Programming IoT applications across paradigms based on WebAssembly

Karl Fessel[1] André Dietrich[1] Sebastian Zug[1]

Abstract: The key to IoT applications' success is the opportunity to exploit data generated by one node for various applications. Solutions for this are either centralized server systems, which aggregate the data and answer corresponding requests from different clients, or the concepts of edge computing, in which individual nodes take over the provision and processing of data directly. Although the advantages of immediate processing are obvious, edge computing concepts have so far been limited to more powerful nodes. Embedded in the DoRIoT project, we transfer the idea to low performance devices. This includes challenging questions related to security, scheduling and coordination issues. Additionally, we have to support the programming process itself. In order to achieve sufficient acceptance in the programming community we have to ensure that "freely programmable" is not bounded by hardware oriented programming paradigms and languages. Furthermore, the developer should be able to implement IoT-requests based on standard building blocks in a programming language of his choice.

In this paper we introduce the architecture and a tool-chain to cope with these challenges based on a WebAssembly-interpreter (WAMR) embedded in the DoRIoT software stack. The prototypical integration provides the applicability of WASM compiler tool-chain, originally focused on web-applications, and supports the orchestration of multiple requests in parallel.

Keywords: IoT; RIOT; DoRIoT; WASM

1 Motivation

The Internet of Things (IoT) [AIM10], Ubiquitous-Computing [We93], Industry 4.0 [HPO15], or Cyber-Physical Systems [Sh11], etc. is a collection of terms, which more or less share the same fundamental idea: in which an assembly of temporal and regional fluctuating heterogeneous systems share their information and capabilities to achieve a certain goal. Capabilities in this case means either sensing, acting, or computational resources.

Although the idea is pretty straight forward, it comes with a variety of yet unsolved problems, such as security and privacy issues, connectivity, the integration of hardware, diverging standards, performance, etc. Especially for low performance embedded nodes these open questions limit flexibility. Due to performance and security issues small sensor/actuator nodes show a closed structure that offers little scope for individual adjustments. In contrast to more powerful edge computing nodes, their behavior cannot be updated or adjusted

[1] Technische Universität Bergakademie Freiberg, Informatik, Germany, {Karl.Fessel, Andre.Dietrich, Sebastian.Zug}@informatik.tu-freiberg.de

doi:10.18420/inf2020_116

according to specific use-cases. Based on a fixed firmware, nodes transmit their measurements unfiltered with a predefined sample rate. The generic configuration intends to balance required communication bandwidth and update frequency to cover the requirements of all applications. But of course, individual messages generated according to tailored requests promise a higher utilization of the node and better system performance[Sh16].

The DoRIoT project[2] intends to overcome this separation between different node performance classes, related to their capability to execute user-specific requests. The project focuses on methods and tools for building self-organized systems, ranging from small sensor nodes (classes C0, C1, and C2 according to RFC7228 terminology) to server solutions. Users specify data aggregation and processing methods, the distributed intelligence assigns the requests to a specific node or a set of nodes according to communication bandwidth, accessible interfaces, timing constraints, etc. Consequently, each request has to be executable on different node architectures and operating systems. Virtual Machines (VM) or interpreters are commonly used to ensure hardware independence of applications. We evaluated their concepts and implementations, related to the chosen node classes, as well as multi-threading and multi-user capabilities, required performance capacities, supported languages, security issues, etc. In parallel, we investigate programming abstractions offered by the provided programming languages and paradigms.

A promising new approach in the field of interpreters are projects that try to transfer WebAssembly concepts to small IoT nodes. WASM is a binary instruction format for a stack-based virtual machine. It was designed as a portable compilation target for programming languages focused on client and server applications. WASM code runs natively in browsers; it is usually run by a combination of a interpreter and different optimizing levels of just-in-time (JIT) and ahead-of-time (AOT) compilation. WASM-bytecode can be generated by a huge number of compilers (from different source languages like C(++), Rust, Go, and many more) most of these compilers are build on top of LLVM-toolchain[3]. In the context of Web and C(++) code, the Emscripten SDK and tool-chain is often used to adapt preexisting C(++) code to the browser, by providing a libc-like API.

Based on the selection process described in Sec. 2 we integrated the WebAssembly Micro Runtime[4] interpreter into our project architecture, which is also supported by the Bytecode Alliance[5]. We identify three basic types of programs/tasks/usage-patterns (request, process, and function) that are typical for sensor networks and by providing generall WASM interfaces we will make these available to many languages.

