
Professorship of Embedded Systems and Internet of Things
Department of Electrical and Computer Engineering
Technical University of Munich

Runtime Deployment, Management and Monitoring of Web of Things Systems

Miguel Romero Karam

Master's Thesis

Runtime Deployment, Management and Monitoring of Web of Things Systems

Master's Thesis

Supervised by Prof. Dr. phil. nat. Sebastian Steinhorst
Professorship of Embedded Systems and Internet of Things
Department of Electrical and Computer Engineering
Technical University of Munich

Advisor Ege Korkan

Author Miguel Romero Karam
Freisinger Landstraße 74
80939 Munich, Germany

Submitted on December 14, 2020

Declaration of Authorship

I, Miguel Romero Karam, declare that this thesis titled "Runtime Deployment, Management and Monitoring of Web of Things Systems" and the work presented in it are my own unaided work, and that I have acknowledged all direct or indirect sources as references.

This thesis was not previously presented to another examination board and has not been published.

Signed:

Date:

Abstract

Internet of Things (IoT) applications have been traditionally programmed using predefined device-level frameworks, tightly coupling software with the underlying hardware. The Web of Things (WoT) on the other hand abstracts interfacing with devices through the WoT Thing Description (TD) standard, allowing to program applications for Systems of Things, referred to as Mashups. In addition to the benefit of being programmed in high-level languages, WoT Mashups can be ported into serialization formats such as the WoT System Description (SD) for better insight and verification of the Mashup. Although WoT readily facilitates the development of WoT Mashups, it lacks a sound mechanism for remote deployment, management, and monitoring of such. In this thesis, a method and its corresponding open-source implementation, the *WoT Runtime Framework*, are proposed to close the development cycle of WoT Mashups. It allows users to deploy WoT Mashups either as code or in System Description format, manage their lifecycle, verify the correct functionality and monitor both runtime and Mashup-specific information. The evaluation proves inter-runtime communication between multiple instances of the WoT Runtime is possible, and demonstrates this with examples from the industrial automation and smart farming domains.

Contents

Abbreviations

API	Application programming interface
CLI	Command-line interface
GUI	Graphical user interface
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
OTA	Over-the-air
RPC	Remote procedure call
SD	System Description
SEC	Script execution contexts
SRD	System Runtime Description
TD	Thing Description
UI	User interface
VM	Virtual machine
W3C	World Wide Web Consortium
WAM	WoT Application Manager
WoT	Web of Things

1

Introduction

The Internet of Things (IoT) has become indispensable across industries and verticals, but exponential growth and lack of standards lead to an ever more fragmented IoT ecosystem. Furthermore, the inherent complexity and interdisciplinary nature of the IoT stack makes its implementation challenging for adopters. Despite an increased offering of off-the-shelf devices to alleviate the complexity from manufacturing through deployment, value-generation only truly begins after devices become operational. At this stage, users are left responsible for harvesting value from their implementations and the lack of interoperability and high-coupling between hardware and software significantly hinders this process.

The W3C Web of Things (WoT) standard by the World Wide Web Consortium (W3C) is intended to enable interoperability across IoT platforms and application domains [1]. It proposes the WoT Thing Description (TD) [2] as the central component to uniformly describe the interaction interfaces of IoT devices or Things, as referred to in this thesis. Analogous to the index.html of web-pages, a TD acts as the single point of entry for interfacing with Things.

1.1 Motivation

An open and community-driven WoT has the potential to bridge the gap between the device and application domains. The W3C WoT concerns itself with the application layer, above the inherent complexities present in the lower-level IoT domain. By doing so it enables programming IoT applications at a higher-level, serving as a unified abstraction for interconnecting Things and Systems of Things, or Mashups. As an extension to the TD, the WoT System Description (SD) [3] introduces a means of declaratively specifying this application logic, bringing composability and serialization to Mashups, as the TD does for Things.

1.2 Problem Statement

The IoT ecosystem has become saturated with devices and manufacturers, each offering diverse solutions to mostly the same problem. Rather than solving the interoperability issue at the last layer of the stack, like the W3C WoT, most apply single frameworks throughout it, resulting in siloed end-solutions. The W3C WoT aims to solve the aforementioned interoperability issue but it lacks a systematic, developer-centric approach to do so. More specifically, the need for an open and incrementally adoptable solution emerges to close the development cycle of the WoT by facilitating deployment, management, and monitoring of WoT applications.

1.3 Goals and Requirements

The goals of this thesis give an overview of what was to be ultimately achieved by means of the developed solution. They enabled a plan to be defined initially in terms of the desired outcomes of the thesis. These goals are:

- **G1:** Facilitate the development of WoT applications to enhance developer experience, make the WoT more accessible, and encourage WoT usage for real-world scenarios.
- **G2:** Design a WoT-compliant methodology and offer a reference implementation as an open, extensible, and scalable software solution.
- **G3:** Ensure solution remains secure and safe for implementation in critical environments with hard requirements such as industrial use cases.

The specific implementation details for the fulfillment of the aforementioned goals came later with the definition of the following requirements:

- **R1:** Remote deployment, management, and monitoring of runtime instances from arbitrary clients.
- **R2:** Exposure over the network to enable consumption of WoT Runtimes as if they were Things, allowing multi-layered architectures.
- **R3:** Monitoring and verification utilities for runtime and Mashup-specific information.
- **R4:** Modular WoT Runtime with its central part being a standalone core module to allow composing robust, loosely-coupled software systems with it.
- **R5:** A reference user interface (UI) client to exhibit core features visually as a standalone and multi-tenant web application.

These requirements shape the research and development of the method.

1.4 Contributions

While the SD facilitates composition and automatic code generation for eventual deployment into WoT Runtimes, doing so is inconvenient for developers in the current WoT ecosystem. This thesis leverages the SD and provides the missing block in the WoT development cycle: a systematic approach for remote deployment, management, and monitoring of WoT Things and Systems applicable to new and existing IoT solutions. An overview of this can be seen in Figure 1.

Parting from the assumption of pre-deployed Things, this thesis introduces a methodology and its corresponding open-source implementation, the *WoT Runtime Framework*, to provide:

- remote deployment, management, and monitoring of WoT Mashups into safe sandboxed runtimes,
- a systematic approach for control and visibility of WoT System Runtimes via a runtime controller in form of a TD-compliant WoT System Runtime Description (SRD), allowing to compose multi-layered architectures,
- observability and verification of WoT Systems via runtime logs, metrics and traces.

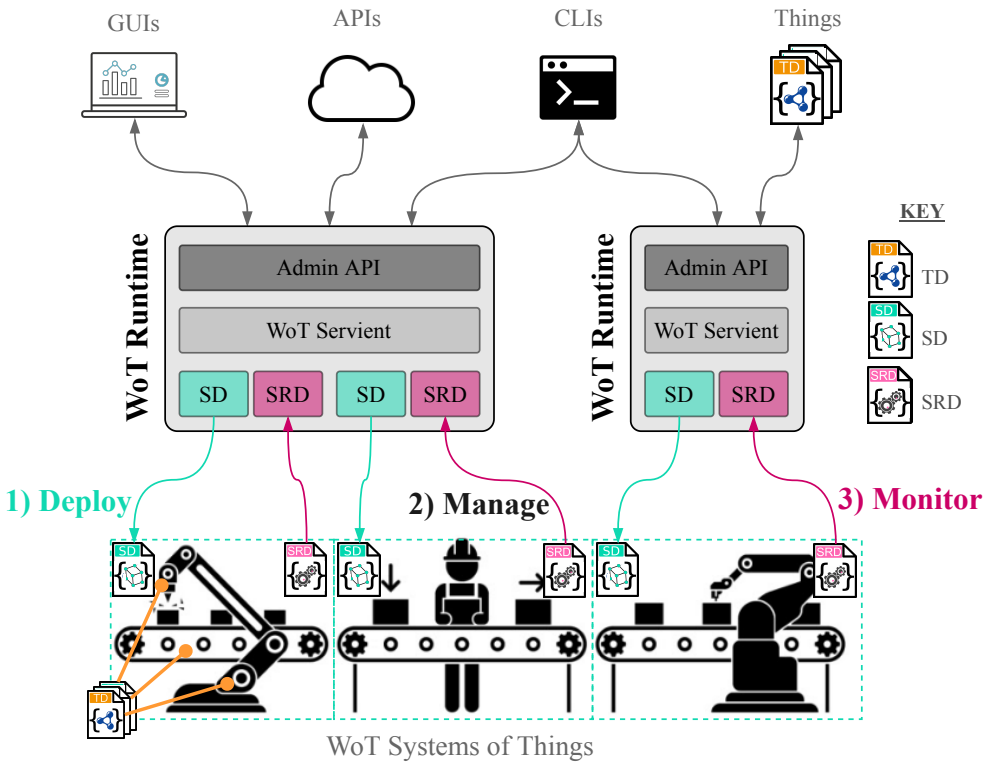

Figure 1: Overview of the proposed *WoT Runtime Framework* with two running instances, each of which can be remotely administered by any number of clients through the admin API, allowing to remotely deploy WoT System Descriptions (SDs) (1), manage their lifecycle (2) and monitor them via the WoT System Runtime Description (SRD) (3).

