Appears in: Proceedings of the IEEE Consumer Communications & Networking Conference (CCNC '25)
Las Vegas, USA, 2025

PFIP: A UDP/IP Transactional Network Stack for Power-Failure Resilience in Embedded Systems

Kai Vogelgesang[1], Ishwar Mudraje[1], Luis Gerhorst[2], Phillip Raffeck[2], Peter Wägemann[2], Thorsten Herfet[1], Wolfgang Schröder-Preikschat[2]

[1] Saarland Informatics Campus (SIC), Germany, {vogelgesang,mudraje,herfet}@cs.uni-saarland.de
[2] Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany, {gerhorst,raffeck,waegemann,wosch}@cs.fau.de

Abstract—**Emerging embedded devices in the Battery-Free Internet of Things have the benefit that they harvest their required energy during runtime from the environment (e.g., through solar power). However, from the perspective of the systems' networking stacks, the main challenge is resilience against power failures: Existing network stacks for such systems (e.g., LwIP) face the problem that stored data, such as for address translation, is likely to be lost or inconsistent after a power outage. Besides the consistency of data, sending a packet without the knowledge about the required and available energy can result in energy inefficiency when the power failure occurs during sending, because of the energy waste of the incomplete packet.**

In this paper, we introduce PFIP, a network stack for UDP/IP specifically targeting scenarios with intermittent power supply. PFIP's primary design consideration is to modularize the network stack into distinct transactions in order to result in a state-machine–compliant structure with states and according transitions. The stack is able to introduce checkpoints between transactions to persistently store the stack's state. Besides handling data consistency, we employ code-analysis techniques that determine the energy demand of states/transitions. Combining the energy demand of operations along with the available energy on our hardware platform eventually yields runtime guarantees such that started transactions will safely be completed without facing power failures.

Index Terms—**power-failure resilience, embedded systems, UDP/IP, intermittent operation**

I. Introduction

Battery-Free Internet of Things: With the Battery-Free Internet of Things [1] appearing on the horizon, several novel use-case scenarios of embedded (consumer) electronics are becoming increasingly feasible: Battery-free systems acquire their operational energy from the environment, for example, by means of solar cells, RF-harvesting techniques, or exploiting temperature gradients. These systems come with the primary benefit of being energy self-sufficient and not requiring periodic battery replacement. Examples of such energy-constrained systems include smart-home devices, wearable devices, medical implants, or devices for predictive maintenance. From a design and implementation perspective, these connected systems come with several new challenges due to their unstable and unpredictable power supply.

Intermittent Operation & Checkpointing: The power supply causes the system to only execute in between power outages. Since our work targets network stacks that maintain a (potentially increasing) state over time, our approach has to persistently store this state within a checkpoint. One example of relevant state within a network stack is its cache for storing resolved mappings from IP addresses to MAC addresses, which is referred to as the Address Resolution Protocol (ARP) cache. Losing this state would require the embedded node to initiate new requests for address resolution, which is an undesired property, especially in highly energy-constrained settings. Even worse, when losing parts of the data in case a power failure hits during checkpointing, the stored state might be inconsistent.

In Search of a Suitable Network Stack: Inherent to these nodes in the Battery-Free Internet of Things is their communication with other nodes, and thus a network stack. Existing approaches in the context of intermittently-powered systems target communication with Bluetooth Low Energy [2] or LoRa [3], [4] due to their focus on low-power communication. However, numerous devices, for example in smart homes, have the possibility to use an existing Wi-Fi network. As a consequence thereof, the focus of our work is on wireless IEEE 802.11 communication.