[2] Dynamic runtime environment for organic (dis-)aggregating IoT-processes
DoRIoT project website `http://www.doriot.net/`

[3] originally “Low Level Virtual Machine”, project website `https://llvm.org/`

[4] `https://github.com/bytecodealliance/wasm-micro-runtime`

[5] `https://bytecodealliance.org/`

2 State of the Art

2.1 Programming Languages and Paradigms

According to the IoT Developer Survey held in 2019 by Eclipse Foundation (cf. [Ecl20]), the highest ranked IoT programming languages on constrained devices in 2019 were C, C++, Java, and surprisingly JavaScript. The available programming languages and paradigms are determined by the underlying Operating System (OS) running on the node. For low performance systems with tailored embedded OS (FreeRTOS, Contiki, RIOT OS[6][Ba18]) C is still the dominating language. As an alternative, TinyOS[7] offers a component-based, event-driven task model implemented in nesC[8], a specific C dialect[Ka07; OB09].

In contrast to previous examples, TinyDB offers a more declarative approach. It is a distributed query processing system for extracting information from a network of (smart) TinyOS sensors ([Ma05]). As the name suggests, it interprets a network of sensors similar to a database and, therefore, also applies a SQL-like syntax to collect data from a heterogeneous network of sensors. It borrows the semantics of `SELECT`, `FROM`, `WHERE`, and `GROUP BY` clauses from SQL, but it also offers further features, which have been especially developed to minimize the power consumption in sensor networks, such as life-time queries, dealing with events, or the creation of Semantic Routing Trees (SRT) for power-efficient information and query propagation.

SelectScript [DZK14] was developed while struggling with the dominant imperative programming paradigm in order ease the development effort for embedded systems and their access. It supports Python's data-types and operations, Lua's object-orientation based on prototypes and dictionaries, LISP's higher-order functions, lazy evaluation and tail-recursion. This was combined with a three valued logic (to simplify error handling) and SQL-like query capabilities, that can also be applied to solve reasoning problems. The key idea thereby was, not to be forced to switch between a program and an interface, so that one syntax or notion can be applied for programming but in the same way also be used for querying, no matter how complex or divergent a query might be. It exists an implementation of an VM[9] that had also been tested on 8-Bit microcontrollers.

Although there are approaches to use multi-paradigm languages, it seems to be more sufficient to support different languages on one device, based on the problem and the developer experience. There are other concepts such as miniKanren (see [By09]), for example. miniKanren is yet another relational programming language, but what makes it important in the context of IoT is, that it allows applying temporal logic [Ru18] and since it can run in "both" directions, it can be used either as a theorem-checker or -prover. For example, given the task to a sensor of measuring for ever or as long as possible under certain

[6] RIOT OS project website: `https://riot-os.org/`

[7] TinyOS project website: `http://www.tinyos.net`

[8] Network embedded systems C project website: `http://nescc.sourceforge.net`

[9] SandhillSkipperVM project website: `https://github.com/andre-dietrich/SandhillSkipper`

timing constraints, the sensor could figure out the optimal schedule based on local energy consumption, or at least give the answer why this goal cannot be fulfilled. This information can be either used to pass this task to another sensor, which meets the defined requirements or to re-plan the global task.

These examples show that tasks might need different tools and a flexible tool-chain for developing IoT-applications. This separates the selection of programming languages from the applied OS, and thus, liberates the user to chose the best solution for a given problem.

2.2 Virtual machines and Interpreters

Virtual Machines (VM) implement this request, but are mostly focused on one language. Java VMs support a large bandwidth of devices including embedded devices implementing a write-once-run-everywhere property based on a generic bytecode. Nevertheless, embedded Java code and run-times differ from desktop VMs. The general concept enables code mobility and thus to move code dynamically to different devices. This was utilized for example by JINI (cf. [Wa99]) and the OSGi framework (cf. [LNH03]), which are both so-called "service delivery platforms" that are used to tackle modularization, collaboration and service discovery in distributed systems. In contrast to fixed services traditionally realized in IoT-nodes, JINI and OSGi allow services to be dynamically installed, started, stopped, updated and even uninstalled. Additionally, services and clients can join or leave a federation anytime. Whereby JINI can also load functionality into a process (locally) even while the process is running.

Next to these not so commonly known examples there are of course also ports of other (mostly imperative) languages to realm of embedded systems, such as Python MicroPython[10], PyMite [PNS09], or Zerynth[11]. Unfortunately Zerynth is not mentioned in the Eclipse IoT survey, but next to an embedded Python VM it also offers an OS[12] abstraction and combines with ChibiOS or FreeRTOS, which allows to running Python and C programs in parallel.

