WiFröst: Bridging the Information Gap for Debugging of Networked Embedded Systems

Will McGrath[1,2], **Jeremy Warner**[1], **Mitchell Karchemsky**[1], **Andrew Head**[1], **Daniel Drew**[1], **Bjoern Hartmann**[1]

[1]UC Berkeley EECS Department
{jwrnr,mkarch,andrewhead,
ddrew73,bjoern}@berkeley.edu

[2]Stanford University
Computer Science Department
wmcgrath@stanford.edu

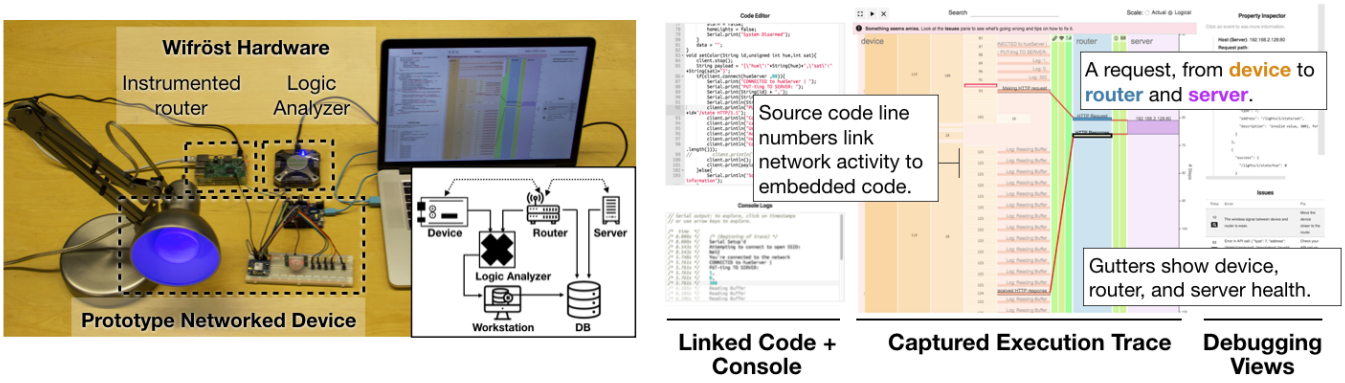

Figure 1. ***Left:*** **Debugging a networked embedded application with WiFröst: Developers connect their wireless device to an instrumented WiFi router; data from the device and router is captured at microsecond-level resolution by a logic analyzer.** ***Right:*** **The WiFröst UI has 5 main panels: a code editor; a console log; a visualization of the captured execution trace; a property inspector which displays contextually relevant information depending on active UI element, like the duration of a line of code, or the parsed content of an HTTP response; and an issues list that automatically displays notifications about system behavior irregularities or errors, e.g., errors returned in the response of a web API call, or loss of connection to the router.**

ABSTRACT

The rise in prevalence of Internet of Things (IoT) technologies has encouraged more people to prototype and build custom internet connected devices based on low power microcontrollers. While well-developed tools exist for debugging network communication for desktop and web applications, it can be difficult for developers of networked embedded systems to figure out why their network code is failing due to the limited output affordances of embedded devices. This paper presents WiFröst, a new approach for debugging these systems using instrumentation that spans from the device itself, to its communication API, to the wireless router and back-end server. WiFröst automatically collects this data, displays it in a web-based visualization, and highlights likely issues with an extensible suite of checks based on analysis of recorded execution traces.

Author Keywords

Internet of Things; Debugging; Embedded Systems; IDE

UIST '18 October 14–17, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5948-1/18/10.

DOI: https://doi.org/10.1145/3242587.3242668

INTRODUCTION

The rise of the Maker movement and associated prototyping tools like the Arduino platform [30], together with the emergence and popularization of the Internet of Things (IoT) has led to a large base of people working on networked embedded systems projects. Many such projects—from commercial products to hobbyist devices—follow an MGC (embedded-gateway-cloud) pattern [29, 11]: one or more embedded devices communicate with a back-end service hosted remotely on the internet through a gateway like a smartphone or home WiFi router. Example devices include mobile payment terminals, smart lights and speakers, and home weather monitors: all involve embedded code, network connectivity, and information received from a back-end server.