PFIP*'s Contributions:* To tackle the problem of intermittency in wireless communication, we introduce the PFIP network stack for power-failure–resilient IP networking. The current state of PFIP builds upon our previous work [5] and addresses UDP/IP networking. The contributions of this work include an abstraction of the networking stack's possible states and transitions. Our implementation of PFIP relies on this state-machine–like representation, specifically on Petri-nets [6]. In short, the contributions of this paper are threefold:

1) Introduction of Petri-net–based semantics into an 802.11 network stack with PFIP
2) Static resource-bound analysis possible due to PFIP's model-based semantics
3) Evaluation of our open-source prototype implementation of PFIP

Paper's Structure: The remainder of this paper is structured as follows: First, we introduce necessary background information and our system model in Section II. Subsequently, we highlight the main problems for intermittently-powered network stacks in Section III. Section IV gives insight into our PFIP approach. We show evaluation results of our approach in Section V. Works related to PFIP are the subject of Section VI. Section VII outlines future work and concludes our paper.

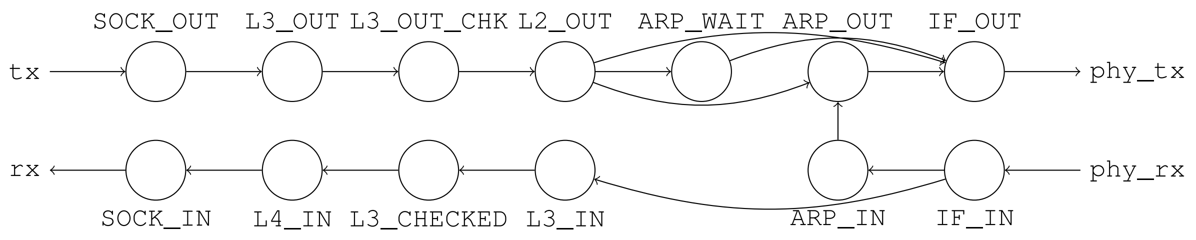

Fig. 1: Places used for packet processing. The `transmit()` function (marked `tx` here) is called from the application similar to a `send()` call for POSIX sockets. It creates a token with the payload in the `SOCK_OUT` place, which is then subsequently passed down the stack. In each place, operations such as appending a header or calculating a checksum for the respective layer are performed. Once the token arrives in the `IF_OUT` place, it contains a fully formed packet, ready to be transmitted over the physical interface. Similarly, any packets received there are placed into the `IF_IN` place and travel up the stack, with headers being parsed and checksums verified at intermediate steps, until the application can receive the payload out of the `SOCK_IN` place by calling `try_receive()`.

II. Background & System Model

Layers in the Stack: The 802.11 standard specifies both the medium access control (MAC) and physical layer (PHY) protocols. For now, PfIP's focus is on the higher layers, that is, Layer 2 and above of the IP suite. Consequently, we target platforms that support the IP suite. We require physical-layer–agnostic transmit and receive primitives. Further, these primitives have an upper bound on their energy demand in order to encapsulate these operations into transactions.

Notion of Intermittency: We target energy-constrained embedded systems that communicate with each other via Wi-Fi. The focus of this work is on the intermittently-powered embedded nodes; we assume routers have a stable power supply. Since the embedded nodes are unable to rely on stable power, PfIP inherently cannot give temporal guarantees when to send packets from the perspective of nodes. However, we require the systems to execute useful work with the available energy. Specifically, we require the energy storage to hold minimum budgets for operations: For example, for address resolution, nodes require energy to send the request and receive the answer within a given amount of time.

Security-Related Aspects: Security-related aspects go beyond the current scope of PfIP. In our implementation, inbound traffic blocks the processing of outbound traffic. Hence, malicious filling of the inbound traffic chain could yield to denial-of-service for outbound traffic and also drain the energy storage. Security aspects have to be considered in future work.

Static Analyzability: PfIP relies on the fact that operations have a bounded amount of energy in order to guarantee the safe completion of transactions. Likewise, interrupts are bounded by a minimum inter-arrival time. To yield such worst-case energy-consumption estimates, we require the system to be analyzable by means of static program-path analysis techniques [7], [8]. Our system model includes all power-consuming operations being explicitly controlled by the software (e.g., sending out a packet).

Notion of Synchrony: In our current UDP/IP-centered stack, we require a notion of synchrony for the individual transactions. That is, waiting for an indefinite amount of time for an acknowledgment, as part of bidirectional communication, goes beyond this paper's scope. For waiting requests, our model requires an upper bound on the time to wait in order to limit the overall time, and likewise energy, of transactions.