1.5 Thesis Structure

The thesis starts by introducing relevant WoT aspects in Chapter 2, while Chapter 3 and 4 go into methodology and implementation details respectively. The contributions mentioned in 1.4 are evaluated in Chapter 5 with an industrial automation and a smart farm simulation use case. Related work is then discussed in Chapter 6 and finally, a conclusion and outlook is given in Chapter 7.

2

State of the Art

The W3C WoT aims to facilitate usability and interoperability of the IoT by exposing devices to applications as WoT Things, decoupling them from the details of their underlying protocols and data models. Despite being first conceptualized in 2009 [4], the first official standards, the WoT Architecture [1] and the WoT Thing Description [2], were published by the W3C WoT Working Group in April 2019. The central concept in the WoT is that Things expose their Interaction Affordances and describe them through a TD document, which can then be consumed by other Things or services for interaction. Interaction Affordances can be Properties, Actions or Events and provide a means of modeling the network-facing interface of physical or virtual Things, serializable through the TD.

2.1 WoT Architecture

The WoT Architecture [1] specification dictates terminology and the underlying abstract architecture of the W3C WoT. Its purpose is to provide use cases, requirements, an abstract architecture and an overview of the WoT building blocks and their interplay (see Appendix Figure A.1). The WoT building blocks are standardized in their own W3C specifications, and are the following:

- **WoT Thing Description**
- **WoT Binding Templates**
- **WoT Scripting API**
- **WoT Security and Privacy Guideline**

The following sections provide an overview of the WoT Thing Description and WoT Scripting API specifications, as well as the WoT System Description proposal [3] which are especially relevant for this thesis.

2.2 WoT Thing Description

A WoT Thing Description provides a data format for describing metadata and network-facing interfaces of Things. In the WoT context, a Thing is an abstraction of a physical or virtual entity that provides interactions to and participates in the Web of Things [2]. A TD instance is composed of four main components: Thing Metadata, a set of Interaction Affordances to describe interactions, Data Schemas for machine-readability of exchanged data and Web Links to express relationships to other Things or documents on the Web. TDs are encoded in JSON-LD[1] by default and instances can be hosted by the Thing itself or externally, allowing both resource-restricted and legacy devices to take part in the WoT. A sample TD for a conveyor belt exposing its `status` and `speed` as Properties, the `toggle` command as an Action and the press of the emergency stop button as an Event is presented in partially in Listing 2.1 (see Appendix Listing A.1 for full example).

```
{
  "@context": ["https://www.w3.org/2019/wot/td/v1", { "@language": "en" }],
  "description": "Conveyor belt with a stepper motor and RPi controller",
  "title": "ConveyorBelt",
  "base": "https://192.168.178.194:8080/",
  "properties": {
    "speed": {
      "forms": [{ "href": "properties/status" }]
    }
  },
  "actions": {
    "toggle": {
      "forms": [{ "href": "actions/toggle" }]
    },
  },
  "events": {
    "emergencystop": {
      "forms": [{ "href": "events/emergencystop" }]
    }
  }
}
```

Listing 2.1: Example of a simple WoT Thing Description describing an industrial conveyor belt exposing its speed as a Property, the toggle command as an Action and the press of the emergency stop button as an Event.

1 https://www.w3.org/TR/json-ld11

2.3 WoT Scripting API

The WoT Scripting API is an optional building block in the W3C WoT and provides a convenient way to extend WoT capabilities and implement WoT applications [5]. Scripting requires the ability to run a WoT Runtime and script management, for which gateways or browsers lend themselves well and are thus commonly used. The WoT Scripting API [5] defines an application programming interface (API) to allow scripts to discover and operate Things and expose locally defined Things. The WoT Interface represented by the WoT Scripting API strictly follows the WoT Thing Description specification and provides layered interoperability based on how Things are *discovered* and used: *exposed* and *consumed*.

Discovering Things: Finds Things in the local WoT Runtime, in the network or in a Thing Directory by providing filters and semantic queries.

Consuming a Thing: Creates a local programmatic object which exposes the described WoT Interactions. A Consumed Thing allows the TD to be introspected, to read, write and observe Properties, invoke Actions and subscribe to Events of the corresponding Thing.

Exposing a Thing: Creates a Thing to be exposed on the network based on its TD by generating the corresponding protocol bindings. An Exposed Thing allows adding, removing and registering service handlers for Properties, Actions and Events and emitting Events.

2.4 WoT System Description

The WoT System Description introduced in [3] extends the TD with additional keywords for the description of WoT Systems, or Mashups. By building on the TD format, the SD shares its advantages to provide a textual format for describing Systems of Things, rather than just Things. To do so, the SD introduces a means to specify the execution of interactions and represent application logic consisting of programming structures e.g. `if`-statements, `for`-loops and `wait` commands. The following represent the most important SD keywords for the contribution:

- `interact` - an interaction sequence, or Atomic Mashup
- `wait` - time delay before an execution of a task
- `case` - conditional execution
- `loop` - repeating execution of another `path` array
- `get` - getter for Mashup variables, Properties or values
- `set` - setter for variables or Properties
- `ref` - reference to an Action or function

For the full list of SD keywords refer to Appendix Table A.1.

3

Methodology

In this chapter, a systematic approach for remote deployment of WoT Systems into individual sandboxed runtime environments as well as the methodology used for management of those runtimes and monitoring of Mashup-specific information are proposed. The requirements of this thesis are first established, to later introduce the approach in an abstract manner. The approach is structured in five main steps: deploy, generate, execute, expose, and consume, as illustrated in Figure 2 and listed below:

1. **Deploy:** The method provides a framework for remote deployment of WoT Systems into distributed runtimes as code or in SD format, implemented in the core module of the proposed *WoT Runtime Framework* (see Section 3.1).
2. **Generate:** An algorithm first presented in the WoT System Description [3] is adapted to generate and transpile SDs into source code for the WoT Runtime to execute programmatically. After transpilation, source code is annotated to enable monitoring code execution (see Section 3.2).
3. **Execute:** To enable safe execution, automatically generated code is run in sandboxed WoT System Runtime environments (see Section 3.3).
4. **Expose:** The SRD is proposed as a standardized means for runtime management and both runtime and Mashup-specific monitoring of running WoT Systems, to be exposed directly over the network by the WoT Runtime host (see Section 3.4).
5. **Consume:** Consuming SRDs permits clients to remotely monitor and interact with running instances of the WoT Runtime. This facilitates development of multi-tiered system architectures in which the WoT System Runtime itself, by means of its SRD, becomes another component of the bigger system (analogously to Things and their TDs) (see Section 3.5).

The above steps are detailed consecutively in Sections 3.1 to 3.5.

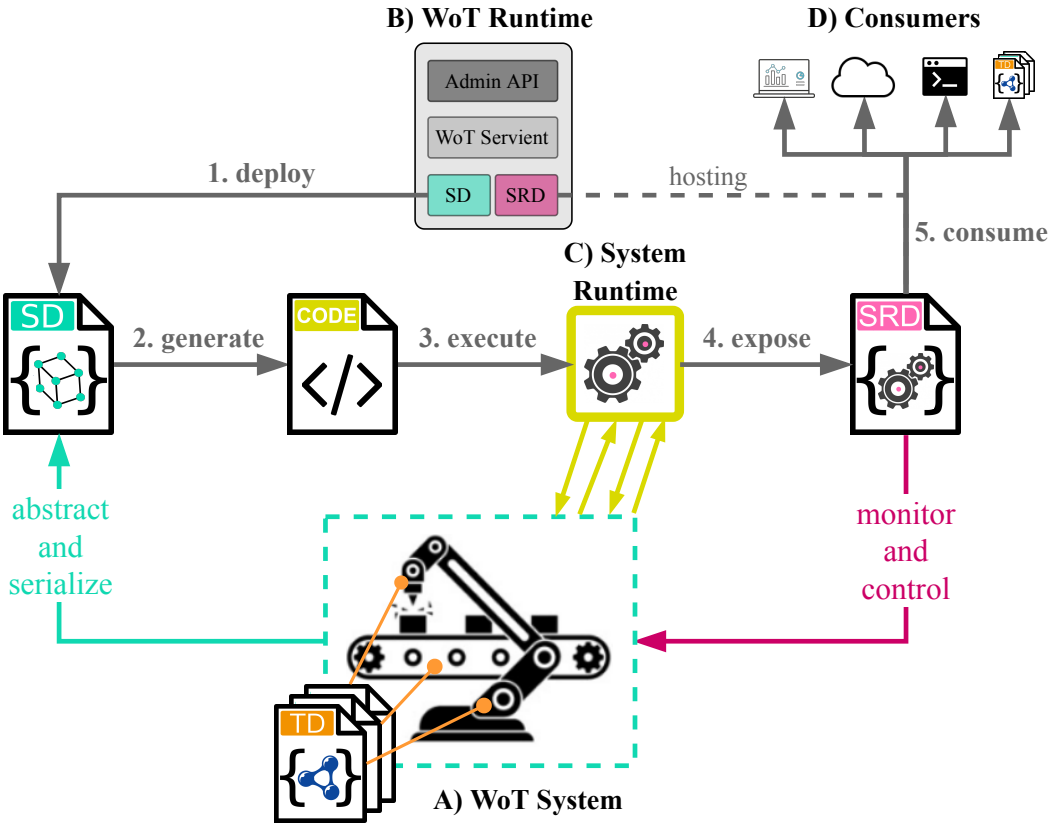

Figure 2: Overview of a WoT System (A) abstracted and serialized as a SD in an instance of the WoT Runtime (B) running in a remote host. Through the admin API of the WoT Runtime, deployment (1) of the SD is made possible, which triggers the code-generation (2) and execution (3) steps. This results in a running WoT System Runtime process (C), which is automatically hosted by the WoT Runtime (B) and exposed (4) as an SRD for clients to consume (5) and thus interact with it, enabling its monitoring and control.