There are a number of Projects that support running JavaScript on embedded devices like Espruino[13] (supporting microcontrollers at C2 level) and Tessel[14],which requires significant more resources than C2 in RFC7228-Scale[15]. Alternative solutions use NodeJS on systems starting from RaspberryPi level. They run JavaScript code either by just in time compilation or interpretation, which means the code has to be transported over the network and be either jit-compiled or interpreted on the target, this code can be preprocessed for compactness and/or execution speed making JavaScript a pseudo assembly and VM.

[10] MicroPython project website: `https://micropython.org/`
[11] Zerynt project website: `https://www.zerynth.com/`
[12] ZeryntOS project website: `https://www.zerynth.com/zos`
[13] Espruino project website: `http://www.espruino.com`
[14] Tessel project website: `https://tessel.io/`
Tessel 1 project website: `http://web.archive.org/web/20150213073259/https://tessel.io/`
[15] Tessel 1: ARM-M3 (180MHz, 32MB RAM); Tessel 2: MIPS (580MHz, 64MB RAM)

2.3 WASM implementations for embedded systems

If a VM is too restricive and taylored for one programming language only, the application programmer is not able to choose the most suitable solution for the job anymore. Another approach is to select a general VM to which multiple languages compile, such as the VM-Model for WASM. It exists a huge number of open-source WASM interpreters[16] some of them are applicable to and/or target small embedded systems. Small systems need the VM to run as an interpreter for bytecode without any JIT compilation. We tested a Rust based approach (wasmi[17]), which at the moment does not integrate well with RIOT and its tool chain, in contrast to C that is well supported by RIOT that targets embedded systems and has a good and active developer community. We applied the WebAssembly Micro Runtime[18](WAMR), which is based on a modular approach that makes it very adaptable, furthermore it already supports multiple embedded operating systems. Other interpreters that may fit such systems are WAC[19], which has been ported to ESP32, or WASM3[20] that relies on tail-call optimization, which might be problematic with some compilers. The WASM3 documentation states that is able to run on system starting at ~64Kb for code and ~10Kb RAM. WAMR claims to use 85Kb for the Interpreter and use a low amount of memory, it also provides a greater set of Post-MVP Features than WASM3.

3 Application concept

3.1 Building blocks of the aggregation

The range of potential user codes reaches from aperiodic single shoot accesses on current measurements over periodic data aggregations in combination with smoothing algorithms up to complex event driven aggregation functions, providing domain-specific data formats. For realizing and monitoring the execution of dynamically assigned user byte-codes we have to structure them in predefined building blocks, implemented on a set of abstract interfaces. Hence, we need to identify common patterns in user code and provide related interface implementations in the WASM tool-chain. System functions like `wait()`, `readSensorData()`, or `display()` close the gap between the OS and WASM interpreter.

Based on the requirement analysis of the DoRIoT-Project we identified the three query types representing abstract building blocks of actual user codes, they are described in a pseudo-code semantic:

Request ... part of a program that run once and in most cases answers one question or triggers one action. All resources (RAM, program memory) are free again after a single run.

[16] `https://github.com/appcypher/awesome-wasm-runtimes`
[17] `https://github.com/paritytech/wasmi`
[18] `https://github.com/bytecodealliance/wasm-micro-runtime`
[19] `https://github.com/kanaka/wac`
[20] `https://github.com/wasm3/wasm3`

```
request(){ send me the temperature at sensor 3 };
```

Process ... part of a program that will stay running and process data that is needed for the app to provide its service or part of this.

```
process(){ wait(10 seconds);
query data and save to database };
```

Function ... part of a program that can provides a specific functionality that can be called from nodes within the network. A function extends the interface of the node for all queries, according to permissions.

```
fuction_avgtemp(){return average of saved temperature-data};
```

These conceptional elements are independent from each other and may be combined in an app as they provide orthogonal functionality. Access to these building blocks and to the SystemAPI will be managed by access control and capability management.

3.2 Implementing multi-paradigm aggregation requests

How can basic components support real world applications, coping with a variable number of different aggregation and processing chains? Let's consider a system of IoT-nodes equipped with different sensors (temperature, humidity). Different users transmit queries for data aggregation and processing.