III. Problem Statement

We address two main problems with PfIP, namely, maintaining consistency in the network stack (see Section III-A) and avoiding retransmit-induced energy waste (see Section III-B).

A. Problem # 1: Consistency of Network Stack

Network stacks consist of several operations (e.g., computing checksums) and a respective state (e.g., the ARP cache for address resolution). Existing network stacks [9]–[11], being unaware of power failures, face the problem that these operations are interrupted, and the state is lost. Even worse, when writing data to non-volatile memory and not having a notion of transactional semantics, a power failure might render the memory in an inconsistent state.

PfIP's Approach to Consistency of Network Stack: In a nutshell, we decompose the network stack into distinct states and their associated transactions. If necessary, the system's state can be stored in non-volatile memory before each transaction in order to guarantee consistency and avoid data loss.

B. Problem # 2: Retransmit-Induced Energy Waste

While the transactional semantics allow for data consistency in the stack, energy-agnostic transactions can still be interrupted by power failures. For example, the system might start to transmit a packet via the radio, and just after starting the transmission, a power failure could interrupt the execution. As a consequence thereof, the energy expended for transmitting the incomplete packet is lost. Numerous approaches in the domain of intermittent systems have the following notion of intermittency: Instructions can be directly executed from the non-volatile memory. That is, the system's semantics allow for incremental progress of the application. In contrast to the direct execution of machine-code instructions from non-volatile memory, when handling communication with radios, the incremental, instruction-wise progress no longer holds: Interrupting a network packet renders the entire packet incomplete. Especially in highly energy-constrained systems, as we target, such energy waste is undesired.

PfIP's Approach to Retransmit-Induced Energy Waste: In short, we estimate the worst-case energy demand of individual transactions along with their transitions. Combined with the online available energy, we can guarantee that transactions are completed without risking a power failure in between.

IV. The PfIP Approach

To communicate with the network, a system is often equipped with multiple physical interfaces, e.g., a Wi-Fi radio or an Ethernet port. For the application running on the system, a high-level abstraction is desirable, and typically provided in the form of sockets. The task of the network stack is to take care of what happens in between by implementing the required suite of protocols.

A. Transactional Semantics

As stated in Section III, our first main goal is to ensure consistency of the network stack in case of a power failure. We approach this goal by modeling the task of the network stack using transactional semantics: The task is split into a set of transactions, in a way that 1) the stack is in a consistent state before and after a transaction is applied, 2) transactions are independent of each other, and 3) the order in which they are applied does not matter. This mimics the ACID principles from database design and guarantees by design that the stack is always in a consistent state. Furthermore, as seen below, the representation of this state is clear and well-suited for creating a checkpoint in case of an impending power failure. In contrast to periodic checkpointing, our selective approach reduces overheads.

We implement a stack for simple UDP/IP communication based on datagrams or packets. This means that we need to provide a primitive to transmit and receive a datagram to the application. For the physical layer, we assume a "driver" primitive, capable of transmitting and receiving layer 2 frames. Implementation details on this are given below in Section IV-B1.

The task at hand is to formulate the processing steps between these primitives in terms of transactions. We decided to use a state machine, where each packet is represented by a token that is moved between states by a respective transaction. We draw inspiration from colored Petri nets, which are well suited to represent this kind of state machine (hence the use of "places" to mean the states in which a token can be).

However, there are a few notable differences: 1) Each transition only takes one token as input. In particular, there is no interaction between tokens. 2) Within a transition, we allow for computation and even side effects (e.g., calculating the checksum for the IP header or inserting an entry into the ARP cache). 3) There may be side effects (e.g., a new entry in the ARP cache), which affect whether a transition can fire. Using "pure" Petri-net semantics to implement the stack is possible, for example, by representing the ARP cache as a