3.1 Remote Deployment

Deploying WoT Systems into secure runtime environments as code or in SD format represents a core aspect addressed by the methodology. While the SD already provides a serialization format for defining Mashups, it lacks a means to deploy these in an automatic and reproducible way. To do so, the core module of the *WoT Runtime Framework* exposes a uniform network interface to allow for both programmatic and over-the-air deployments. By detaching the deployment target (the core module) to the deployment source (any client capable of communicating with the admin API), the framework remains decoupled and thus flexible. Borrowing from well-established REST principles, the approach enables composition of distributed architectures, where WoT Runtimes become another component of the overarching system.

3.2 Runtime Code Generation

Considering how WoT Systems should be deployable both as code and in SD format, a means of generating executable code from SDs is necessary. The algorithm for this consists of 3 steps, code generation, transpilation, and adaptation, further described in Sections 3.2.1 to 3.2.3.

3.2.1 Code Generation

For the code generation step, the methodology extends on the open-sourced algorithm introduced by the WoT System Description[1] [3] to generate and transpile SDs into executable source code for the WoT Runtime to execute programmatically. At this stage, the well-parseable SD logic permits generating highly-optimized code for efficient execution and alignment to WoT Scripting API standard.

3.2.2 Code Transpilation

The code generation algorithm is adapted to take in SDs and output *TypeScript*[2] code as strings rather than as source files. However, *TypeScript* is not directly executable by the *JavaScript V8 Engine*[3] used internally, for which an additional transpilation step is required after code generation. This is achieved internally by porting the built-in *TypeScript* compiler into the core module of the runtime, providing the functionality to transpile strings of *TypeScript* code to *JavaScript*. The entire transpilation functionality is encapsulated in the `transpileTStoJS` function with provided default transpile options, both visible in Listing 3.1.

3.2.3 Code Adaptation

In addition to the code generation step and transpilation steps, a code adaptation step is necessary to permit tracking code execution without polluting the auto-generated WoT System code, neither syntactically nor in its execution. This way, WoT System logic remains true to the originally deployed version. Static code analysis and insertion of asynchronous annotations as wrapper functions, or hooks, are proposed to enable monitoring code execution at runtime. Insertion of these asynchronous hooks remains non-blocking, maintaining the efficiency of code execution.

1 https://github.com/tum-esi/wot-system-description
2 https://www.typescriptlang.org/
3 https://v8.dev/

```
import { TranspileOptions, transpileModule } from 'typescript'

const defaultTranspileOptions = {
  compilerOptions: {
    target: 'es6',
    lib: ['es2015', 'dom'],
    module: 'CommonJS',
    outDir: 'dist',
    alwaysStrict: true,
    sourceMap: true,
    noLib: false,
    forceConsistentCasingInFileNames: true,
    noImplicitReturns: true,
    noUnusedLocals: false,
    strictNullChecks: true
  },
  exclude: ['node_modules'],
  include: ['src/**/*', 'dev/**/*']
}

export function transpileTStoJS(
  inputTS: string,
  transpileOptions: TranspileOptions = defaultTranspileOptions
): string {
  const { outputText: outputJS } = transpileModule(inputTS, transpileOptions)

  return outputJS
}
```

Listing 3.1: Code excerpt used to transpile *TypeScript* strings to their valid *JavaScript* representation. The Function accepts an object of transpile options, with defaults set to transpile *TypeScript* code with ES modules `import`/`export` syntax, and returns a string of code executable by the *JavaScript V8 engine*.

3.3 Execution Control and Environment

Next to secure deployment of WoT Systems to remote hosts, the safety of the WoT System Runtime environment itself is of high priority. Sandboxing the WoT System Runtime in a secure, self-contained execution context is proposed. This prevents code from escaping its execution context into the host, which has the capability of running multiple execution contexts simultaneously. The sandbox runs its own isolated process, side-by-side to the main process of the core module and can also require permitted external libraries and built-in modules. Each process can be monitored and controlled remotely while an internal control flow mechanism guarantees state and state transitions of the WoT System Runtime remain deterministic, as shown in Figure 3.

3.4 WoT System Runtime Description

The TD-compliant SRD (see Listing 3.2 is introduced as a means for standardizing runtime management and both runtime and Mashup-specific monitoring of running WoT Systems. The SRD is dynamically generated and automatically exposed over the network by the WoT Runtime host for each running WoT System Runtime instance. Unique SRD

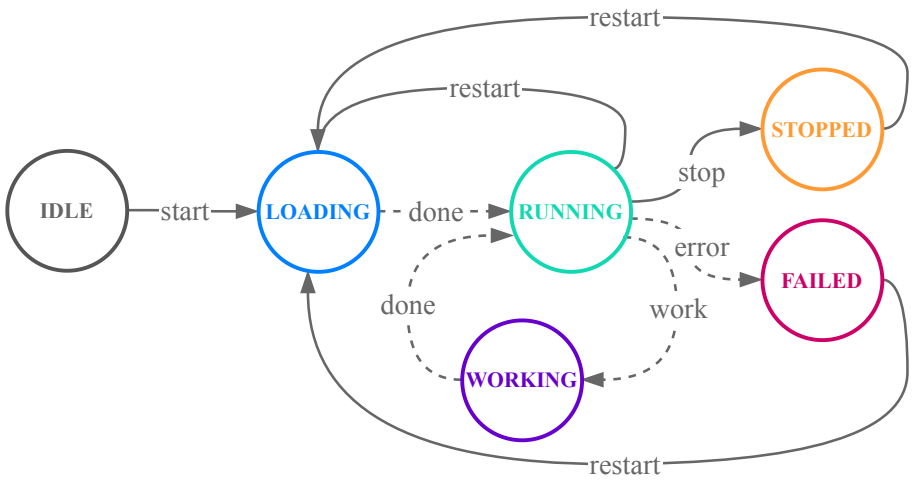

Figure 3: State machine illustrating the possible operational states and transitions for running WoT System Runtimes. Deterministic aspects of the state machine ensure correct states and transitions at all times, to allow for a single-source of remote truth between the actual running runtime and its state, managed by the core module and consumable by clients. The state machine is directly mapped into the SRD, which exposes the current active state via the `status` Property and maps transitions as Events which fire on state change.

instances are created from a common SRD template, by replacing variable values under angled brackets with their corresponding values for each WoT System (e.g. `<systemId>` would be replaced by the ID of the system in the WoT Runtime). The SRD exposes WoT System Runtime information such as `status` and `memoryUsage` as Properties, `start`, `restart`, and `stop` commands as Actions and state changes or transitions as Events, such as `start`, `work`, and `error`. Mapping the entire operational state chart of the WoT System Runtime shown in Figure 2 into corresponding Interaction Affordances in the SRD enables real-time sync between the runtime and its digital representation, resulting in precise verification of operational safety. In Figure 3, state transitions available as Actions for manual invocation appear as solid lines, while internal and automatically invoked state transitions are represented with dashed arrows.

3.5 Consuming the SRD

By fully complying to the TD specification, the SRD (see Listing 3.2) permits its consumption in the WoT context as if it were a Thing. This enables composition of multi-layered WoT applications in which WoT System Runtimes are available for interaction, as shown in Figure 1. Concretely, consuming SRDs permits clients to remotely manage and monitor running instances of the WoT Runtime. Moreover, the serializable SRD format facilitates automatic graphical user interface (GUI) generation for visual interaction in the form of visuals and controls. Thus, the SRD describes a flexible yet capable interface for arbitrary client consumption, such as within WoT applications or

```
1  {
2    "@context": ["https://www.w3.org/ns/td"],
3    "id": "urn:dev:org:wot-runtime:<systemId>",
4    "title": "<systemName>",
5    "description": "WoT System Runtime Description hosted by the WoT Runtime",
6    "properties": {
7      "system": { ... },
8      "logs": { ... },
9      "hostOS": { ... },
10     "memoryUsage": { ... },
11     "resourceUsage": { ... },
12     "uptime": { ... },
13     "traces": { ... },
14     "status": {
15       "type": "string",
16       "enum": [ "idle", "loading", "running", "working", "stopped", "failed" ]
17     },
18     "statusSvg": { ... },
19     "MashupLogic": { ... },
20     "seqD": { ... },
21     "codeTS": { ... },
22     "baseTS": { ... },
23     "code": { ... },
24     "base": { ... }
25   },
26   "actions": {
27     "start": { ... },
28     "restart": { ... },
29     "stop": { ... }
30   },
31   "events": {
32     "start": { ... },
33     "done": { ... },
34     "work": { ... },
35     "restart": { ... },
36     "stop": { ... },
37     "error": { ... }
38   }
39 }
```

Listing 3.2: The SRD is dynamically generated and hosted by the WoT Runtime for each running WoT System instance. Unique SRD instances are created from a common SRD template, by replacing variable values under angled brackets with their corresponding values for each WoT System (e.g. <systemId> in Line 2). The SRD exposes WoT System Runtime information as Properties, control commands as Actions and state changes as Events.

from GUI clients. The SRD is automatically hosted for each running WoT System.

