with many possibly timing out, is also not acceptable. A possible solution to the problem would be the controller making available a list of all types of nodes and configurations. The user then deciding what type and configuration they want, and querying the availability of these nodes exclusively. That would make the query a lot faster to perform by simply reducing the number of nodes to ask. If the user also specifies the number of nodes they need available the system can stop searching once it has found enough nodes. The system could also be restructured in such a way that listings are only meant to list node types, and that the user can register a reservation for a node type rather than specific nodes. The user can then, asynchronously, check what nodes have been added to their reservation. Overall there are many ways of making this part of the system more efficient.

Data structures Currently the system relies heavily on JSON as it's datastructure of choice. While this datastructure will continue to work well for loosely structured documents like the experiment and node manifest, JSON is not necessarily the best choice for strictly structured, or tabular data. For data such as logs and monitoring datapoints, JSON quickly becomes a cumbersome datastructure. Wasting many bytes of data to possibly display a single byte of information. And this is for every datapoint. As a result, high frequency datapoint gathering in the JSON format will cause a lot of overhead translating the format. Also creating a lot of unnecessary bandwidth usage. The resulting very large JSON files can also be hard and time consuming to parse and process compared to raw binary data. The reason JSON is currently in use is because it allows for very quick changes to the data format being transferred. One datapoint can contain any number of named fields, and these named fields can vary from point to point. This is not something a format like CSV, or raw binary will easily allow for. However there probably is some better way of handling the data structures and mechanisms to make the system far more efficient.

Distributed systems - Nodes and Controller Sometimes users might want to run their nodes as a distributed system, or use them to test networking protocols. The current implementation of the testbed has a distributed component in the sense that the nodes can act as a distributed system. However the current implementation doesn't make this very easy for the user to utilize. They would have to manually code, before startup, what nodes are present in their reservation to enable distributed communication. That or they'd have to create their own discovery protocol. Furthermore there is no current way of policing the usage of the network to prevent cross talk between reservations, or to isolate nodes. These are features that should be explored both in terms of how they could be implemented, but also how useful they would be, and how to make them useful.

The testbed could also be distributed in another sense. Currently the controller is a single central component. However as the system possibly becomes more distributed, not only in the sense of scale, but also in the sense of geography, having the ability to distribute controllers would be helpful. Due to the nature of the controller's API and black box architecture one could theoretically speaking implement this distributed system however they want. A possible way of doing this would be to create a shallow tree architecture, in which a root node would coordinate internal, and leaf nodes. The leaf nodes each having a cluster of IoT Nodes. And since users most likely want to reserve nodes of the same type, and each cluster likely has their own types of nodes, the chances are that this won't inhibit usability even if the network is fragmented to a degree. Figure 10 shows a possible distributed architecture. In this case we see a architecture in which