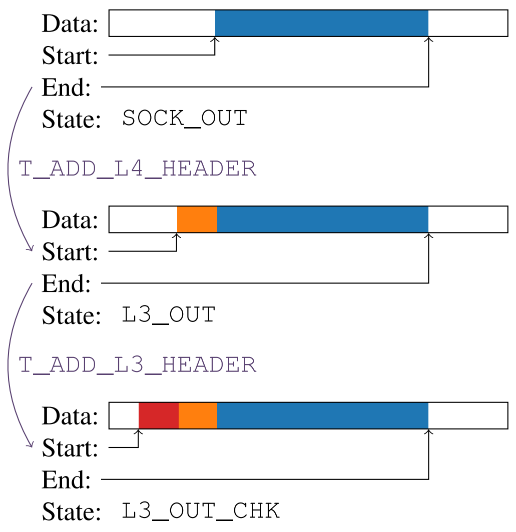

Fig. 2: An outgoing token going through the first transitions in the stack. Initially, it only holds the payload (blue) and resides in `SOCK_OUT`. From there, the `T_ADD_L4_HEADER` transition prepends the UDP header (orange), decreasing the start offset accordingly, and places the token in `L3_OUT`. Then, the IP header (red) is added by `T_ADD_L3_HEADER`, again decreasing the start offset and moving the token to `L3_OUT_CHK`, from where the next transition proceeds.

token as well. We decided against this approach for the sake of simplicity, and to have a correspondence between tokens and packets. Fig. 1 shows the places and transitions in our implementation.

Like in colored Petri nets, each transition has a corresponding guard function. This function also takes one token as input and returns a boolean whether this token is eligible for processing by the respective transition.

Consider the following example of how a payload is processed by the stack: After being called from userspace, the transmit primitive creates a token, holding the given payload, in the initial state for outgoing packets. Only one transition leads out of this state, and it prepends the UDP (L4) header to the payload and moves the token to a state for L3 processing. From there, the next transition prepends the IP header, and so on, until the token arrives in a state for outgoing L2 frames. It is then removed, and its content – a now fully formed L2 frame – is passed to the low-level transmit primitive. The first steps of this process are shown in Fig. 2.

B. Implementation

Our target platform is the RISC-V–based ESP32-C3 [12] from Espressif. The ESP-IDF environment uses FreeRTOS and offers development support for a wide variety of applications. Under the ESP-IDF, the default TCP/IP stack is LwIP [9], but it can be exchanged for a custom implementation. ESP-IDF provides a network interface library (`esp_netif`), which interfaces the low-level Wi-Fi driver to the TCP/IP stack.

The Wi-Fi drivers provide high-level APIs for various tasks, including but not limited to:

- Configuration of the MAC and PHY layer hardware
- Support for station mode operation, including WPA-based authentication and association with an access point
- Registration of callbacks for packet reception

All operations can be monitored through FreeRTOS events triggered by the Wi-Fi driver. However, the low-level 802.11 MAC and PHY layer operations are only accessible through binary blobs that are tightly integrated with the FreeRTOS operating system. We use an SPI-based FRAM chip to save the state of the network stack in the event of a power loss.

1) Primitives: The hardware is initialized and configured by an RTOS task. Wi-Fi tasks are carried out in a separate thread. We identified the undocumented public functions for transmission and reception in order to circumvent the `esp_netif` library. The internal transmit function, `esp_wifi_internal_tx` allows transmission of raw L2 packets in the Ethernet frame format. The 802.11 MAC and PHY headers are crafted by the Wi-Fi driver, which also queues the packet for transmission over the wireless hardware.

For packet reception, a callback is registered with the driver using the internal `esp_wifi_internal_reg_rxcb` function. The Wi-Fi thread concurrently invokes our handler whenever an L2 packet is received. Towards the userspace, we provide an interface similar to calling `write` and `read` on a socket: The `transmit` function takes a buffer and its length as arguments and starts the transmission of the buffer content as one UDP packet. To receive a packet, we provide a non-blocking `try_receive` function, also taking a buffer and its length. If a packet is ready for reception and fits inside the buffer, it is moved into the buffer and can be used by the application. Otherwise, the function immediately returns.