3.6 Observability

For the actual monitoring and verification of WoT System Runtime instances an approach that assimilates that of conventional software is introduced, which borrows from standard observability patterns, namely logs, metrics and traces. Aligning to well-know standards allows monitoring of WoT Runtimes via existing software monitoring methods and tools, such as OpenTelemetry[4]. This further reduces implementation effort

4 https://opentelemetry.io/

and promotes monitoring of WoT Runtimes with a systematic, out-of-the-box approach based on the three pillars of observability [6]: logs, metrics, and traces.

3.6.1 Logs

Logs are immutable, timestamped records of discrete events over time. As such, they are an essential component for software monitoring since they provide the most detailed and in-depth information of the three. Logs consist of predominantly unstructured data since most of them come from logging statements in the code itself, with developers usually providing human-readable messages for debugging. The challenge with logs however, is that despite usual availability, valuable analysis tends to be complex to perform, requiring going through vast amounts of unstructured log data manually.

In the *WoT Runtime Framework*, logs are used to collect as much information as possible from all components in the framework. To facilitate analysis, all logs follow a common format and are pre-pended with the module and component from where they origin (e.g. `[core/worker]`). All logs in the system are collected, stored, and made available under the observable `logs` Property of the SRD (see Listing 3.2). Figure 4 illustrates how logs are displayed in a convenient, real-time table view in the Logs tab of the WoT Runtime UI.

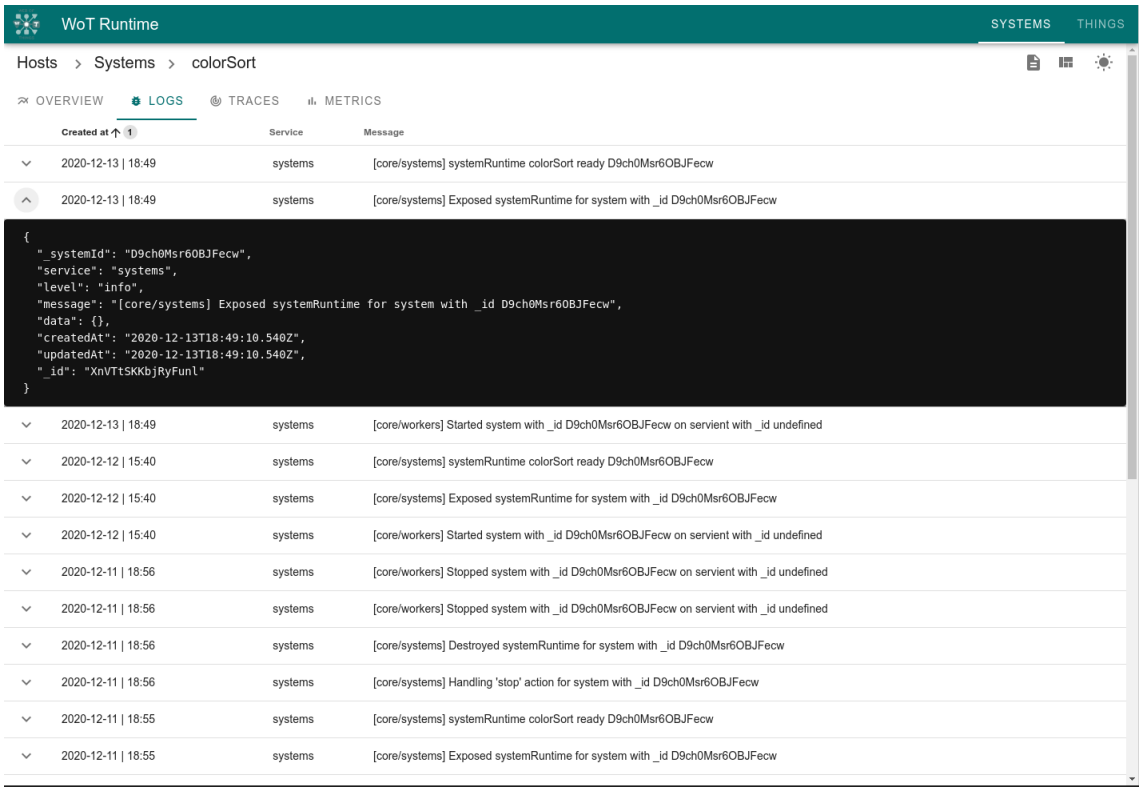

Figure 4: The Logs tab of the WoT Runtime UI provides a convenient table view for log-based observability of WoT System Runtimes. The table updates in real-time with logs coming from the core module over the network.

3.6.2 Metrics

A metric is a numeric representation of data over a time interval. Metrics are usually considered the most valuable of the three observability pillars, as they are efficient, frequently generated and easily correlated. Metrics consist of at least a name, a timestamp and a field to represent a value. Moreover, they enable longer retention periods and easier querying, making them ideal for building dashboards and historical trends and perform mathematical, probabilistic, and statistical transformations such as sampling, aggregation, summarization, and correlation. These characteristics make them best suited to report the overall health of a system. A negative aspect of metrics is that they are system-scoped, making it hard to understand anything other than what is happening inside a particular system.

By virtue of the aforementioned reasons, metrics are used by the WoT System Runtime to convey runtime-specific metrics such as `resourceUsage` (CPU time, filesystem read/write operations, etc), `memoryUsage`, and `uptime` (refer also to the SRD shown in Figure 5, which exposes all WoT System Runtime metrics as Properties. Since all collected metrics are timestamped and of numerical type, they can easily visualized in time-series charts. Figure 5 shows several metrics made available by the SRD as displayed in the Metrics tab of the WoT Runtime UI.

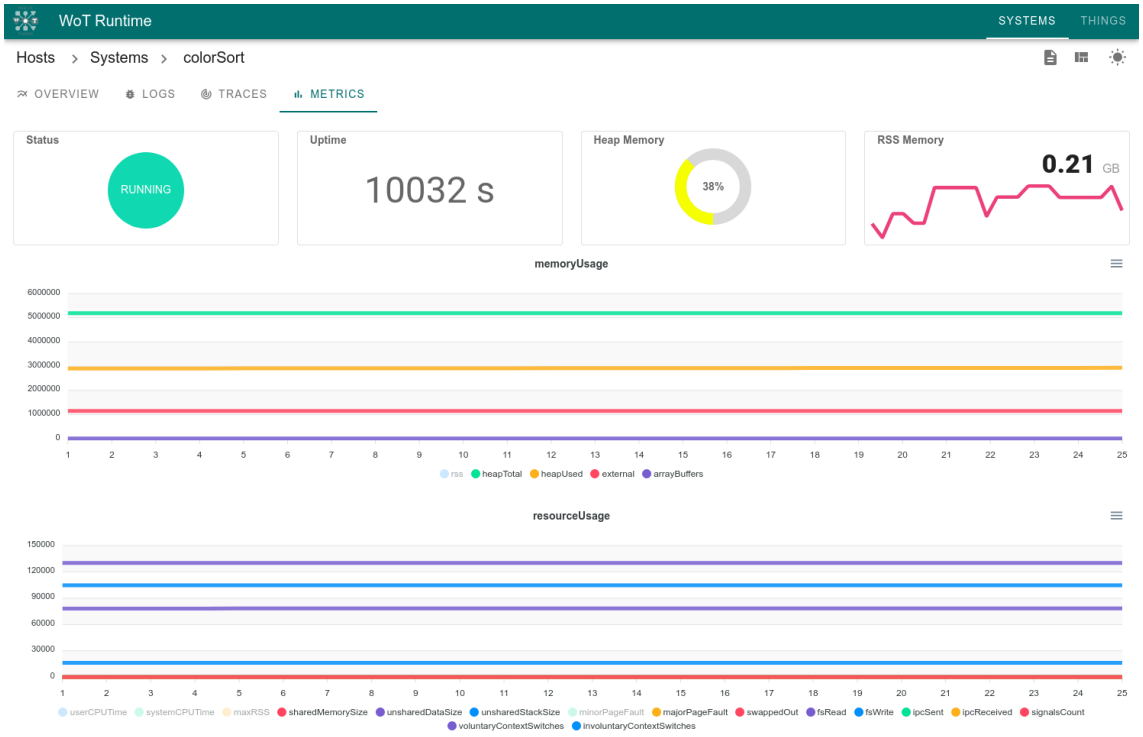

Figure 5: The Metrics tab of the WoT Runtime UI showing metrics information as custom, real-time visuals for a sample WoT System of name "colorSort". Starting at the top-left, these include the `status`, `uptime`, the percentage of heap memory used, as well as the resident set size (RSS) memory. In the central line charts, the values of `memoryUsage` and `resourceUsage` are displayed.

3.6.3 Traces

Traces are defined as a series of events that encode the end-to-end request flow through a software system. As such, traces are optimal for observing information on specific operations such as multi-service request flows. Due to this, traces usually are the most challenging to implement since every component in the path of a request must be modified to propagate tracing information. The trace data structure looks very similar to that of logs, and provides visibility into the path traversed by a request as well as its structure. They are therefore optimal for identifying function calls, remote procedure call (RPC) boundaries, and concurrency segments such as threads, continuations and queues in the path of a request. In addition to their expressiveness, they allow for observability across services, making them adequate for tracking WoT requests across the network.