Wireless modules with associated software libraries as well as IoT-focused APIs like Particle Cloud [33] and Adafruit IO [21] have decreased the knowledge barrier for incorporation of connectivity in an embedded systems project; nevertheless, there is still a significant amount of difficulty associated with development and debugging. For example, when developing a device such as a WiFi-connected scale, developers have to solve issues like establishing and maintaining local network connectivity as the device is moved through the house and sleeps/resumes to preserve battery life; dealing with potentially unreliable connections between a local network and the

internet; and implementing and debugging API calls to a back-end on a device with limited hardware resources and limited debugging support. In other words, the concerns cross all levels of the conceptual OSI model of networking [39] and are exacerbated by the difficulties of embedded development.

Years of building and teaching students to build such systems lead us to believe that this difficulty arises from a combination of the following factors: *1) lack of visibility into measurements of interest, and how they interrelate.* Software-specific errors, hardware signals, and network traffic are accessible only by using separate tools (e.g., code IDE, multimeter, packet capture) or not at all (e.g., "black-box" gateway devices and wireless chips); *2) mixed time domains of interest*, where important aspects of the system occur on different time-scales between the relatively fast microprocessor execution speed, relatively slow user input, and varying network traffic rate and processing time; and finally, the *3) broad requisite knowledge base*, where users may face a large body of obstacles including compiler-specific, hardware component-specific, and networking-specific (e.g., HTTP or REST API) errors, each with their own associated error codes.

Our hypothesis is that **providing the information relevant to understand the behavior of a networked embedded system in a single linked environment can allow users to debug more efficiently through holistic methods (e.g. pattern recognition) and also help preemptively identify problem areas in system behavior across domain boundaries**. A key technique is to combine information from different domains (from code to network packets and server events) in a joint visualization that allows users to traverse domain boundaries as they seek to understand problems. A key insight is that instrumentation of the network gateway device provides a good deal of both control flow information as well as specific error information (e.g. through remote API call returns).

We introduce WiFröst, a debugging interface for networked embedded systems. WiFröst operates on traces of embedded device application, router, and server measurements that can be collected and shown either in real-time or offline. WiFröst focuses on the common pattern of devices communicating with HTTP REST interfaces. WiFröst specifically addresses the needs of intermediate to expert programmers seeking to develop and debug networked embedded systems projects by:

- Providing a unified visualization and exploration environment for data from instrumentation of the entire system stack (device, router, server), so that users who have a correct mental model of the desired behavior of their system can quickly test hypotheses and evaluate failure modes based on pattern recognition and situational context across levels of application and communication infrastructure.
- Preemptively checking for common errors as well as providing in-situ explanations for error types at different levels (e.g., WiFi connection errors, HTTP error codes, or API usage errors) in order to decrease the knowledge required for both localizing and interpreting bugs particularly for less-experienced users.
- Revealing information at the communication interfaces between domains such as device RSSI, which is typically outside the scope of standard development tools.
- Letting developers explore traces both as logical events or based on absolute timing.

In this paper, we first situate our contribution with respect to related work, then introduce the user experience of working with WiFröst, followed by our implementation strategy and a discussion of limitations of our approach.

RELATED WORK

We first discuss studies of general debugging techniques and end-user debugging patterns that informed the design of our system, then cover tools for debugging specific aspects of networked embedded systems.

General Debugging Techniques and Patterns

Studies of people during physical computing tasks [6, 31] and formative interviews with domain experts [12] support the hypothesis that one of the largest difficulties when debugging an embedded systems project is correctly localizing the bugs. Users often misdiagnose error sources across domains, for example believing a problem originated from a software error when it actually had to do with their physical circuit. Gugery and Olson [16] show that novice programmers have difficulty generating high-quality hypotheses about faults.

Several research tools introduce techniques to aid debugging: DejaVu[22] and d.tools [19] include visualizations for interpreting time-series sensor data. The Whyline [24, 25] and Timelapse [7] enhance debugging by providing interfaces to examine and interrogate recorded program behavior. Gneiss [9] helps users make more effective use of the structured data provided by web APIs.

Implications for Design: Our system should preemptively provide fault localization when possible and communicate where to start looking for failure points by annotation or highlighting when automatic localization is impossible and support flexible navigation through program traces.