Note that our implementation only supports a single, fixed peer. The architecture can be expanded to multiple peers, for instance, by attaching additional metadata to the tokens. However, our current prototype is sufficient to demonstrate the possibility of worst-case resource-consumption analysis.

2) Semantics: We implement tokens as buffers, together with attached metadata (current place, and content start and end offsets) as shown in Fig. 2. The size of these buffers is 1500 bytes, since this value is most likely to fit the path's maximum transmission unit. To avoid unnecessary copies, we leave a "headroom" before the payload when we create a new token, similar to the `sk_buff`[1] structure used in Linux. Headers can then simply be prepended by writing them into the available headroom and decreasing the start offset.

Transitions are implemented as functions operating on these buffers, modifying their content and metadata. These functions receive a token via a pointer and return `void`, token output is realized by updating the place of the given token. In case more than one token needs to be produced, they are allocated. Tokens are allocated from a fixed-size pool. This removes the overhead of dynamic allocations, lowering the variance of worst-case execution time and energy cost.

Each transition has a corresponding guard function, which also receives a token via pointer and returns a boolean. For these functions, we keep a low-cost, in most cases just one, memory lookup since they are often evaluated in the code.

To implement the "business logic" of the stack, we aim for one transition per ISO/OSI layer, but we extract, e.g., the IP checksum calculation into its own transition. Where to "draw the lines" between transitions is a design decision: Having smaller but more transitions gives better granularity and lower worst-case cost per transition. We refer to Section V for our approach to determining this worst-case cost. Choosing the transitions too small leads to overhead, and moves the control flow into the state machine. This makes it difficult to reason about the code, and whether progress is ensured in case of power failures.

Up until layer 2, tokens go through a "pipeline" of transitions. The transmit chain adds headers, while the receive chain removes them. If an unexpected value is encountered in a received packet, for instance, if the IP checksum is invalid or the UDP destination port is wrong, the token is discarded.

Other than this, the only token-level branching happens for ARP: In case the corresponding MAC address is unknown for a destination IP, the packet is put into the `ARP_WAIT` state and a request is scheduled in form of a token in `ARP_OUT`. Otherwise, the layer 2 header is appended, and the packet is put into `IF_OUT` and subsequently transmitted.

C. Resource-Consumption Bounds

In order to give guarantees on the completion of individual transactions, we require an upper bound on the energy demand of single states and transactions. Namely, we require estimates of their *worst-case energy consumption*, or WCEC for short. These resource-consumption estimates can be determined with static worst-case analysis tools, such as the *Platin* toolkit [13]. Such tools rely on two central pillars: (1) a hardware-agnostic program-path analysis and (2) a hardware-specific cost model. First, the program-path analysis derives path constraints from the control flow of the program under analysis. To facilitate this analysis part, we implemented PFIP with a focus on analyzability. Specifically, PFIP uses loops with a statically determinable loop count and avoids recursion. Secondly, the cost model gives the analysis tool a notion of the actual cost of individual operations, for example, the time and energy demand for sending a packet over the physical layer (i.e., layer-2 packets). The results of both the hardware-agnostic and the hardware-aware pillar are combined in a mathematical problem formulation (i.e., an integer linear program). Solving this formulation eventually yields a worst-case estimate for the analyzed part. These offline-determined estimates are then used during the runtime phase: Assessing the currently available energy and comparing it with the worst-case estimate answers whether the completion of an operation can be guaranteed.

[1]https://docs.kernel.org/networking/skbuff.html

D. *Checkpointing*

We implement a simple checkpointing scheme to save the network state. Owing to the transactional semantics of PFIP, it suffices to only store currently used tokens from the token pool. For each token, its place, start and end point of data and the data itself is saved to FRAM. The ARP cache is also stored to ensure progress of tokens in the `L2_ARP_WAITING` state. In addition, a value indicating the number of stored buffers as well as a flag to indicate a power loss are saved.[2] Upon waking up, the ARP cache and the tokens are restored.

V. Evaluation

We design our measurement-based evaluation to validate that each of our networking stack's transactions has a predictable execution time and energy consumption. The longest-running and most energy-intensive transaction dictates the size of the energy storage that must be provisioned for the system. Therefore, our evaluation focuses on identifying the upper bound of our transaction's energy consumption.