While traditional traces are usually collected from HTTP requests, the WoT involves many more communication protocols, for which tracing had to be implemented. The proposed methodology enables a protocol agnostic approach to tracing, by tracking the protocol agnostic WoT Scripting API invocations rather than only the network request made by specific communication protocols such as HTTP. The approach is therefore tailored to the WoT and allows tracing both background interactions as well as those executed internally by the Mashup logic itself, enabling traceability of Mashup-specific information as well. Consider a running WoT System consisting of several TDs such as the one presented in Listing 2.1. In such a case, the presented approach allows would allow tracking Property write operations, Action invocations, as well as emission of Events. The Traces tab of the WoT Runtime UI shown in Figure 6 provides a real-time waterfall chart and a table view that facilitates search and drilling into traces for monitoring of WoT Systems.

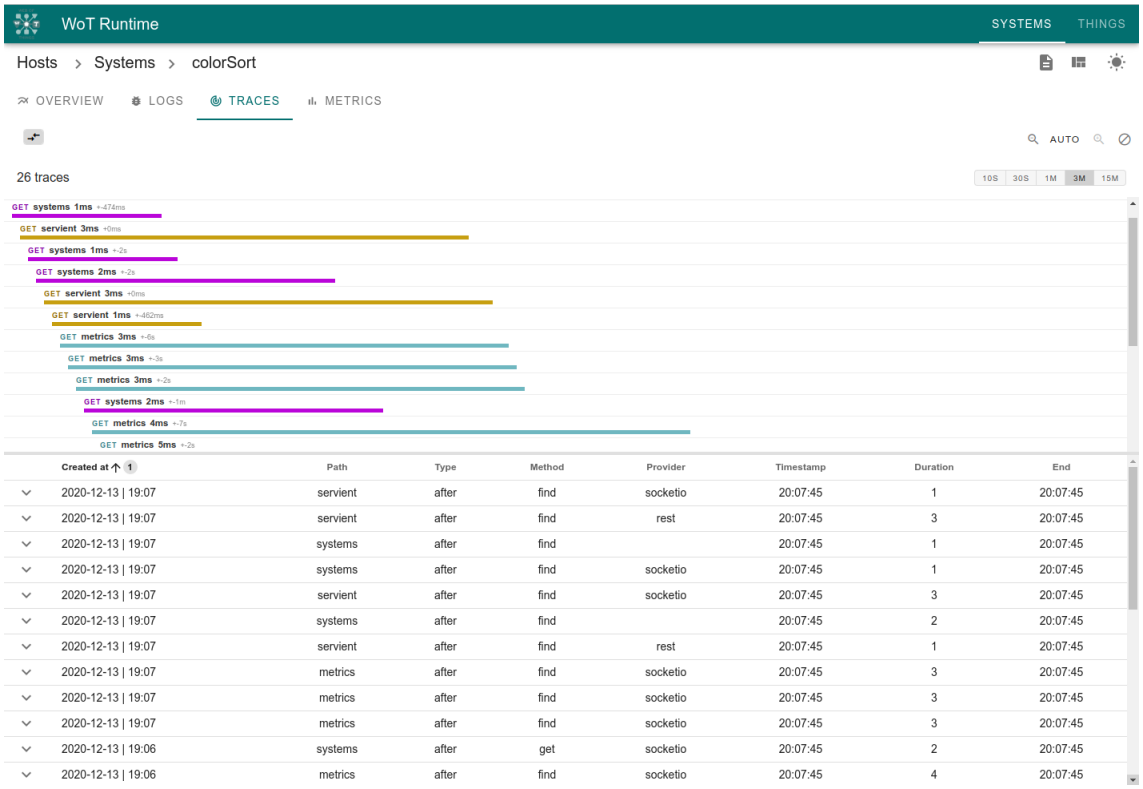

Figure 6: The Traces tab of the WoT Runtime UI displays real-time traces in a waterfall chart and a table view to provide detailed traceability to WoT System Runtimes.

4

Implementation

The implementation is delivered in form of a modular and extensible software framework, with the functional source code [1] available in the public domain, ready for usage and eventual extension. Details of the implementation are presented in the following sections.

4.1 WoT Runtime Framework

This section presents an architectural overview of the modular software framework proposed by this thesis: the WoT Runtime with its core and UI modules.

4.1.1 Core Module

The entire framework is structured around the standalone core module, which by itself should provide all intended functionality (remote deployment, management, and monitoring) and a portable distribution to allow for decentralized deployments both to the edge and cloud. Packaged as `wot-runtime/core`, it provides all the main features in a single package and enforces loose coupling through its RESTful admin API to promote development of third-party clients to directly interact with it, rather than locking users into specific clients. This allows integrating any number of optional clients, such as GUIs or command-line interfaces (CLIs). Development of the Core Module strictly aligns to the requirements presented in Chapter 3.

4.1.2 UI Module

In addition to the core package, a multi-tenant GUI is provided as a reference implementation of the entire set of features from the core package as a visual interface

1 https://gitlab.lrz.de/tum-ei-esi/wot-team/wot-runtime

to further streamline the development cycle of WoT Systems (see Figure 7). Packaged as `wot-runtime/ui`, it consists of a web application to allow targeting multiple WoT Runtime processes running across any number of hosts, both at the cloud or edge of the network, as long as they are accessible through it. The single requirement being that the IP address and port of the host is known and accessible for the GUI to connect to.

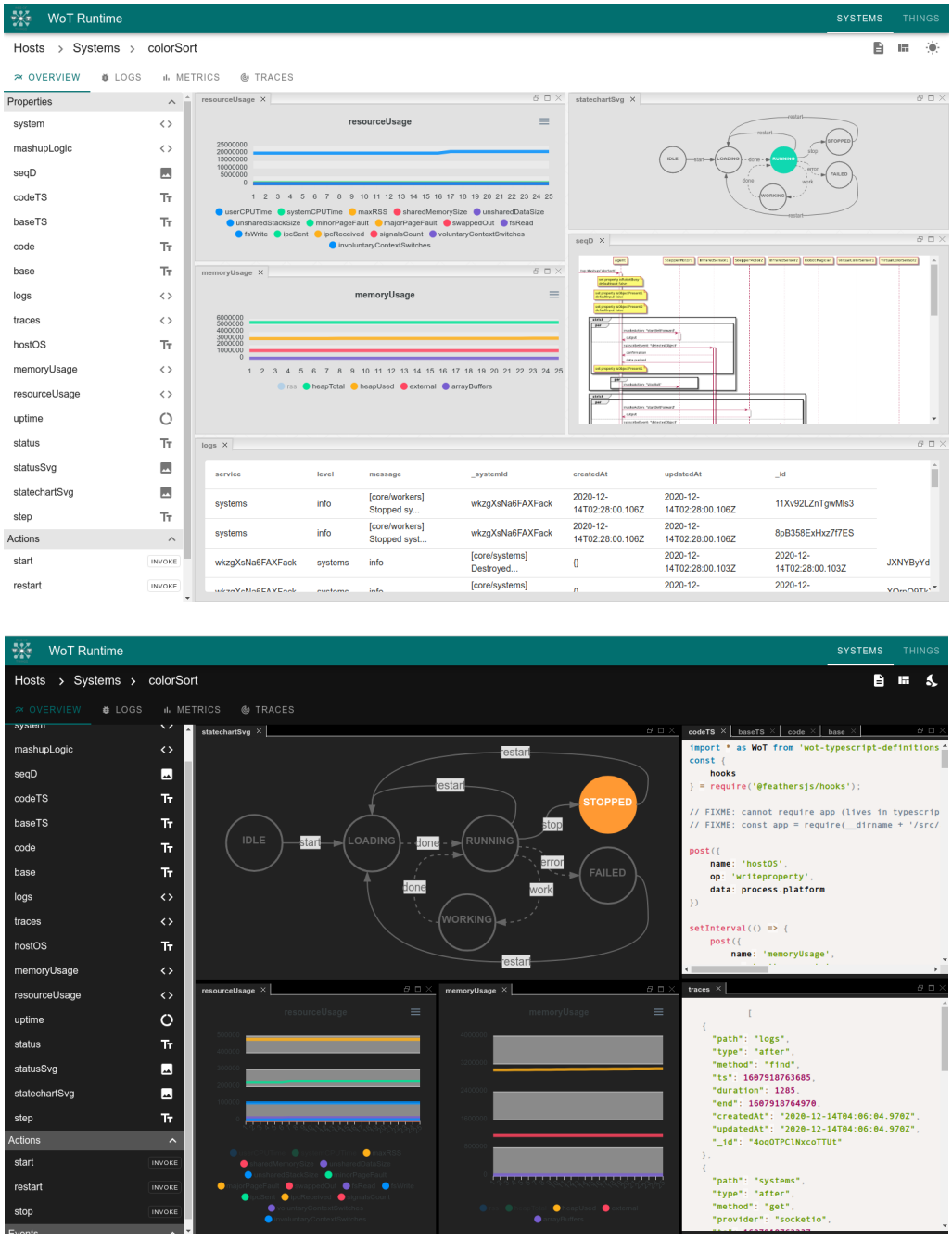

Figure 7: The WoT Runtime UI drag-and-drop dashboard seen in light mode (top) and dark mode (bottom). The present view allows dynamic composition of dashboards by dragging-and-dropping SRD Interaction Affordances from the drawer on the left into the dashboard, for each of which an adequate visual or control is displayed on drop. The panel layout is responsive, resizable, and stored between sessions for convenience.