Embedded Systems and Electronics Debugging

The concept of "always-on" visualizations has been shown to improve debugging performance in code [26]. The Toastboard [12] provides preemptive error checking of breadboarded circuits given a user-supplied virtual schematic to compare measurements against. Bifröst [28] extends checking across the domains of software and hardware by instrumenting both user code as well as the output pins of an embedded system; it also provides an extensively linked visualization environment. Scanalog [37] and Splish [23] both change the physical computing workflow by providing a block diagram-based environment for instrumentation and development.

Several tools allow developers to track resource usage in embedded systems. The *nesC* language [15] includes provisions for hardware resource aware error checking of user code at compile time. Quanto [13] tracks energy usage of wireless sensor motes (typically "invisible" information). It performs accurate hardware measurements of power consumption and

ties them to programmer activity through an instrumentation API spanning the hardware and software domains.

Implications for Design: The knowledge barrier for working with embedded systems and electronics is decreased by combining information from multiple domains into a single environment. Cross-domain tools must take into account disparate time scales of interest; there are also often domain-specific resource constraints that should be conveyed to the user.
The Contribution of WiFröst: We will preemptively provide root cause analysis for behavior errors that manifest across the domains of embedded system software, network routing, and server communication. Passive instrumentation of the user device, router, and server will provide insight into previously hidden barriers to successful operation of a *networked* system (e.g. RSSI, CPU load) in a single development environment.

Network and Distributed System Debugging

The most relevant work concerns trace-based debugging of networks and distributed systems.

The *ndb* network debugging tool [17] extends techniques from software debugging (e.g. breakpoints, steps) to the realm of software-defined networks. It uses passive instrumentation of the network to reconstruct control flow, allowing users to trace network errors back through the sequence of events leading up to them. The NetSight platform (which uses *ndb*) [18] supports aggregation and filtering of packet histories through a network; they are explicitly grouped based on passively generated "postcard" instrumentation. Whodunit [8] provides a similar concept of "transaction flows," where information is tracked across access tiers (e.g., from client to server).

X-Trace [14] provides a network debugging tool that crosses network layers and administrative boundaries. OFRewind [38] implemented record-replay debugging of traces across network layers with a focus on low-overhead instrumentation functionally separate from the data plane. Pinpoint [10] specifically uses an instrumented middleware component to collect traffic data and control flow traces to perform root cause analysis of distributed systems problems.

Some systems include visualization methods for collected control flow and behavioral data. Dapper [36] provides additional information about performance of distributed systems using a low-level, transparent instrumentation scheme for low-overhead trace collection and annotation. It includes a visualization of call trees alongside annotated execution traces containing information about timing. Pip [34] provides a language for writing checks on system behavior across recorded traces that decreases the amount of data a user has to parse to localize invalid behaviors; it also includes a GUI for search tree-based visualization of these checks. Spectroscope [35] is a tool to evaluate distributed system request flows and compare them over time; it explicitly looks for differences between captured behaviors and displays annotated side-by-side visualizations for user review. Unravel [20] provides an interaction visualization of behavior (i.e. changes between captured traces, function calls) in the realm of web development. VEE [27] built an interface for summarizing the communications between groups of physical devices. These projects make use of the now common techniques of brushing and linking [4] and multiple levels of zoom as in [5].

Implications for Design: Grouping information into transactions/histories across levels allows for richer filtering and evaluation of trace data. Instrumentation overhead on individual packets must be minimized for scalable implementations. For visualization, stacked models and explicitly linked transactions can show the user useful control flow information and allow them to debug behavioral differences based on pattern matching, without having to parse dense data.
The Contribution of WiFröst: We extend the concept of using transactional histories within recorded behavior traces for debugging by *including control flow within the software on the embedded device*. Our visualization is designed with *an emphasis on interaction instead of passive analysis*.

A GATEWAY TO WIFRÖST

In this section we illustrate some of the uses of WiFröst using a running example and refer to the layers of the conceptual OSI model of networking [39] that are addressed in each step.

Wendy is a Maker; she is building a keycoded alarm for one of her windows that can control her smart-lights wirelessly. When Wendy is home, her house lights stay neutral white. She can "arm" her alarm, which sets the lights to purple and periodically checks a sensor to ensure that no one has broken in. If the window is opened, her code moves into a triggered state and changes the lights to orange. The intruder now has 30 seconds to input a keycode which is authenticated against a remote key server. If the server returns a `200 OK` message, the user has a valid keycode and the alarm system is put back into the "home" state. If they do not disable the system in time, it moves to an alarm state with a flashing red light.