A. *Evaluation Setup*

To evaluate PFIP, we use two devices connected through a 2.4 GHz 802.11bgn Wi-Fi Network with WPA2 PSK (CCMP) security. One ESP32-C3 runs PFIP and sends out UDP packets to a second device that relays the payload back. The second device and the router therefore only impact the networking latency and not our measurement results. The payload has random content and a random size of 0 B to 1024 B, and we repeat each measurement 100 times. In our test setup and configuration, the Wi-Fi chip wakes up at the DTIM interval of 204.8 ms. This background activity potentially influences the energy consumption of our network-stack transitions.

To measure PFIP's energy consumption, we use the Joule-Scope JS220 energy measurement device. To synchronize the code execution with the measurement device, we use a GPIO-based trigger. We validate our execution-time measurements with the chip's internal µs-accurate hardware timer.

For guaranteed upper bounds on the energy consumption, we analyze each of PFIP's transitions with the *Platin* [13] toolkit. As *Platin*'s approach involves collecting high-level information about the program control flow by the compiler, source-code access is detrimental. As the low-level driver code for the Wi-Fi chip of the ESP32-C3 is not available, we had to exclude the transmission `T_TX_PACKET` from our analysis, as it relies on using the closed-source function `esp_wifi_internal_tx`.

[2]In our prototype, we measure that this unoptimized approach to checkpointing takes 27.3 ms for saving the state with an 8 kB Adafruit SPI Non-Volatile FRAM Breakout Module (ADA1897) on our evaluation platform. Even without switching to a higher-performance non-volatile memory, we estimate that this could be reduced to 4-5 ms since only some buffers are useful and each buffer must not be stored entirely. We choose not to pursue this, as this work focuses on the predictability of the networking stack.

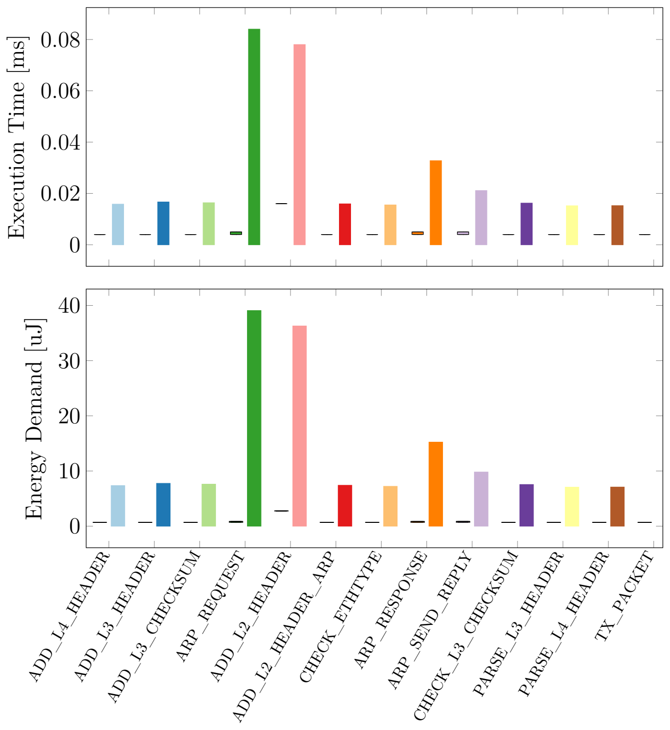

Fig. 3: Execution time and energy demand of transitions: measurements as box plots on the left, analysis bounds as a bar on the right for each transition.

B. *Evaluation Results*

Fig. 3 shows the timing and energy results of our evaluation. In our measurements, PFIP's transactions take less than 30 µs to execute and consume less than 10 µJ of energy. The measurement results are plotted as box plots, showing a very narrow interquartile range. Cross-validation with the chip's hardware timer confirms that most transactions take less than 5 µs. On the ESP32-C3, which is a relatively fast embedded system running at 160 MHz, there is a high probability of executing multiple transactions in succession. On less powerful chips, the granularity of our transactions ensures the ability to make progress, although the computations take longer.