4.2 Execution Environment

Refering back to goal G3, security and safety of the WoT Runtime is to be ensured to allow for its implementation in critical environments such as industrial use cases. The WoT Runtime parts from the assumption that code or SDs deployed by system owners are not malicious. However, runtime security and safety remain an important aspect to be considered. The approach by this thesis to ensure secure and safe code execution is further described in the following sections.

4.2.1 Sandboxing

The *WoT Runtime Framework* sandboxes each running WoT System Runtime according to Figure 8 in its own sandboxed execution context. Internally, the *WoT Runtime Framework* uses the `runScript()` method of the `thingweb.node-wot` library[2] of the Eclipse Thingweb project [7], the reference implementation of the WoT Scripting API [5]. This method uses the `vm2` library[3] internally, which is a sandboxing utility to allow execution of untrusted code, protecting against known methods of attack assuming the sandboxed code is not malicious itself. Still, the process is self-contained and does not have access to the external execution context, namely that of the core module. The sandbox created by `vm2` utilizes *JavaScript ES Proxies* for this functionality.

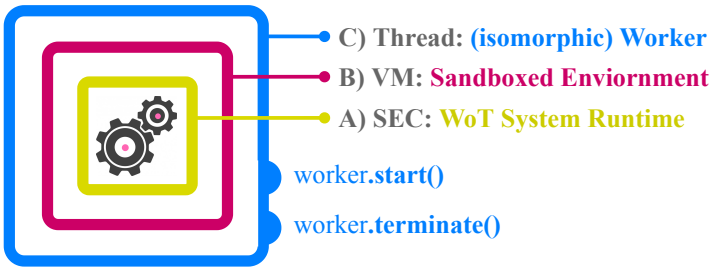

Figure 8: The sandboxing approach implemented by the *WoT Runtime Framework* to encapsulate self-contained script execution contexts (SEC) of each WoT System Runtimes (A) in a sandboxed environment (B) using virtual machines (VMs) and consequently in an optionally isomorphic worker (C), enabling execution control of the core process by directly controlling the outermost worker instance, which conveniently exposes a `start()` and `terminate()` methods.

4.2.2 Dependency Injection

Additionally, built-in `require` or `import` is overridden to control module access and allow requiring dependencies from inside the scripts to be run. Isolation is thus achieved by spawning an independent sandbox for each running WoT System Runtime instance, while importing third-party libraries and built-in modules is still possible from

2 https://github.com/eclipse/thingweb.node-wot
3 https://github.com/patriksimek/vm2

inside. Figure 8 illustrates the aforementioned sandboxing architecture with the sandbox environment (B) wrapping the WoT System Runtime (A).

4.3 Execution Control

Considering WoT Systems are to be deployed, managed, and monitored remotely, the need rises for a reliable execution control mechanism that ensures safety and security compliance, especially for use cases where WoT Runtime malfunctions could be a threat to human agents, such as in industrial contexts. For software, (finite) state machines provide a deterministic behavior model consisting of finite states and transitions which ensure that, based on the current state and a given input, the machine will produce the same output, or state change. The approach presented in this thesis for execution control is modelled as a state machine and implemented as such internally in the code, ensuring the WoT Runtime state can be controlled in a deterministic way.

4.3.1 Mapping to SRD

The deterministic execution control flow implemented internally and shown in Figure 2 is directly mapped to and made available by the SRD as Interaction Affordances for consumption (refer to the SRD template in Appendix Listing A.2). The current active state is exposed by the `status` Property which takes one of the finite list of states: `idle`, `loading`, `running`, `working`, `stopped` or `failed`. Analogously, all transitions between states are represented by Events which fire on state change. The manual state transitions (solid arrows in Figure 7) are exposed via the `start`, `restart`, and `stop` Actions for manual invocation, while automatic state transitions (dashed arrows in Figure 7) remain internal. The SRD is thus entirely available for consumption and interaction in the WoT context allowing a single-source of remote truth between the actual runtime and its digital representation on the consumers.

4.3.2 Control Mechanism

An additional mechanism for execution control of the sandbox is required since the VM implementation (B in Figure 8) used internally by the `runScript()` method of `thingweb.node-wot` library lacks the ability to stop or kill running processes. Because of this, each sandboxed WoT System Runtime is spawn inside its own worker thread (C in Figure 8), using the native `worker_threads` library of *Node.js*[4], which enables using threads to parallelize code execution. Apart from being useful to perform data-intensive operations, workers provide the WoT Runtime with the ability to gain execution control

4 https://nodejs.org

of the WoT System Runtimes by controlling the wrapping worker instance directly, which exposes a `terminate()` method to stop the worker instance (refer to Figure 8).

Apart from gaining an execution control mechanism, the worker adds an additional layer of security to the already sandboxed WoT System Runtime environment. In addition to the current implementation, usage of workers could eventually be adopted in browser clients via the web-native WebWorker API, instead of the `worker_threads` module. By doing so, the WoT Runtime could benefit from isomorphic deployments into either the core module (*Node.js*) or the UI module (browser). The optionally isomorphic worker (C) can be seen in Figure 3 wrapping the already sandboxed environment (B).

5

Evaluation

In this chapter two use cases are presented to exemplify the contribution and evaluate the approach. A smart farm simulation[1] and an industrial automation use case, illustrated in Figure 9 and consisting of a robot arm and two conveyor belts, each with an infrared sensor and color sensor, are chosen.

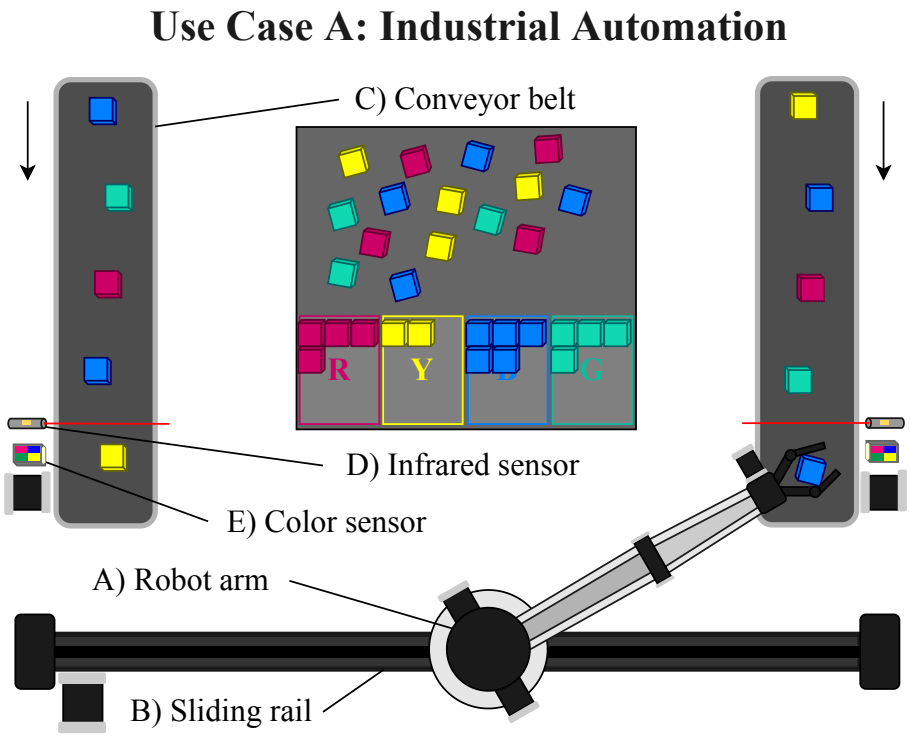

Figure 9: Top view of industrial automation scenario setup (use case A) with a robot arm (A) mounted on a sliding rail (B) and two conveyor belts (C), each with an infrared sensor (D) and a color sensor (E). The evaluation consists of a WoT System with the task of sorting the blocks by color and placing them in their corresponding drop zone in the middle platform.

1 https://github.com/relu91/WoTSimFarm

5.1 Metrics

The correct deployment of both WoT Systems in SD format, proper code generation and execution, as well as the ability to manage and monitor the running WoT System Runtime through its SRD are evaluated.

5.2 Procedure

The steps taken to evaluate the WoT System of each use case using the WoT Runtime UI are:

1. Create a new WoT Runtime host with default WoT Servient configuration.
2. Upload proper SD to host to describe the WoT System.
3. Start the System and check for correct Property readings, Action invocations, and Event notifications from the SRD in the UI and directly through an HTTP client.
4. Stop the System, modify its SD, re-deploy and repeat steps 2-3 for the System described by the new SD.

5.3 Setup

Evaluation is made on two independent hosts, each running a containerized WoT Runtime process with the default WoT Servient configuration. Table 5.1 gives an overview of the setup for each.

Criteria	**A) Industrial automation**	**B) Smart farm simulation**
Host OS	*Ubuntu 20.10*	*Raspberry Pi OS*
Protocol bindings	HTTP, CoAP, MQTT	HTTP, CoAP
No. of Things	5	9
Lines of TS code[a]	1393	252:
Lines of JS code[b]	1466:	288
Deployment format	SD	Code

[a]TypeScript code before transiplation.
[b]JavaScript code after transiplation.

Table 5.1: Evaluation setup for the industrial automation scenario (A, illustrated by Figure 9) and the smart farm simulation (B) (shown in Figure 10 use cases comparing them based on relevant criteria, listed in the first column.