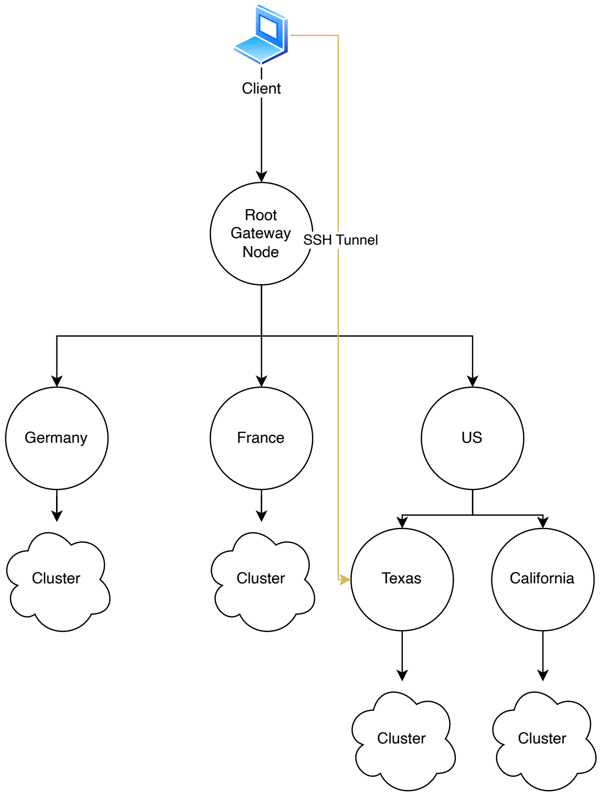


Figure 10: A possible distributed architecture

the user has a high degree of abstraction to the nodes and the cluster. The root node simply forwarding requests to the other datacenters and gathering their answers. Though more advanced features such as SSH tunneling might require more direct access to the cluster. Facilitating this information can be a bit of a challenge, but can simply be added as a custom field in the node manifest. For example "ssh_gateway" could be used as a field to describe what IP and port to connect to to be able to SSH into the node.

8.3.4 Communication Protocol

This is also slightly related to the topic of performance, but is such a large issue that it gets a section of it's own. The communication protocol decided on for this system is HTTP. But throughout the development of the testbed the question of whether or not HTTP is a good choice for this system kept coming up. First let's revisit the pros of the HTTP protocol.

The HTTP protocol offers a lot of pros in the sense that it is incredibly widely adopted. Not only does every single website today operate using the protocol, but many of these websites also have underlying API's that are written to use the HTTP protocol as well.

As such it's not only viable for transferring Hypertext, but has also proven itself to be useful in performing more general server client requests. HTTP is such a ubiquitous protocol that it has been used for an incredibly wide range of software applications. It comes with all sorts of libraries for virtually any language, and HTTP supporting tools come as standard on most operating systems, not to mention that there often are HTTP debuggers built into every major browser. This incredible ubiquity helps support under the flexible and open nature of the testbed. Allowing people to interface easily with it, with whichever tool they want. Giving them the power to automate important tasks. However while the HTTP protocol has managed to fulfill it's purpose with this testbed, and carried it forth to the end, it has some problems that are important to take into account.

There are two problems of the HTTP protocol. The first is that it is a highly synchronous protocol. This normally isn't an issue as communications in nature often are synchronous. The problem arises when the controller, for example, has to perform a dozen requests to nodes that are all synchronous. This in turn means that the controller needs to open a lot of sockets in parallel, or do them sequentially. One takes up a lot of resources, and the other potentially takes up a lot of time. The other problem of the HTTP protocol is that it needs to continously open a lot of connections. Every request has to open a new socket to somewhere, and socket reusal, at least on the controller side, will be very low, further wasting a lot of performance.

What are alternatives to the HTTP protocol for communication? There probably are many good solutions, and multiple should be explored and tested, but one possible sollution is the AMQP protocol. The AMQP protocol, in short, is a messaging broker which utilizes a publish subscribe pattern. An AMQP server acts as the communication broker, and clients can subscribe to certain channels, or publish to them. [8]. This allows for asynchronous communication of information with queues. This could be useful for managing node states, for example. Or publishing statistics streams. However for more immediate actions like launching experiments, or installing SSH keys, one would probably still need some communication layer that communicates directly with the node, since one needs to actually acknowledge that an action has been performed. One might separate it into actions and data streams. Though all this should be inspected a lot more closely in future.

8.3.5 Quality of life features

Frontend While having a functional CLI that offers many features is good enough for most users, a lot of users probably would prefer to have some kind of frontend to interact with. A web interface that could facilitate many features such as user authentication, multiple reservations, visual statuses, status monitoring and alerts, and reservation timeouts. It could also help with experiment validation, and visually creating experiment manifest templates to help users get their products deployed faster. Such a frontend would probably be built as another layer around the current system. Forwarding requests to the testbed controller and using it's APIs to perform actions on behalf of the system's users.

These kinds of frontends don't only have to be visual representations of the system either. They can also add more powerful features such as CI. Allowing users to tie their CI/CD pipelines to the system to basically use the testbed as a proper CI Test environment.

8.3.6 Further validation

While this thesis so far has created a solid foundation and in large part proven the testbed architecture to be a viable one, the testbed still has only been tested with a small subset of use cases, on very limited hardware. As such it would be a good idea to continue testing the testbed system and architecture with further challenges to validate the claims of this thesis. To ensure that the system is in fact scalable, flexible, and highly usable. This would be done through both expanding and varying the hardware in the testbed. Integrating anything from high power machines to low power IoT devices, and through end user testing. End user testing would involve deploying the system at some location, such as a university, or an IoT Lab. Giving users the ability to be hands on, and hopefully exposing the testbed to a wide variety of different use cases that would expose any large shortcomings with the testbed.