Due to the analysis-aware implementation of PFIP, we are able to statically derive upper bounds on the resource consumption for all transitions (except `T_TX_PACKET`). The energy bounds account for the worst case of ongoing background activity. All values represent overestimations of the observed values. The largest bounds of all transactions are 84 µs and 39 µJ. In summary, our evaluation shows that the variation in the UDP-packet processing delay and energy consumption can be eliminated on embedded systems.

VI. Related Work

To our knowledge, PFIP is the first approach that targets the property of transactional semantics for a UDP/IP network stack. Nevertheless, several works on energy-constrained and real-time systems inspired our work, outlined in the following.

Energy-Constrained Systems: For embedded devices, IP implementations such as uIP [9] and lwIP [10] are available. However, they were not designed for intermittent systems. In contrast to these stacks, PFIP introduces transactional semantics to the network stack. Analyzing the energy demand of individual transactions enables us to give guarantees that they do not face a power failure once they are started.

Communication under Intermittency: De Winkel et al. [2] introduce the FreeBie communication stack for intermittently-powered Bluetooth Low Energy (BLE) devices. Routing problems for BLE-mesh–based intermittent systems were explored by RICS [14]. With PFIP, we follow a similar path of checkpointing the system's stack; however, with the difference of considering Wi-Fi communication. Behnke et al. proposed modifications to existing IP stacks to differentiate between traffic flows and ensure progress guarantees under heavy incoming traffic [15]. PFIP complements these approaches since it also guarantees progress during packet processing. In general, networking stacks for intermittent systems require protocols designed with intermittency in mind [16], [17].

Predictable (Real-Time) Systems: Related to the aspect of giving runtime guarantees are real-time systems where guarantees on the timely execution are crucial. Similar to timeliness in real-time systems, our goal is to guarantee the execution of transactions within given energy budgets during runtime. In the context of real-time systems, Schoeberl et al. introduced the TPIP stack [11] for time-predictable real-time systems. PFIP and TPIP share several design considerations, such as statically bounded loops in the control flow or the avoidance of dynamic memory allocations. However, TPIP is not directly suited for intermittently powered systems and the requirement of checkpointing their state, in contrast to PFIP.

VII. Future Work & Conclusion

An interesting avenue for future work is the open MAC implementation for the ESP32 processor family [18], [19]. Our current implementation for the RISC-V–based ESP32-C3 platform has to rely on a comparably large binary blob for transmitting packets from the platform's vendor. The open MAC project [18] managed to successfully reverse engineer the hardware for the Xtensa-based ESP32 for sending and receiving Wi-Fi packets. However, the open-source stack is not (yet) capable of handling the ESP32-C3 platform. Having deeper access to the platform's Wi-Fi stack presumably yields more fine-grained energy-budget handling. In addition, future work will explore how to include more complex networking semantics, e.g., TCP. It remains an open question how application-related aspects can be propagated through the communication stack. This is also related to our current assumption of synchrony, which we strive to loosen in our future work.

In this paper, we introduced our idea of PFIP, a UDP/IP network stack with the target of having transactional semantics. We use Petri nets to model the network stack's distinct states and transactions. By exploiting worst-case analysis techniques, we are able to give runtime guarantees on the uninterrupted execution of states/transitions.

The prototype implementation of PFIP*'s is available:*
https://git.nt.uni-saarland.de/open-access/pfip

Acknowledgments

The author order corresponds to the SDC (*sequence determines credit)* model. This work is supported by the German Research Foundation (Deutsche Forschungsgemeinschaft, DFG) as part of SPP 2378 (Resilient Worlds: project number 502615015, *Resilient Power-Constrained Embedded Communication Terminals, ResPECT*).

References

[1] S. Ahmed, B. Islam, K. S. Yildirim, M. Zimmerling, P. Pawełczak, M. H. Alizai, B. Lucia, L. Mottola, J. Sorber, and J. Hester, "The internet of batteryless things," *Communications of the ACM*, vol. 67, no. 3, pp. 64–73, 2024.