5.4 Results

Both use cases are evaluated with the aforementioned procedure with positive results. The evaluation proves that the approach and implementation presented by this thesis works in practice. With reference to the requirements outlined in Chapter 3, the results of this thesis succeed in enabling remote deployment, management, and monitoring of WoT System Runtime instances from multiple arbitrary clients (R1), exposing them via the SRD to allow for consumption within the WoT context (R2). The SRD conveniently provides monitoring and verification utilities for both runtime and Mashup-specific information (R3). Moreover, the modular design of the WoT Runtime implementation allows composing loosely-coupled software systems (R4), exemplified by the also implemented WoT Runtime UI (R5), which serves as a reference visual client. A proof of concept is realized in an industrial automation and a smart farm simulation use case, enough to assert that an improved version of the proposed implementation could be used in production, or equivalently in domains with strict performance requirements.

The two distinct setups, shown in Table 5.1, indicate that the proposed methodology works for both high and lower-end devices like the *Raspberry Pi 4* used for the smart farm simulation use case.

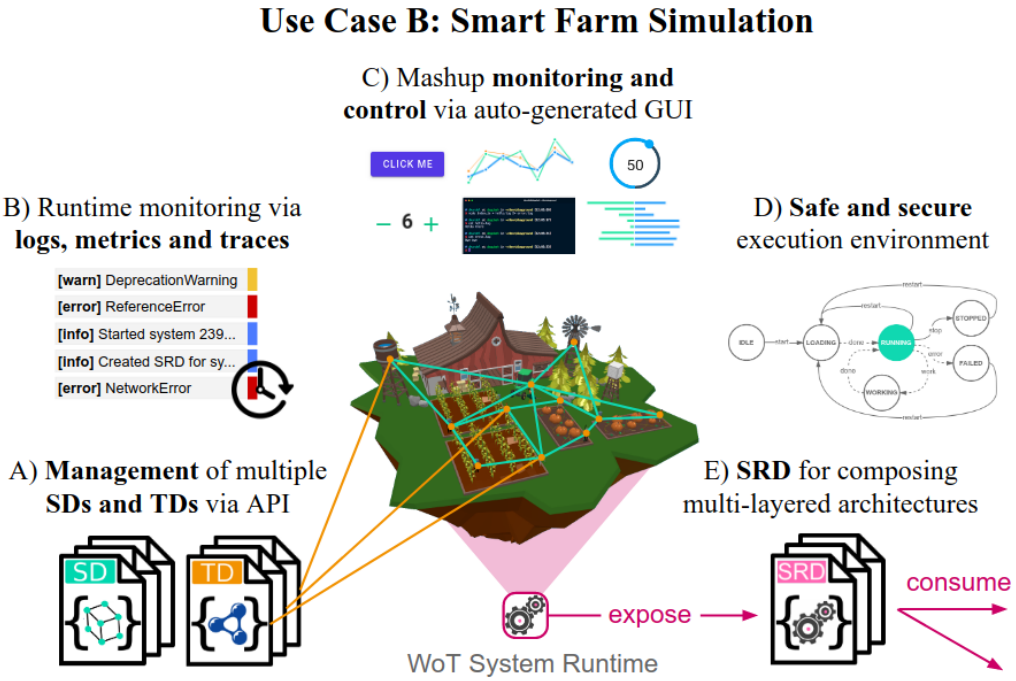

Figure 10: Overview of evaluation results in the example of the smart farm simulation (use case B). Through the WoT Runtime, management of multiple SDs and TDs (A), runtime monitoring via logs, metrics, and traces (B) and mashup monitoring and control via auto-generated GUI (C) is made possible. The WoT System Runtime runs in a safe and secure environment (D) exposed by the SRD for consumption and composition of multi-layered architectures (E).

6

Related Work

The WoT Runtime introduces a high-level framework for deployment, management and, monitoring of WoT Systems by leveraging the WoT Thing Description [2] and WoT System Description [3]. Despite similar methodologies existing for the deployment, management and, monitoring of software systems, only few target the W3C WoT. The approach remains unique in that it brings together all three features while focusing on the WoT exclusively. It parts from the assumption of pre-deployed Things and Systems to enhance the WoT development cycle, rather than providing a complete opinionated framework independent of the WoT. The following sections examine some of these methodologies and how they compare to the *WoT Runtime Framework*.

6.1 Deployment

Several solutions come to mind for the remote deployment of software and firmware in the IoT. For practical purposes, the distinct approaches rather than specific solutions are listed. On the cloud infrastructure side, Serverless and Containers are noteworthy. Serverless allows for the remote deployment and execution of functions in an as-a-Service manner, but remains best-suited for event-driven workflows. Containers on the other hand provide an encapsulated runtime for executing arbitrary code, and are thus similar to the approach proposed by the WoT Runtime. Nevertheless, container solutions (e.g. Docker[1]) do not provide a mechanism for deploying SDs directly like the WoT Runtime does, and are thus not streamlined for WoT applications. On the IoT side, multiple solutions for over-the-air (OTA) firmware deployment directly to devices exist. These however are usually suited for device-specific logic and security patches, rather than WoT Mashups to programm interactions between them such as the WoT Runtime. Specific to the W3C WoT, CLI tools like *WoT Application Manager (WAM)*[2] and the

1 https://www.docker.com
2 https://github.com/UniBO-PRISMLab/wam

`thingweb.node-wot` library CLI itself exist, which provide a lower-level command-line interface for executing arbitrary WoT scripts at the host. However, rather than deploying to a WoT specific runtime engine, they simply run by evaluating the code, apart from not supporting WoT System deployments in SD format and being solely command-line-based.

6.2 Management

No other solution for management of W3C WoT Systems in a single or across multiple remote hosts was found. WAM[7] provides scaffolding tools to speed-up and facilitate development of WoT applications in a lower-level CLI tool. This makes it adequate for development of WoT Mashup scripts, but not for taking those scripts into production such as the WoT Runtime. While many alternative platforms for management of containerized workloads and services exist (e.g. Kubernetes[3]), these are not specific to WoT nor do they come with support for direct deployment of WoT Systems from code or SDs. The WoT Runtime adds this functionality while providing management features common to these platforms.

6.3 Monitoring

No other monitoring solution was found with direct support for monitoring of W3C WoT applications. Generalizing, traditional software monitoring tools like Prometheus[4], OpenTelemetry[5] could be used to monitor WoT scripts based on logs, metrics and traces. However, these solutions require programmatically handling the monitoring functionality inside the WoT scripts, making them impractical for WoT Mashup development. Moreover, while great for observability at the network layer (e.g. incoming/outgoing network requests), they have no support for monitoring neither runtime nor Mashup-specific information such as the WoT Runtime. Visual monitoring solutions like Grafana[6] could be used to monitor WoT applications in flexible ways. However, this would require a high level of configuration from the user, as these tools are generally data source agnostic, but not WoT specific nor do they have support for TDs. The WoT Runtime on the other hand leverages TDs directly for zero-configuration, automatic GUI generation of both runtime and Mashup-specific information, making it practical for streamlined WoT development.

3 https://kubernetes.io
4 https://prometheus.io
5 https://opentelemetry.io
6 https://grafana.com

7

Conclusion

This thesis introduces an additional building block to the Web of Things for the remote deployment, management and, monitoring of WoT Systems. With it, WoT System development is streamlined and maintenance reduced, accelerating the W3C WoT innovation cycle. The presented approach is applied and evaluated in the publicly available implementation with two use cases: an industrial automation scenario and a smart farm simulation, demonstrating how the proposed solution works in practice. In these two use cases it is proven how the WoT Runtime enables scalable, reliable, and safe WoT Systems with minimal development effort, bringing the W3C WoT closer to users, reinforcing its adoption, and motivating the advancement into distributed, multi-layered WoT architectures.

7.1 Outlook

The disitributed Web presents an interesting case for continuation of the work presented in this thesis. Already, the WoT Scripting API provides an abstract framework to facilitate development of WoT applications, while the proposed *WoT Runtime Framework* provides a systematic approach to execute such applications in controlled runtime enviornments. However, these enviornments remain limited to the server-side runtimes, for which a similar methodology for deploying WoT applications into browser enviornments would be of great value. In light of this, a natural next step would be porting the WoT runtime core to run directly in the browser in addition to server-side environments.

By embracing the Web as a distributed computing platform, the benefits presented by the WoT would be exponentiated. By shifting control to the client, in most cases the browser, software applications become "unhosted"[1] and independent from remote servers. The sandboxing approach presented by this thesis and described in Section

1 Term coined first in https://gist.github.com/pfrazee/8949363

4.2.1 could easily be ported to run on the browser with an isomorphic abstraction over browswer-native `WebWorkers`. Departing from the traditional centralized model of the Web could pave the way for an open and distributed script-sharing model for the WoT. The methodology and open-source implementation presented in this thesis considers the eventual distribution of the Web in its design, serving as basis for future contributions in this direction.