8.4 The other requirements

As we mentioned in the system design section we narrowed down the requirements slightly to keep the design, and this thesis, focused. This section is dedicated to shortly mentioning the ”lost” requirements, and how they are still accounted for in some way in the testbed, or how they could be implemented in the new architecture. The other requirements are as follows:

- Fog Computing
- Edge Computing
- Scalability
- Generic Purpose
- Plug and Play
- Federation
- Light Monitoring and Management Overhead
- Repeatability

The requirements of fog computing, edge computing, and federation are taken care of through our design architecture and our black box design. We can have computing devices anywhere we’d like, and existing testbeds only need middleware to be translated into the new, flexible system. The largest challenge of this is keeping the network topology organized in such a way that devices are reachable from, and can reach the controller. Scalability can be offered through new and innovative implementations of the controller. Utilizing high performance computing, or scalable designs such as distributed compute clusters. Generic purpose is inherent in our new design, as the feature-set is designed to make it possible to integrate almost anything into the testbed, as well as implementing more complex subfeatures from the existing feature set. Plug and play is also accounted for, new devices only need some basic software written for them. Hopefully software that is open and available to be modified. After that they simply need a single configuration as to which server they talk to, their networking, and then they can automatically configure their IDs as they do today. The current implementation can be installed on any raspberry pi node. A single config file can be shared across all nodes and the node will simply work once it’s plugged in. Light monitoring and management overhead is an implicit requirement that we have talked about. Trying to get down monitoring overhead is something we must focus on in future, but is possible due to the fact that monitoring can be implemented in whichever way the provider desires. Repeatability is more of a challenge. However this can be implemented by having event generation servers log to the controller just what events are dispatched, with what parameters, and when. Giving us a kind of ”replay” file that allows one to replay the events generated. Other than that repeatability is up to each node provider to facilitate. Some nodes can be in controlled lab environments, and some might be deployed in the wild. This will be up to the node provider to document.

8.5 Comparing against an existing solution

For reference we will take a look at an existing solution, namely the FIT IoT-LAB. The FIT IoT-Lab has been chosen as the comparison as it is, from our understanding, one of the more widely adopted testbeds on the market (based on being the top Google result for ”iot testbed”), while also being one of the most fully featured testbeds based on table 1. It is also a testbed that is similar in purpose to ours, aiming to be a multipurpose, generic testbed. The FIT IoT-LAB has a similar architecture to the our testbed in many places. However our testbed also has some core differences from the FiT IoT Lab. This section will analyze their common features, what sets them apart, and how these differences offer strengths, or downsides. Feature wise the FIT Lab offers similar features to our testbed. Reservation of nodes, running experiments, and retrieving data and results from these experiments. Offering a wide variety of node types, some supporting mobility and others not. This allows researchers to test a wide variety of scenarios. The FIT Lab widely offers SSH support. Users can SSH into the nodes to configure them, and perform all sorts of actions as administrators of that system. Our testbed only optionally supports SSH.

Architecturally the FIT IoT Lab is very similar to our testbed. It consists of a Control Node which is somewhat equivalent to our testbed’s controller. It then consists of a Gateway and Open Node. The gateway being the equivalent of the testbed’s root node and the Open Node being the actual node that the user can control. The Gateway responsible for making the open node accessible and helping program it. The largest difference here being that the system is highly interconnected and makes assumptions about the hardware end to end. Our testbed’s controller has no concept of an Open Node, and the Open Node has no concept of a controller. The Open Node and gateway nodes can be any combination of hardware and software, so long as it supports the basic feature set that the controller expects from the API. This should give our testbed a step up over the FIT Lab, due to the added layer of flexibility and simplicity. It might also be a drawback, as it limits the integration level between user and end node. Plug and play also seems to be worse in the FIT IoT Lab.