The first example implements an isolated process, continuously calling the system functions `wait()`, `getTemperature()` and `display()`. These functions are implemented in separate headers and linked during compile time. We assume that individual IoT-nodes provide a specific collection of these functions. Continuous updates of a temperature-display that directly mounted to the node is realized by an imperative programming paradigm.

```
process(){
  wait(10 seconds);
  var x = getTemperature();
  display(x);
};
```

Lst. 1: Process configuration for a single sensor data aggregation and output

If we extend the scenario with an additional node that controls a ventilation system, the aggregation process may consider external sensory data too. Hence, an abstract `query()` replaces the system function call from the previous example. The `query()` criteria is evaluated at run-time and references local system functions or/and remote aggregation methods. If external sensor data is relevant, the interpreter spreads out corresponding `request()` queries to surrounding nodes. This way, the second example implements a spatial criteria (`room`) and amplitude (`>22C`) for filtering in Lst. 2, line 3. In order to realize such potentially complex filters, the query concept has to integrate declarative programming concepts.

```
1   process(){
2       wait(1 minute);
3       if query(num of all room temperatures where temp > 22C)> 0:
4           switchhigh();
5       else:
6           switchlow();
7   };
```

Lst. 2: Ventilation system: instead of requesting it to switch its airflow, a process is installed that automates this task

By adding humidity sensors to the system, its performance can be further improved. The calculation of absolute humidity requires relative humidity information and air temperature, which should be located in-between.

```
process(){ //virtual sensor, calculating absolute humidity
    wait(10 seconds);
    var sat_humidity = calcSatHumidityatNormalPressure( query(temperature where room is same as
         this ) );
    var rel_humidity = getHumidity();
    store abs_humidty = rel_humidity/sat_humidity;
};

function_absHumidity(){ //precalculated value
    coap_return(abs_humidty);
}

process(){ //ventilation controller
    wait(1 minute);
    if query(num of all temperatures where temp > 22C) > 0:
        switchhigh();
    else if query(num of all temperatures where temp < 16C) > 0:
        switchlow();
    else if coap_request(outdoor, absHumidity) < coap_request(indoor, absHumidity):
        switchhigh();
    else
        switchlow();
};
```

Lst. 3: Extending the temperature controlled ventilation system by a humidity sensor

The sensors act either as a function or as a process node, to keeping this information up to date. A single shot calculation would be a request, thus it would require to transport the code every time. This information and the air quality tracked at the outside and the inside over time, to discover trends in combination with a weather service, may be a good measure to decide when to open and close ventilation. A system like this may not need a full climate control and thus save energy. The required calculation may reside on a node that has access to all of this data and may publish control information for the ventilation system. Such a compilation needs memory and access to the weather service and therefore may reside on an edge node. It spawns multiple processes and functions which send requests to other nodes.

The three pseudo code examples illustrate the vision of the project, an intuitive combination of context-sensitive building blocks with complex logical statements. The user code accesses actual hardware functions through system-API-functions for timing, memory access, periphery access, and communication. This functionality gets combined and distributed via self-defined functions.

4 Integration and Implementation

DoRIoT integrates the multi-user approach on three layers into the general system architecture on top of RIOT OS. Actual user queries are realized by so-called DoRIoT-Apps, combining the mentioned building blocks. `WAMR` implementation of a WASM interpreter connects the app level and the OS, supervised by the Runtime Access Control layer. An integrated supervisor creates and manages modules as well as VM instances and generates monitoring information. The general access to OS interfaces is controlled by an internal a Run-time Access Control Unit, control information will be provided by LCap-Lightweight Capability Based Access Control [BG17].

The transfer of WASM code is managed by CoAP. The CoAP message may contain further attributes that enable the evaluation of LCap's access rules and performance restrictions of an app.

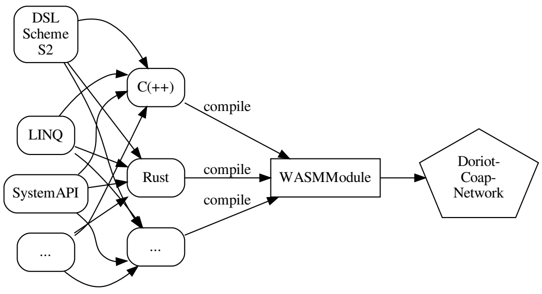

Fig. 1: DoRIoT compile chain

Based on WASM integration and corresponding tools the system already support the inclusion of multiple programming languages. The availability of WASM as a target within the LLVM-tool-chain further help the adaption of its massive number of llvm-front-ends that implement the translation of source-code and therefor programming languages within the LLVM toolchain. Other languages or combinations of multiple may be compiled in a multi-step process. Fig. 1 illustrates this compile-chain.