[2] J. de Winkel, H. Tang, and P. Pawełczak, "Intermittently-powered bluetooth that works," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '22)*, MobiSys '22, (New York, NY, USA), pp. 287–301, Association for Computing Machinery, 2022.

[3] P. Raffeck, J. Maier, and P. Wägemann, "WoCA: Avoiding intermittent execution in embedded systems by worst-case analyses with device states," in *Proceedings of the 25th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '24)*, 2024.

[4] M. Nardello, H. Desai, D. Brunelli, and B. Lucia, "Camaroptera: A batteryless long-range remote visual sensing system," in *Proceedings of the 7th International Workshop on Energy Harvesting & Energy-Neutral Sensing Systems (EnsSys '19)*, pp. 8–14, 2019.

[5] K. Vogelgesang, P. Raffeck, P. Wägemann, T. Herfet, and W. Schröder-Preikschat, "WIP: Towards a transactional network stack for power-failure resilience," in *Proceedings of the 21st IEEE Consumer Communications & Networking Conference (CCNC WiP '24) - Work-In-Progress*, 2024.

[6] J. L. Peterson, "Petri nets," *ACM Computing Surveys (CSUR)*, vol. 9, no. 3, pp. 223–252, 1977.

[7] Y.-T. S. Li and S. Malik, "Performance analysis of embedded software using implicit path enumeration," in *ACM SIGPLAN Notices*, vol. 30, pp. 88–98, ACM, 1995.

[8] P. Puschner and A. Schedl, "Computing maximum task execution times: A graph-based approach," *Real-Time Systems*, vol. 13, pp. 67–91, 1997.

[9] A. Dunkels, "Design and Implementation of the lwIP TCP/IP Stack," *Swedish Institute of Computer Science*, vol. 2, no. 77, 2001.

[10] A. Dunkels, "uip-a free small tcp/ip stack," *UIP*, vol. 1, pp. 1–17, 2002.

[11] M. Schoeberl and R. U. Pedersen, "tpip: A time-predictable tcp/ip stack for cyber-physical systems," in *2018 IEEE 21st International Symposium on Real-Time Distributed Computing (ISORC)*, pp. 75–82, IEEE, 2018.

[12] Espressif Systems, *ESP32-C3 Series Datasheet*, 2022.

[13] E. J. Maroun, E. Dengler, C. Dietrich, S. Hepp, H. Herzog, B. Huber, J. Knoop, D. Wiltsche-Prokesch, P. Puschner, P. Raffeck, M. Schoeberl, S. Schuster, and P. Wägemann, "The platin multi-target worst-case analysis tool," in *Proceedings of the 22nd International Workshop on Worst-Case Execution Time Analysis (WCET '24)*, 2024.

[14] G. Liu and L. Wang, "Routing for intermittently-powered sensing systems," in *2023 IEEE International Performance, Computing, and Communications Conference (IPCCC)*, pp. 274–282, IEEE, 2023.

[15] I. Behnke, C. Blumschein, R. Danicki, P. Wiesner, L. Thamsen, and O. Kao, "Towards a real-time iot: Approaches for incoming packet processing in cyber–physical systems," *Journal of Systems Architecture*, vol. 140, p. 102891, 2023.

[16] S. Fu, V. Narayanan, M. L. Wymore, V. Deep, H. Duwe, and D. Qiao, "No battery, no problem: Challenges and opportunities in batteryless intermittent networks," *Journal of Communications and Networks*, vol. 25, no. 6, pp. 806–813, 2023.

[17] A. Torrisi, K. S. Yıldırım, and D. Brunelli, "Reliable transiently-powered communication," *IEEE Sensors Journal*, vol. 22, no. 9, pp. 9124–9134, 2022.

[18] ESP32 Open MAC Contributors, "esp32-open-mac: Reverse engineered wifi driver for the esp32." 2024.

[19] J. Devreker, "Unveiling secrets of the esp32: creating an open-source mac layer." 2024/12.