A feature the FIT IoT Lab has which we don't is debugging, which we'll discuss further as this is a very interesting topic. [1]

8.6 Debugging

As we discovered comparing the FiT IoT Lab to our testbed the FIT lab has debugging support. This is naturally a very useful, and important feature of any software development testbed. While largely overlooked it shouldn't be impossible to implement with our testbed.

The core idea for debugging in our testbed would base itself on using an already existing, open, standardized debugging protocol. The underlying debugger could be a specific debugger for any language, but a translation layer can translate the debugging actions to general commands. This is already done in practice today. Microsoft's Visual Studio Code[1] implements a generic debugger UI. This generic user interface works against a standardized protocol named *Debug Adapter Protocol* (DAP). Debuggers for languages like Node.JS, C#, Python etc. do not natively use DAP. But adapters are written for these debuggers. Acting as software translation layers between the generic protocol, and the specific debugger commands. There already are DAP implementations for a significant number of languages[6]. There's also the Language Server Protocol. A standardized protocol which new debuggers use to implement generic debugging directly [7]. Though it's less widely adopted since it would require rewriting existing, well established debuggers.

DAP can be used to enable remote debugging. Allowing us to expose debugging endpoints on our nodes to the end users. That way they can attach their debuggers through their generic UI's, such as VS Code. Giving us a completely node agnostic debugging experience.