We emphasized the need of a multi-paradigm concept while developing code for IoT-systems and intend to realize the combination of imperative and declarative program parts following the LINQ (Language Integrated Query) concept. While LINQ[21] itself is a specific group of implementations within .Net, its approach was ported to many other languages. The example shows how the pseudo-code `query` might be adapted to a LINQ programming style, that is already parseable for (pre)compilers with numerous adaptations of this concept.

```
request(){
    TermperaturSensors.update();
    coap.return( Query( from(s, TermperatureSensors).where( s.temp > 22).orderby(s.temp)));
}
```

Lst. 4: Query and filter temperature sensors based on Lst. 2 (similar but extended)

5 Outlook & Summary

We believe that there is a strong desire in porting different programming paradigms to IoT applications, even to the smallest devices, since it liberates the development process. Some standard tasks can and shall be realized in C while others can concentrate onto functional or logical programming, or execute snippets in order to enable complex queries, and thus shift some of the "global" task's execution logic down to the end devices.

The basis, of course, is a working API that allows to access an all-of-systems-service for all tasks (written in different languages). The application of a VM furthermore enables some form of service control, that is not common in this particular case. On the one hand, it is possible to restrict the memory consumption, which is vital. On the second hand, the usage of an API and the VM's possibility to enforce restrictions, it is also possible to define more fine granular access and execution control for different tasks.

Acknowledgements

This work is funded by the German Federal Ministry of Education and Research (BMBF) as part of the project “DoRIoT” under grant number 01IS18071A.

References

[AIM10] Atzoria, L.; Ierab, A.; Morabitoc, G.: The Internet of Things: A survey. Computer Networks 54/15, pp. 2787–2805, 2010.

[Ba18] Baccelli, E.; Gündogan, C.; Hahm, O.; Kietzmann, P.; Lenders, M.; Petersen, H.; Schleiser, K.; Schmidt, T. C.; Wählisch, M.: RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. IEEE Internet of Things Journal, Vol. 5, No. 6, pp. 4428-4440/, Dec. 2018.

[21] .Net LINQ Manual `https://docs.microsoft.com/en-us/dotnet/standard/using-linq`

[BG17] Buschsieweke, M.; Güneş, M.: Securing critical infrastructure in smart cities: Providing scalable access control for constrained devices. In: 2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC). 2017.

[By09] Byrd, W. E.: Relational programming in miniKanren: Techniques, Applications, and Implementations, PhD thesis, Indiana University, 2009.

[DZK14] Dietrich, A.; Zug, S.; Kaiser, J.: SelectScript: A query language for discrete simulations. ACM Transactions on Programming Languages and Systems (TOPLAS)/, 2014.

[Ecl20] URL: `https://iot.eclipse.org/community/resources/iot-surveys/assets/iot-developer-survey-2019.pdf`, visited on: 06/08/2020.

[HPO15] Hermann, M.; Pentek, T.; Otto, B.: Design Principles for Industrie 4.0 Scenarios: A Literature Review, tech. rep., Technical University Dortmund, Fakulty of Mechanical Engineering, 2015.

[Ka07] Kabadayi, S.; Julien, C.; O'Brien, W.; Stovall, D.: Virtual sensors: a demonstration. In: The 26th international conference on computer communications: demonstrations track (Infocom). Pp. 10–12, 2007.

[LNH03] Lee, C.; Nordstedt, D.; Helal, S.: Enabling Smart Spaces with OSGi. Pervasive Computing, IEEE 2/3, pp. 89–94, 2003.

[Ma05] Madden, S. R.; Franklin, M. J.; Hellerstein, J. M.; Hong, W.: TinyDB: An acquisitional query processing system for sensor networks. ACM Transactions on Database Systems (TODS) 30/1, pp. 122–173, 2005.

[OB09] OBrien, W. J.; Julien, C.; Kabadayi, S.; Luo, X.; Hammer, o.: An architecture for decision support in ad hoc sensor networks. Electronic Journal of Information Technology in Construction 14/, pp. 309–327, 2009.

[PNS09] Pedersen, R. U.; Nørbjerg, J.; Scholz, M. P.: Embedded programming education with lego mindstorms nxt using java (lejos), eclipse (xpairtise), and python (pymite). In: Proceedings of the 2009 Workshop on Embedded Systems Education. Pp. 50–55, 2009.

[Ru18] Rudavsky-Brody, N.: Temporal Logic, μKanren, and a Time-Traveling RDF Database. ACM on Programming Languages/, 2018.

[Sh11] Shi, J.; Wan, J.; Yan, H.; Suo, H.: A Survey of Cyber-Physical Systems. In: International Conference on Wireless Communications and Signal Processing. IEEE, pp. 1–6, 2011.

[Sh16] Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L.: Edge computing: Vision and challenges. IEEE internet of things journal 3/5, pp. 637–646, 2016.

[Wa99] Waldo, J.: The JINI Architecture for Network-Centric Computing. Communications of the ACM 42/7, pp. 76–82, 1999.

[We93] Weiser, M.: Ubiquitous computing. Computer 26/10, pp. 71–72, 1993.