A

Appendix

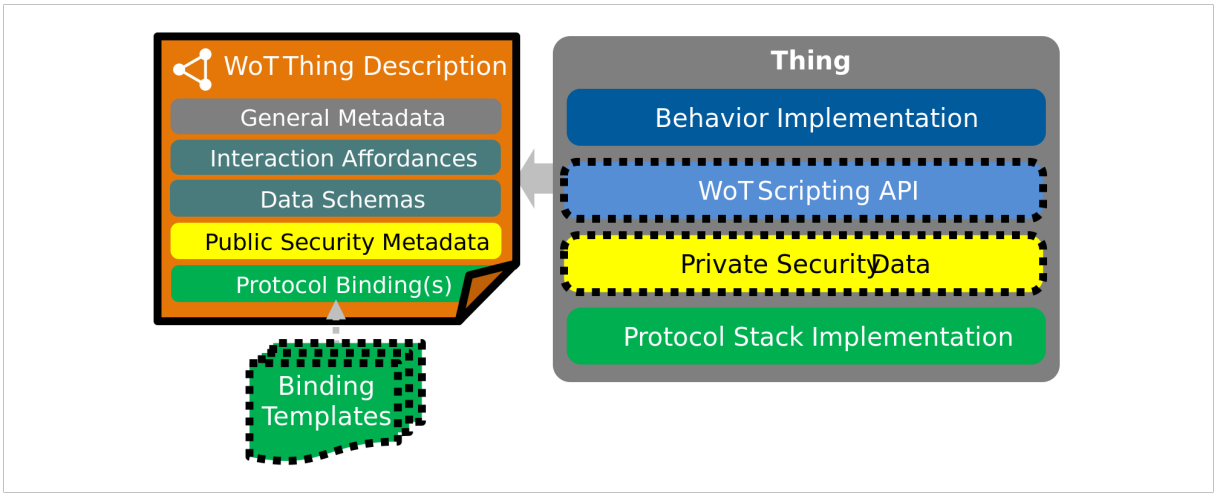

Figure A.1: WoT building blocks as specified by the WoT Architecture [1]

Keyword	Description
`loop`	loop construct for path array
`defaultInput`	input for setter functions, loop/count period, case comparison or default variable value
`$ref`	reference to a variable, property, function or action (borrows syntax from JSONSchema)
`wait`	command to pause execution of a path array
`interact`	an atomic mashup of a path array
`send`	array of sending interactions (part of an atomic mashup)
`receive`	array of receiving interactions (part of an atomic mashup)
`breakOnDataPushed`	defines when to execute sending interactions in an atomic mashup
`set`	setter function in a path array or as part of a receiving interaction
`get`	getter function in a path array or as part of another element which requires a variable as input, e.g., `send`, `loop`, `case`
`case`	parent keyword of a conditional execution. Contains `if`, optionally `else`, and `then`
`if`	contains condition of a `case` statement
`then`	contains the application logic to execute if condition of a `case` statement is true
`else`	contains the application logic to execute if condition of a `case` statement is false
`oneOf, allOf, anyOf`	used as children of `if` with array of further conditions, computed with the corresponding Boolean operator
`not`	can be used as a child element of `if` and has to contain a second child condition, which is inverted
`sync`	determines whether loop is to be executed synchronously or asynchronously
`type`	the type of loop, can be either `timed` or `logical`
`isUpdatedOnDemand`	used to determine when the value of a path is computed. The computation can happen periodically or on request
`defaultOutput`	used to determine the output of an action if it has no other return value

Table A.1: List of keywords introduced by the WoT System Description [3] which extend the WoT Thing Description [2] to allow representing Mashup logic.

```
{
  "@context": ["https://www.w3.org/2019/wot/td/v1", { "@language": "en" }],
  "@type": "",
  "description": "Conveyor belt with a stepper motor and RPi controller",
  "id": "urn:dev:ops:32473-ConveyorBelt-001",
  "security": "basic_sc",
  "securityDefinitions": { "basic_sc": { "in": "header", "scheme": "basic" }
    ↪ },
  "title": "ConveyorBelt",
  "base": "https://192.168.178.194:8080/",
  "forms": [
    {
      "contentType": "application/json",
      "href": "properties",
      "op": ["writeallproperties", "writemultipleproperties"]
    }
  ],
  "properties": {
    "speed": {
      "description": "Speed of the conveyor belt (negative values indicate
    ↪ backwards direction)",
      "forms": [
        {
          "contentType": "application/json",
          "href": "properties/speed",
          "op": ["readproperty", "writeproperty"]
        }
      ],
      "maximum": 100,
      "minimum": -100,
      "title": "Speed",
      "type": "integer",
      "unit": "%",
    }
  },
  "actions": {
    "toggle": {
      "description": "This action toggles (starts/stops) the conveyor belt
    ↪ with the current speed value",
      "forms": [
        {
          "contentType": "application/json",
          "href": "actions/toggle",
          "htv:methodName": "POST",
          "op": ["invokeaction"]
        }
      ],
      "output": { "const": "Conveyor belt toggled" },
      "safe": false,
      "synchronous": true,
      "title": "Toggle"
    },
  },
  "events": {
    "emergencystop": {
      "description": "Fires when conveyor belt emergency stop button is
    ↪ pressed.",
      "forms": [
        {
          "contentType": "application/json",
          "href": "events/emergencystop",
          "op": ["subscribeevent"],
          "subprotocol": "longpoll"
        }
      ],
      "title": "Emergency stop button pressed"
    }
  }
}
```

Listing A.1: Full example of a TD for a conveyor belt exposing its `status` and speed as Properties, the `toggle` command as an Action and the press of the emergency stop button as an Event.

```
{
  "@context": ["https://www.w3.org/2019/wot/td/v1"],
  "id": "urn:dev:org:wot-runtime:<systemRuntimeId>",
  "title": "<systemRuntimeName>",
  "description": "A TD for the runtime controller exposed by the wot-runtime",
  "securityDefinitions": {
    "nosec_sc": { "scheme": "nosec" },
    "basic_sc": { "scheme": "basic", "in": "header" }
  },
  "security": ["no_sec"],
  "properties": {
    "system": { "type": "object" },
    "mashupLogic": { "type": "object" },
    "seqD": { "type": "string" },
    "codeTS": {
      "type": "string",
      "contentMediaType": "text/x.typescript"
    },
    "baseTS": {
      "type": "string",
      "contentMediaType": "text/x.typescript"
    },
    "code": {
      "type": "string",
      "contentMediaType": "text/javascript"
    },
    "base": {
      "type": "string",
      "contentMediaType": "text/javascript"
    },
    "logs": { "type": "object" },
    "traces": { "type": "object" },
    "hostOS": {
      "type": "string",
      "enum": ["aix", "darwin", "freebsd", "linux", "openbsd", "sunos",
    ↪ "win32"]
    },
    "memoryUsage": { "type": "object" },
    "resourceUsage": { "type": "object" },
    "uptime": { "type": "number", "unit": "s", "minimum": 0.0 },
    "status": {
      "type": "string",
      "enum": ["idle", "loading", "running", "working", "stopped", "failed"]
    },
    "statusSvg": { "type": "string" },
    "statechartSvg": { "type": "string" },
    "step": { "type": "string" }
  },
  "actions": {
    "start": { "description": "Starts a WoT System Runtime process" },
    "restart": { "description": "Restarts a WoT System Runtime process" },
    "stop": { "description": "Stops a WoT System Runtime process" }
  },
  "events": {
    "start": { "description": "Status changed from 'idle' to 'loading'" },
    "done": { "description": "Status changed to 'running'" },
    "work": { "description": "Status changed from 'running' to 'working'" },
    "restart": { "description": "Status changed to 'loading'" },
    "stop": { "description": "Status changed from 'running' to 'stopped'" },
    "error": { "description": "Status changed from 'running' to 'failed'" }
  }
}
```

Listing A.2: Compact representation of the SRD template used to dynamically generate consumable WoT System Runtime controllers for each running instance in the WoT Runtime.

Bibliography

[1] M. Lagally, R. Matsukura, T. Kawaguchi, K. Toumura, and K. Kajimoto, "Web of things (wot) architecture 1.1," April 2020. Available from: https://www.w3.org/TR/2020/REC-wot-architecture-20200409/ [Accessed: November 4, 2020]. cited on p. 1, 5, 34

[2] S. Käbisch, T. Kamiya, M. McCool, V. Charpenay, and M. Kovatsch, "Web of things (wot) thing description," April 2020. Available from: https://www.w3.org/TR/2020/REC-wot-thing-description-20200409/ [Accessed: November 4, 2020]. cited on p. 1, 5, 6, 29, 34

[3] A. Kast, E. Korkan, S. Käbisch, and S. Steinhorst, "Web of things system description for representation of mashups," *International Conference on Omni-layer Intelligent Systems (COINS)*, pp. 1–8, 2020. cited on p. 1, 5, 7, 9, 11, 29, 34

[4] D. Guinard and V. Trifa, "Towards the web of things: Web mashups for embedded devices," *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences)*, vol. 15, p. 8, April 2009. cited on p. 5

[5] Z. Kis, D. Peintner, J. Hund, and K. Nimura, "Web of things (wot) scripting api," November 2020. Available from: https://www.w3.org/TR/2020/NOTE-wot-scripting-api-20201124/ [Accessed: November 5, 2020]. cited on p. 7, 21

[6] C. Sridharan, *Distributed Systems Observability*. O'Reilly Media, Inc., July 2018. cited on p. 15

[7] D. Peintner, M. Kovatsch, C. Glomb, J. Hund, S. Kaebisch, and V. Charpenay, "Eclipse thingweb project," 2018. Available from: https://projects.eclipse.org/projects/iot.thingweb [Accessed: November 26, 2020]. cited on p. 21