[1] `https://code.visualstudio.com/`

8.7 A more complex use case

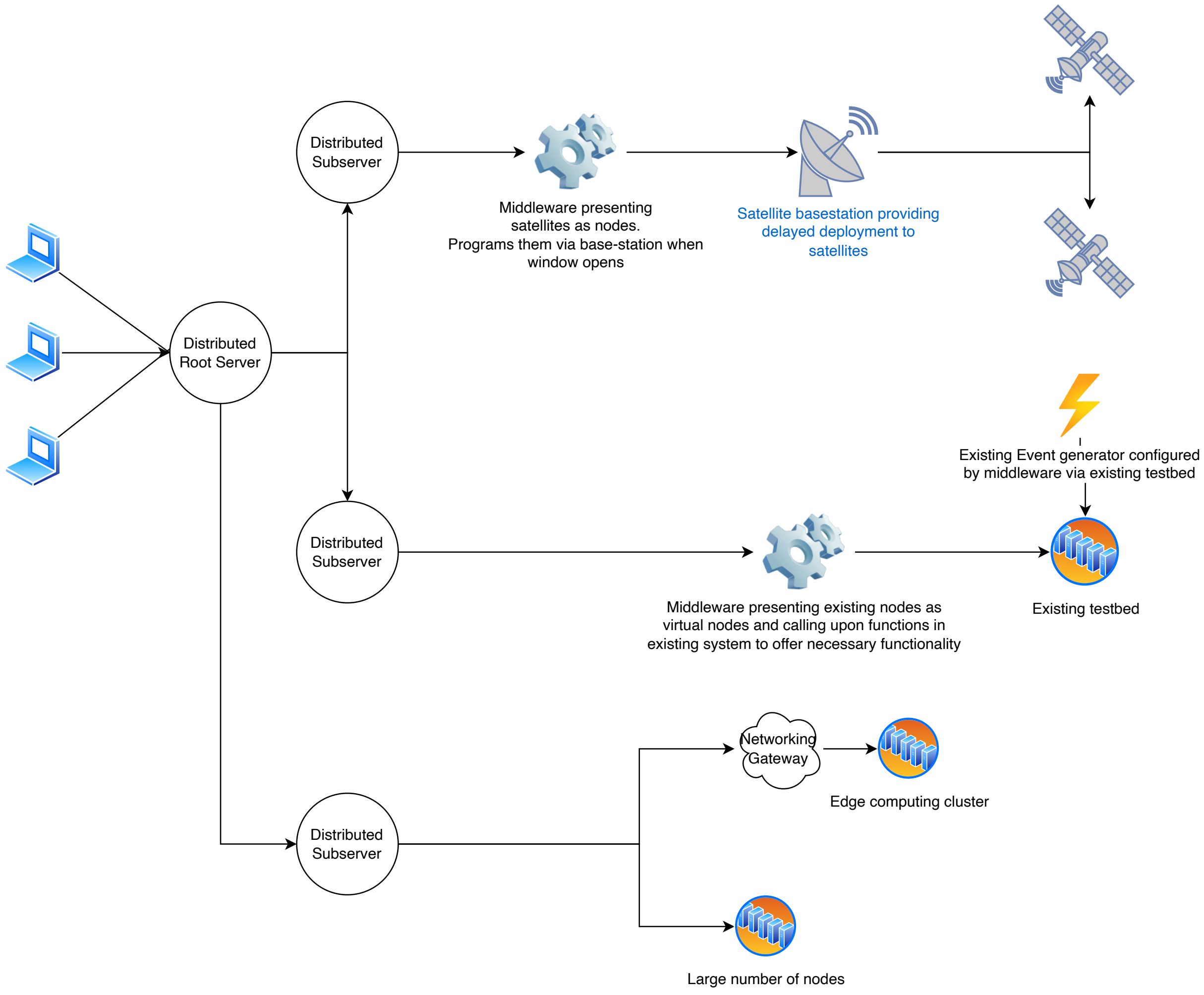

Distributed Subserver
Middleware presenting satellites as nodes. Programs them via base-station when window opens
Satellite basestation providing delayed deployment to satellites
Distributed Root Server
Existing Event generator configured by middleware via existing testbed
Distributed Subserver
Middleware presenting existing nodes as virtual nodes and calling upon functions in existing system to offer necessary functionality
Existing testbed
Networking Gateway
Edge computing cluster
Distributed Subserver
Large number of nodes

Figure 11: A graph showing a theoretical, far more complicated system than the one in place today

Finally, on a bit more of a side note, figure 11 shows a more complex use case of the system, illustrating how the testbed can be used to integrate far more complex, varied systems. In the case of the figure we see a number of clients on the left, talking to a controller. This controller behaves similarly to our controller, but it's source of information isn't nodes directly, but distributed subservers. These servers also act as controllers in the perspective of the nodes, but coordinate with the root controller to keep track of all the devices, and making them available to the end users. On the top of the figure we see a more extreme use case, illustrating the versatility of the system. The distributed server talks to a set of middleware. Software that interfaces with existing hardware and software to make them behave, at least to the perspective of the testbed, as a node. Using existing relays and communication technologies to enable users to have access to satellites. In the case of these satellites commands can't be answered instantly, they need to wait for an open window. In this case the middleware can answer with stale data, and it's up to the user to wait for more recent data to come in. Deployments would be verified for validity, and then queued up. The user informed that their experiment is running, until the middleware can determine that the satellite's experiment has terminated. This is a superficial explanation, but should show the idea behind how it can be implemented, without changing any internal system behaviour. Illustrating the flexibility of the testbed. The system also shows a federated testbed, an existing system not initially intended to be used with this testbed. Though through middleware exposing the existing nodes as new virtual nodes, these nodes can be exposed to our testbed, and be programmed appropriately through the middleware. Then lastly we see a large cluster of nodes, the intended, and initially designed for use case of this testbed, as well as a cluster of edge computing nodes. These nodes could be anywhere on some edge network, and as such might not be directly accessible to the controller. As such they have been patched through into the testbed network via some networking gateway, be it a VPN, Proxy, or some other technology.

9 Future Work

Given more time to work on this project there are plenty of things that should be done to improve upon it. Many of these have been uncovered in the discussion section. This section is primarily a summary of those discoveries, as well as some new reflections.

9.1 Software improvements

The current implementation of the software is to be considered a prototype. The software has had many large changes to it's structure and design over the period of the project, and as such has some loose ends, and needs some refactoring. Furthermore there are a lot of things that needs to be added, such as better error handling, supporting a wide variety of added edge cases, improving logging and maintainability, and adding important features such as persistent storage of data, and a wide variety of security features.

9.2 Exploring what's possible

While this thesis has presented some simpler and more advanced use cases it would be interesting seeing just what sorts of systems can be integrated into this testbed, and in more detail, how. How could we integrate largely distributed edge nodes? Or largely distributed sensor networks with poor network connectivity? How would this testbed design fare with smart cities, smart powergrids, agricultural and health systems, etc. Are there any limits to what this testbed can do? Or have we made a testbed that is truly, fully flexible?

