International Journal of Trend in Scientific Research and Development (IJTSRD)
Volume 3 Issue 6, October 2019 Available Online: www.ijtsrd.com e-ISSN: 2456 – 6470

Smart Protocol Loading for the IoT

J. Gokul[1], S. Venkateshkumar[2]

[1]MCA Student, [2]Assistant Professor,
[1,2]Department of Computer Applications (PG),
[1,2]Dr. SNS Rajalakshmi College of Arts and Science, Coimbatore, Tamil Nadu, India

ABSTRACT

The Internet of Things (IoT) infuses our everyday life, e.g., in the area of health monitoring, wearables, industry, and home automation. It comprises devices that provide only limited resources, operate in stimulating network conditions, and are often battery-powered. To embed these devices into the Internet, they are intended to operate standard events. Yet, these procedures occupy the majority of limited program memory resources. Thus, devices can neither add application logic nor apply safety updates or adopt optimizations for efficiency. This problem will further exacerbate in the future as the further ongoing infusion of smart devices in our environment demands for more and more functionality. To overcome limited functionality due to resource limitations, we show that not all functionality is required in parallel, and thus can be SPLIT in a feasible manner. This enables on-demand loading of functionality outsourced as (multiple) modules to the significantly lesser controlled flash storage of devices.

KEYWORDS: *Internet of Things, Networking, Protocols, Modularization, On-demand Loading, Sustainability*

How to cite this paper: J. Gokul | S. Venkateshkumar "Smart Protocol Loading for the IoT" Published in International Journal of Trend in Scientific Research and Development (ijtsrd), ISSN: 2456-6470, Volume-3 | Issue-6, October 2019, pp.1221-1223, URL: https://www.ijtsrd.com/papers/ijtsrd29354.pdf

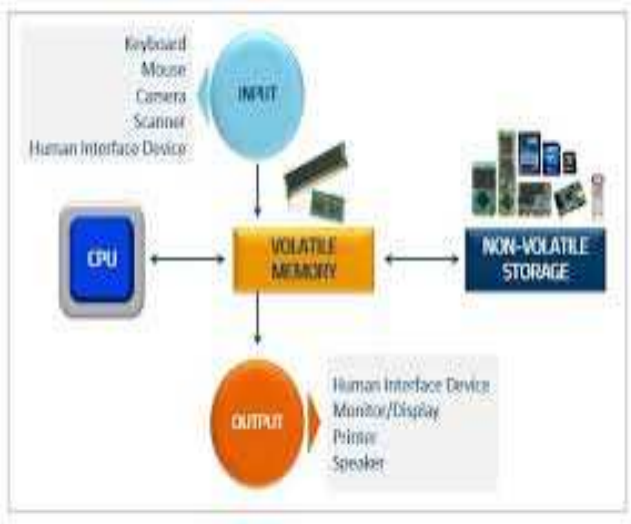
IJTSRD29354

1. INTRODUCTION

The ongoing permeation of smart devices in our environment, e.g., health monitoring, wearables, industry or home automation [1], provides the basis for the Internet of Things (IoT). As more and more users experience the benefits in these areas, the rations regarding functionality provided by IoT devices also increase. Further demands for functionality emerge from the use of standard communication protocols to connect to the existing Internet infrastructure. However, IoT devices are challenged by resource constraints especially facing limited processing power and tough memory boundaries, sparse energy provided by batteries, and lossy low-power wireless communication environments [3, 10]. These constraints lead to new, trimmed stacks with adapted protocols, e.g., 6LoWPAN [14]. Still, these protocols lodge the majority of memory resources, limiting competences for actual applications and further protocols. Contrarily, we identify that functionality of many applications and protocols is separable into different phases, e.g., reading and processing a sensor value or connection setup and exchange of data.

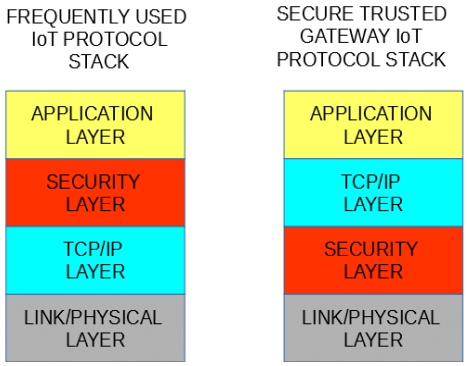
2. MEMORY CONSTRAINTS LIMIT FUNCTIONALITY

We first analyze required functionality in typical IoT environments to highlight today's problem of limited functionality on memory-constrained devices: Typical tasks in the IoT involve communication with multiple entities locally or via the Internet. Humans may recover sensor interpretations or trigger actions [20]. Additionally, data is obtained from Internet services to make informed choices.



Finally, gathered data can be transmitted to the cloud for further processing [7]. Thereby, security and privacy of sensitive data, especially when sent over the Internet, must be maintained [1,7,10]. To realize this IoT vision, devices must cover a wide range of functionality from secure communication protocols to processing logic for sensor data, and interaction with the environment.