After drawing a schematic of the components (Fig. 2a), she begins development. She can write, compile, and upload code to the microcontroller from inside the WiFröst IDE (Fig. 2b).

The first failure Wendy faces is at the physical connection to the network (OSI Layers 1 and 2). Wendy sets up her WiFi device to connect to her router but uses incorrect credentials to connect. WiFröst's checkers alert Wendy that the router rejected her requests because her password was incorrect. By clicking the issue, Wendy zooms the timeline to the lines of code and network events where this issue was detected (Fig. 2d), and pulls up the relevant lines of code executed when that issue was detected. Without WiFröst, her Arduino WiFi library would only alert her that she was unable to connect to the given network; the WiFröst instrumented router allows Wendy to see specifically which part of the credentials were incorrect—the password—as opposed to other errors (e.g., if her router was not powered, she would receive the same error on the microcontroller as having an incorrect password).

With the router connection established, the next step is to establish a connection to a web application (OSI Layers 3 and 4). When Wendy writes a line of code to send an API call to a server, she notices that although her request is sent, it is not reaching the intended destination. The core Arduino WiFi library would just tell Wendy that she was unable to connect to the server; WiFröst removes ambiguity by showing that

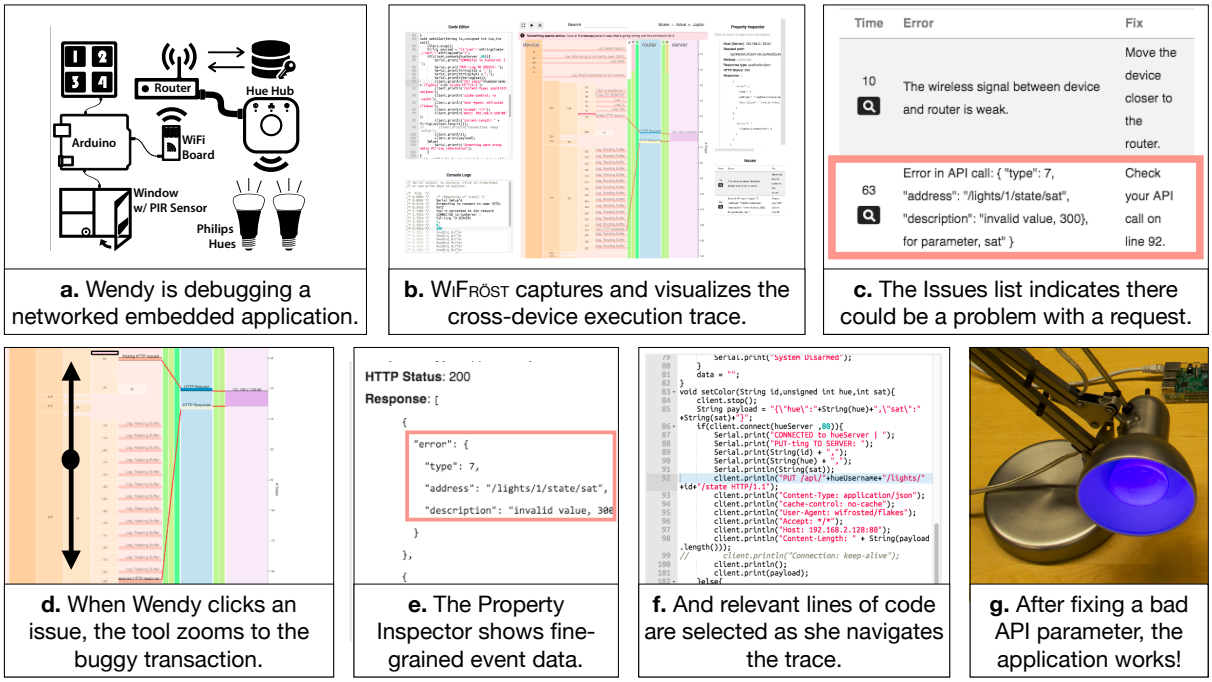

Figure 2. An example workflow of using WiFröst to debug an embedded networked application.

her message has reached the router but not server, displaying a red health bar for internet connectivity between the two, and warning Wendy in the Issues pane that her home network was not connected to the internet at the time of the request.