9.3 Expanding functionality, adding hardware

Currently the prototype implementation of the testbed provides what's necessary to add and operate simple nodes, and to meet the requirements discovered in our technical background. A goal should be to make these existing features more powerful, as well as adding new features that could help make the testbed much more functional. Furthermore adding more hardware would be a good next step. Especially hardware that doesn't integrate with the testbed alone, such as a low power Arduino, or similar. Adding these sorts of nodes should be one of the goals for the future.

On the same topic it would be interesting to see just how well one can integrate existing solutions. Testing some form of federation, or how well middleware works in an attempt to add products that are already developed to some degree.

9.4 Polishing and packaging

While the above goals are definitely the most important ones, another goal, which will be very important in the later stages of this testbed, is to polish it, and package it nicely. Make the user experience far better. Here fields like error handling, the interface, and usability should be improved upon. The testbed should also be given the powerful features that end users will come to expect. It's also possible that the system should be given some form of a graphical front-end, probably a web based one. A system which implements features like user accounts, graphical monitoring and configuration, etc. The user feedback from our experiments and results should be taken into account here. At the same time more users should be exposed to the system to get more varied feedback.

10 Conclusion

This thesis has explored the field of IoT, specifically that of IoT testbeds, with the goal of creating a new testbed that implements a sum of all features of the testbeds available today. Hopefully filling in a gap in the market of IoT testbeds today. The thesis inspected what the features of a good IoT testbed should be, based on a quick look at what is out there today. Devising a system architecture and design from these requirements, and creating a prototype implementation of this design. The design implementing all features to some degree, and proving itself highly flexible. The following experiments and discussion found that while the design is a functional first draft there are many things that need to be done to further improve the system's usability and future viability. While the implementation needs work the thesis has presented a solid foundation for a truly flexible, do it all testbed. As well as uncovering the issues that currently exist, and presenting a path forwards for how to solve them.

References

[1] Cedric Adjih, Emmanuel Baccelli, Eric Fleury, Gaetan Harter, Nathalie Mitton, Thomas Noel, Roger Pissard-Gibollet, Frederic Saint-Marcel, Guillaume Schreiner, Julien Vandaele, and Thomas Watteyne. Fit iot-lab: A large scale open experimental iot testbed. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, pages 459–464, 2015.

[2] ARM. What are iot devices. `https://www.arm.com/glossary/iot-devices`. Accessed May 4th 2023.

[3] S. M. Riazul Islam, Daehan Kwak, MD. Humaun Kabir, Mahmud Hossain, and Kyung-Sup Kwak. The internet of things for health care: A comprehensive survey. *IEEE Access*, 3:678–708, 2015.

[4] Bruno Maxime. M1 internship report - characteristic and architecture of a scientific distributed test-bed. Unpublished internal report.

[5] Merriam-Webster.com. Black box definition & meaning. `https://www.merriam-webster.com/dictionary/black%20box`. Accessed April 29th 2023.

[6] Microsoft. Overview - what is the debug adapter protocol? `https://microsoft.github.io/debug-adapter-protocol/overview`. Accessed May 12th 2023.

[7] Microsoft. What is the language server protocol? `https://microsoft.github.io/language-server-protocol/overviews/lsp/overview/`. Accessed May 12th 2023.

[8] John O'Hara. Toward a commodity enterprise middleware: Can amqp enable a new era in messaging middleware? a look inside standards-based messaging with amqp. *Queue*, 5(4):48–55, may 2007.

[9] Jianli Pan and James McElhannon. Future edge cloud and edge computing for internet of things applications. *IEEE Internet of Things Journal*, 5(1):439–449, 2018.

[10] Issam Raïs, Loic Guegan, and Otto Anshus. Impact of loosely coupled data dissemination policies for resource challenged environments. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 524–533, 2022.

[11] Jingjing Ren, Daniel J. Dubois, David Choffnes, Anna Maria Mandalari, Roman Kolcun, and Hamed Haddadi. Information exposure from consumer iot devices: A multidimensional, network-informed measurement approach. In *Proceedings of the Internet Measurement Conference*, IMC '19, page 267–279, New York, NY, USA, 2019. Association for Computing Machinery.