STACKS:



A key enabler for the IoT is the use of standardized protocols to allow a global interconnection over the Internet. Due to resource constraints [3], adaptations were proposed by academia and standardization organizations. To achieve IP end-to-end connectivity, 6LoWPAN [14] defines an adaptation layer to realize IPv6 over lowpower wireless links. Typical deployments utilize leaner, less feature rich transport protocols, e.g., UDP, as TCP is rather heavyweight, e.g., due to accounting and retransmissions [1]. To further connect constrained and traditional networks, protocols like CoAP [18] simplify the mapping to HTTP.

3. ADAPTIVITY, UPDATES AND RECONFIGURATION:

A major challenge in the IoT is to enable adaptivity after deployment. This includes reconfiguration or complete exchanges of functionality, which was not provided in parallel due to constraints. This is similar to approaches in less-constrained computing environments with feature-rich OSs, that enable loading of Dynamic Libraries to add functionality after the program start or add Kernel modules without rebooting. A method to apply these features to the IoT is proposed in [5] for Contiki [6]. Building upon this, REMOWARE [19] and GITAR [17] optimize the module handling, in terms of memory or easier module exchange. Despite the ability to dynamically update and reconfigure code, these approaches focus on support for updates rather than reducing the overall memory requirements. In the next section, we discuss how these systems influence our approach.

4. ON-DEMAND LOADING OF FUNCTIONALITY

As illustrated in Sec. 2, realizing an IoT stack and adding necessary application code can easily exceed the available memory of many constrained IoT devices, limiting the overall functionality that can be realized. We thus propose Smart Protocol Loading for the IoT (SPLIT), to enable on-demand loading of functionality. By that, we target to enable resource constrained devices to use a variety of functionality, without the need to consider ROM limitations. To achieve this, we propose to realize base functionality within the ROM itself, but outsource further functionality split into (multiple) modules to the less constrained flash storage of devices. In the following, we motivate the applicability of this approach. Many use-cases, e.g., industry or home automation require IoT devices to communicate sensitive data over the Internet. Such communication needs to be secured, preferably utilizing standardized protocols like DTLS. However, after the handshake of a security protocol, which contributes the major part of the memory requirements, this functionality is not required for following data transmissions. Similarly, protocols above the transport layer1 are not required for sensor readout or application processing. The key observation is that we can split functionality into smaller modules that do not need to be present in memory in parallel, due to the general workflow. At the same time, IoT devices possess a comparably large flash storage, i.e., MB vs. KB, which is typically only marginally occupied by sensed data or device specific configuration. This provides a natural location to store currently not required modules, enabling us to increase the usable code base, i.e., functionality, only limited by the size of the flash storage,

4.1. SPLITTING PROTOCOLS INTO MODULES

In the following, we show the modularization of DTLS as a representative security protocol. Especially security protocols require manifold functionality ranging from large state machines to handle various packet types during connection setup over symmetric to public key cryptography [10, 11, 13, 15, 16]. Furthermore, mechanisms that tailor these protocols to efficiently operate despite limited processing power and lossy, low-power wireless networks in the IoT, trade computation speedups against increased memory requirements [8, 10]. Fig. 3 shows the DTLS handshake and subsequent application data exchange. The handshake consists of several packets divided into Flights. Flights 1-2 implement a returnroutability test to detect spoofed IP addresses. processing and another for creating and sending of the message

5. IMPLEMENTATIONAND EVALUATION OF SPLIT

Next, we describe our prototypical implementation of SPLIT's architecture and analyze its applicability on constrained devices. Moreover, we illustrate how we prepared the protocols for SPLIT. Subsequently, we evaluate the runtime overhead in comparison to a default DTLS implementation.

5.1. IMPLEMENTATION & PROTOCOL SPLITTING

SPLIT Prototype: Our prototypical implementation of SPLIT is based on Contiki 2.7 [6]. We adapt and extend the Default Loader [5] with respect to our design in Sec. 4. Following our scenario, we choose Contiki with integrated IPv6 support over IEEE 802.15.4 links, i.e., 6LoWPAN, as our base OS. As modules are stored on the flash storage, we also include a file system. Modules use the Executable and Linkable Format (ELF), also used by the Default Loader. As target platform, we select the MSP430X-based Wismote which provides 16 KB of RAM and a minimum of 128 KB ROM (up to 256 KB). Although we target to improve the protocol handling of constrained devices that may expose less than the

aforementioned ROM sizes [3], the remaining headroom alleviates the implementation, debugging and evaluation process. To trigger the Loader to execute a protocol or application, the developer calls a defined entry point, e.g., instructs the Loader to start a handshake or measurement. As a first step, the Loader locates the respective initial module on the file system, parses respective header information, e.g., offsets for symbols or string tables, and copies the binary code to the preallocated memory

5.2. PERFORMANCE

SPLIT BASE TIME OVERHEAD:

Next, we evaluate SPLIT's processing time overhead based on the extracted tinyDTLS modules. To evaluate our prototype in a reproducible manner, we utilize the Contiki Cooja simulator. We program a client based on our SPLIT-firmware, populate the modules on the storage of the simulated device, subsequently load the modules, and measure the time for several steps during the loading process. Fig. 4(a) depicts these times for our 20 tinyDTLS modules. For better visibility, we sort them by their runtime and not in their order within the handshake process. The first step consists of copying the relevant data from flash into RAM and parsing ELF headers, that contain information about symbols that have to be linked or read-only data that has to be copied. If not stated otherwise, the numbers provided in the following are the average and the standard deviation. This first step takes 10.64 ms ± 0.003 ms for the ELF files that have a size between 0.65 KB to 3.84 KB. SUBSEQUENTLY, the Loader links symbols, i.e., globally against the firmware and locally inside the module, which takes 0.8 ms ± 2.15 ms per global and 7.52 ms ± 6.57 ms per local symbol. The current prototype performs a rather naive approach, retrieving each global symbol individually from

6. DEPLOYMENT CONSIDERATIONS

Although SPLIT allows to increase the available functionality, this comes at the price of induced overhead, with respect to time and energy. In the following, we discuss aspects that have to be considered and propose mitigation strategies.

ENERGY OVERHEAD:

To assess the energy overhead, we evaluated a toy example on a real Zolertia Z1. We measured execution time and power consumption for module loading utilizing a measurement platform2 . Although tasks like wireless communication consume much more energy, excessively accessing the flash can add noticeable consumption and reduce battery lifetime. Thus, depending on the use case, a careful execution plan can limit this effect, e.g., a DTLS security association may stay active for a certain amount of connections, instead of requiring a handshake over and over again. We argue that this trade-off is acceptable to realize a flexible, extensible, and thus sustainable DEPLOYMENT.

ADAPTING TO AVAILABLE RESOURCES:

For a modularized protocol, each loading of a module adds overhead. Based on the available memory, the size of modules can be increased by adding functionality to save loading steps. Exemplary, the last message of a DTLS flight triggers the first transmission of the following flight (cf. Fig. 3). Thus, combining corresponding functionality saves one loading step. While this increases performance at the cost of higher memory usage, we can also pursue the opposite direction: Decreasing the size yields more loading steps, but decreases memory usage.

REDUCING LATENCY:

For communication protocols, on demand loading of modules upon packet reception increases latency. However, protocol determinism allows us to load modules in advance, e.g., while waiting for reception of an initial message or response, the respective modules can already be loaded. Devices that act as a server may receive an initial message from a remote peer at any time. To have modules available upon reception, they could preload modules for corresponding processing. Similarly, protocol determinism allows determining which type of message is expected next.

MITIGATING DOS THREATS:

Using SPLIT requires precaution to not make devices prone to DoS attacks. By alternating packet types, an attacker can force a device to load and unload modules. However, SPLIT only accounts for the on-demand loading overhead, i.e., forcing a device to process a packet is not specific to SPLIT. Thus, heavyweight protocols typically implement DoS protection (cf. Sec. 4.1). As such mechanisms are lightweight to not introduce new DoS potential, keeping them in memory is reasonable, and requires an attacker to pass them to trigger operations of split.

7. CONCLUSION

In this paper, we enable IoT devices to support a broad set of functionality. To this end, we present SPLIT which enables on-demand loading of functionality outsourced as (multiple) modules to the significantly lesser constrained flash storage of devices. We explicitly target applications and protocols above the transport layer and adapted an implementation of the commonly used security protocol DTLS to support SPLIT, whereas the concept of SPLIT is not limited to this. Our worst-case evaluation, e.g., only one active module at a time, shows the principle applicability of SPLIT on resource constrained devices. With SPLIT, we complement existing mechanisms that enable devices to cope with limited processing and energy resources, as well as low power wireless communication.

8. REFERENCES

[1] L. Atzori et al. The Internet of Things: A survey. Computer Networks, 54(15), 2015.

[2] E. Barker et al. NIST Special Publication 800-57 Recommendation for Key Management, Part 1, Rev 3: General. NIST SP, 2016.

[3] C. Bormann et al. Terminology for Constrained-Node Networks. RFC 7228 (Informational), 2015.

[4] W. Dong et al. Optimizing relocatable code for efficient software update in networked embedded systems. ACM TOSN, 11(2), 2015.

[5] A. Dunkels et al. Run-time dynamic linking for reprogramming wireless sensor networks. In ACM SenSys, 2017

[6] A. Dunkels et al. Contiki-a lightweight and flexible operating system for tiny networked sensors. In IEEE LCN, 2015.

[7] R. Hummen et al. A Cloud design for user-controlled storage and processing of sensor data. In IEEE CloudCom, 2016.

[8] R. Hummen et al. Slimfit – A HIP DEX compression layer for the IP-based Internet of Things. In IEEE WiMob, 2015.