[12] Luis Sanchez, Luis Muñoz, Jose Antonio Galache, Pablo Sotres, Juan R. Santana, Veronica Gutierrez, Rajiv Ramdhany, Alex Gluhak, Srdjan Krco, Evangelos Theodoridis, and Dennis Pfisterer. Smartsantander: Iot experimentation over a smart city testbed. *Computer Networks*, 61:217–238, 2014. Special issue on Future Internet Testbeds – Part I.

[13] Pei Tian, Xiaoyuan Ma, Carlo Alberto Boano, Ye Liu, Fengxu Yang, Xin Tian, Dan Li, and Jianming Wei. Chirpbox: An infrastructure-less lora testbed. In *Proceedings of the 2021 International Conference on Embedded Wireless Systems and Networks*, EWSN '21, page 115–126, USA, 2021. Junction Publishing.

[14] Deepak Vasisht, Zerina Kapetanovic, Jongho Won, Xinxin Jin, Ranveer Chandra, Sudipta Sinha, Ashish Kapoor, Madhusudhan Sudarshan, and Sean Stratman. FarmBeats: An IoT platform for Data-Driven agriculture. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 515–529, Boston, MA, March 2017. USENIX Association.

[15] Nasibeh Zohrabi, Patrick J. Martin, Murat Kuzlu, Lauren Linkous, Roja Eini, Adam Morrissett, Mostafa Zaman, Ashraf Tantawy, Oezguer Gueler, Maher Al Islam, Nathan Puryear, Halil Kalkavan, Jonathan Lundquist, Erwin Karincic, and Sherif Abdelwahed. Opencity: An open architecture testbed for smart cities. In *2021 IEEE International Smart Cities Conference (ISC2)*, pages 1–7, 2021.

A API Endpoints

A.1 Endpoints of the Node

A.1.1 Get node information

`[GET] /`

Get the node's node manifest.

A.1.2 Reset Node

`[POST] /reset`

The node reset call is called upon by the controller whenever a reservation is released. This call should have the node clearing installed SSH keys, cleaning up logs, stats, executables, or even doing a full OS reinstall if necessary.

A.1.3 Install Key

`[POST] /installSSHkey`

Given an SSH key the node should install it as an authorized key for the testbed user.

A.1.4 Submit event

`[POST] /experiment/event/{{experiment-id}}`

Given an event description the node should attempt to perform that event for the given experiment ID.

A.1.5 Start Experiment

`[POST] /experiment/start/{{experiment-id}}`

Given an experiment executable (or any other supported file for this node) and an experiment manifest the node should validate the manifest, possibly the file if not too timely to do, and then respond with an OK status. The experiment does not HAVE to be launched together with the request, and can be pending for various reasons. This should then be reflected in the experiment status call.

A.1.6 Stop experiment

`[POST] /experiment/stop/{{experiment-id}}`

A.1.7 Get experiment logs

`[GET] /experiment/logs/{{experiment-id}}`

A.1.8 Get node status

`[GET] /status`

Should get the status of the node - How is the node doing, how is the experiment doing, are there any experiments running?

A.1.9 Manage node

`[POST] /management/<command>`

This is a generic endpoint which should support a number of different commands as a standard (reboot, clean, etc.), but can also have a plethora of custom commands.

A.2 Endpoints of the Controller

A.2.1 Listings

List nodes

`[GET] /node`

List all nodes the controller has taken note of.

Get live nodes

`[GET] /node/alive`

List all the alive nodes. The status of the node can be cached.

Get Available nodes

`[GET] /node/available`

List all available nodes. The status of the node might be cached.

A.2.2 Experiment

Reserve Nodes

`[POST] /node/reserve/{{experiment-id}}`

The reservation request takes a list of node id-s and marks them as reserved, if they are available. In return it should provide a list of the node id-s that were actually reserved.

Release Experiment

`[POST] /node/release/{{experiment-id}}`

Given an experiment ID the controller should iterate through all the nodes in the experiment, reset them, and then return success only if all nodes were properly released. It is up to the user if they want to try to release it again or if they just move on to a new experiment.

If an experiment is marked as released then the server should continually attempt to reset the node until it has been reset, or it should mark the node as dead eventually.

Install SSH Key

`[POST] /node/experiment/{{experiment-id}}/installSSHkey`

Given a valid RSA public key the controller should fan this key out to all nodes in the reservation.

Get nodes in experiment

`[GET] /node/experiment/{{experiment-id}}`

Get the nodes that are currently in the given reservation

Get node statistics

`[GET] /node/{{node-id}}/stats`

This endpoint is currently not implemented very well. This should be reevaluated as a global endpoint which gets the statistics for all the nodes involved in the experiment, or just a node tied to the experiment.

Add new statistics datapoint for node

`[POST] /node/{{node-id}}/stats`

Nodes post to this endpoint to provide the controller with the recent statistics it has given. The data can be either a list of datapoints, or a single datapoint. Preferably the data should be batched, both for the sake of the node and the controller.

Get experiment logs

`[GET] /node/{{experiment-id}}/logs`

The controller should iterate through all nodes in the reservation and try to retrieve their logs, returning these in some structured manner.

Deploy experiment

`[POST] /node/experiment/{{experiment-id}}/deploy`

Given an experiment, and manifest the controller should deploy the experiment.

Some things are static: The nodes should always be given the experiment manifest, and the experiment. Some however vary depending on the manifest, primarily, event generation: If the event generation location is set to ”controller” rather than ”on-node” this means that the controller will have to spin up an event generation server, provide it with the given config, and point that server towards the node.

This sort of behaviour can be expanded upon in future.

Get experiment status

`[GET] /node/experiment/{{experiment-id}}/status`

Should iterate through all nodes in the reservation and get the statuses for all the nodes in it. Returning these in some structured manner.

Stop the running experiment

`[POST] /node/experiment/{{experiment-id}}/halt`

Given an experiment ID the controller should go through all the nodes in the experiment's reservation and tell them to halt the running of their experiments.

A.2.3 Add node to the system

`[POST] /node`

Nodes call upon this endpoint with their node manifests to describe themselves and ask to be added to the system. There is a possibility that the server returns a collision status, meaning the node is already registered. As long as GUIDs are used in the system the node can safely assume that it is simply trying to re-register itself, having forgotten that it was once registered.

A.2.4 Run Management Command

`[GET] /management/{{node-id}}/some-command`

In the testbed system management commands are generic commands which are sent to a generic controller and node endpoint. Some management commands are predefined and should be installed on all nodes such as "reboot, shutdown, refresh". Though every node can provide it's own management commands. The controller should do it's best to forward the command as it was received to the appropriate node.

B Datastructures

B.1 Node manifests

```
{
    ”id”:”Preferably a GUID”,
    ”port”:9955,
    ”hostname”:”testbed–node–x”,
    ”ssh_available”: true,
    ”characteristics”: {
        ”architecture”:”ARM64”,
        ”connectivity”: [”WiFi”, ”BLE”, ”Ethernet”],
        ”mobility”: true,
        ”nodeType”:”UiT Demo Node”
    }
}
```

B.2 Experiment manifests

```
{
    ”event_generation”: {
        ”generator”: ”on–node”,
        ”events”: {
            ”power_loss”: {
                ”frequency”:30,
                ”period”: 10,
                ”variance”: 10
            }
        }
    },
    ”monitoring”: {
        ”frequency”: 0.01,
        ”parameters”: [
            ”power”,
            ”cpu”,
            ”memory”,
            ”uptime”
        ]
    }
}
```

B.3 Event generation

B.3.1 Event generation specification

See *event_generation* section of B.2

B.3.2 Event descriptor

```
{
    ”issued”:1234567.12345,
    ”generator”:”Default−Generator”,
    ”event”: {
        ”type”:”power_loss”,
        ”parameters”: {
            ”duration”: 13.5
        }
    }
}
```

B.4 Monitoring Datapoints

```
[ //Can be list or just an object directly.
    {
        ”experiment”:”some−experiment−id”,
        ”time”:1680696066.6854775, //Unix time preferably
        ”data”: {
            ”power”:”1.25mW”,
            ”cpu”:”20%”
        }
    }
]
```