With the internet connection re-established, Wendy attempts to make an HTTP request to her application server (OSI Layer 7), only to encounter another error. Without WiFröst, the WiFi library would still show the same error — that she is unable to connect to the server. Instead, WiFröst's library instrumentation is able to see that the DNS lookup for her URL failed, and WiFröst precisely localizes and highlights the error via an automated check.

Finally able to send requests to the API over HTTP (OSI Layer 7), the device sends several requests before hitting yet another issue. WiFröst alerts Wendy that an error was detected in the response received from the web API—while the response carried a `200 OK` status code, WiFröst detected an `error` key in the response with a non-null value (Fig. 2c). Wendy takes a closer look at the error by clicking on the router response event to load the response content into the property inspector (Fig. 2e). Reading the response, the culprit is easy to see—an invalid value was passed as an argument to the API. By clicking on the response event, the IDE's linked navigation has already highlighted the line of code invoking the transaction (Fig. 2f). Wendy traces the code back a few lines from the request to the definition of the parameter, and fixes the value to the range the API expects. Having inspected and fixed each of these errors, Wendy's embedded networked application is now connected to the web and running as expected (Fig. 2g).

DEBUGGING WITH WIFRÖST

WiFröst's instrumentation and visualization infrastructure provides both explicit localization of faults for the user and facilitates effective use of standard debugging techniques.

Holistic Debugging Assistance

By connecting visualizations of network events and the embedded software execution in a single environment, we allow users to differentiate and compare behavior of the entire system based on context and visual patterns.

Visualizing Event and Transaction Context

WiFröst provides five simultaneous views of the same underlying trace: code, log output, event properties, an issues list, and the cross-device control flow visualization. Each of these is more than a static view: interactions with one view updates the focus of all others based on the relevant transaction history (Fig. 3). Despite a large amount of measured data presented in our IDE, these interactive techniques for conveying event context should decrease user cognitive load by guiding attention within the interface.

Due to the disparate time scales (e.g. microsecond embedded code versus millisecond server requests) involved in a networked embedded system, many of the interesting events that occur during an application trace—log messages, HTTP responses, logically important lines of code—are obscured when the events are visualized proportionally to their execution time. Trace recordings can often contain long empty spans where no events occur and short bursts of activity where many events overlap; this requires a user to zoom out to navigate between clusters of events to gain context, but zoom in to understand the actual contents of those clusters. To increase the legibility

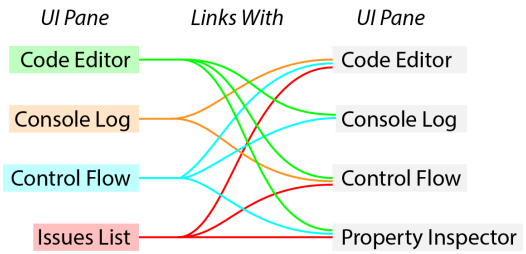

Figure 3. This graph shows the interactive linking between interface elements. For example, clicking on an output line in the serial log will both highlight the line of user code that created that output line and refocus the visualization to show the relevant transaction.

of the trace, the visualization includes a "logical" time scale where all events are given roughly the same height and empty time spans are eliminated; the trace is then reported in terms of chronological steps instead of absolute time (Fig. 4).

At-A-Glance Health Indications

Sometimes, failure of a networked embedded system is due to externalities either outside of direct user control or outside the scope of a user's ordinary monitoring capabilities. WiFröst displays various domain boundary and resource conditions, like RSSI, ping, and embedded device RAM usage, alongside the measured trace (Fig. 5). These values are updated continuously during trace recording to reflect their dynamic nature.

This information can both help prevent error misdiagnoses (e.g. if a failure to receive information from the server is due to a lost internet connection instead of coding error) and help explain behavioral differences between different traces (e.g. low RSSI during one interaction, but not another, affecting system response time).

Transaction Grouping for Pattern Recognition

Behavior of a networked embedded system can be expected to follow either one-way (e.g. an HTTP POST with no required response) or bi-directional set patterns. By grouping information into transactions across the domains of the device code and network activity, we can draw explicit connections between related events; users can then use these connections as tools for diagnosing incorrect behavior based on expected visual pattern (Fig. 6).

Pre-emptive Fault Localization

WiFröst helps users locate the causes of unexpected behavior by detecting and flagging unexpected behaviors from the trace

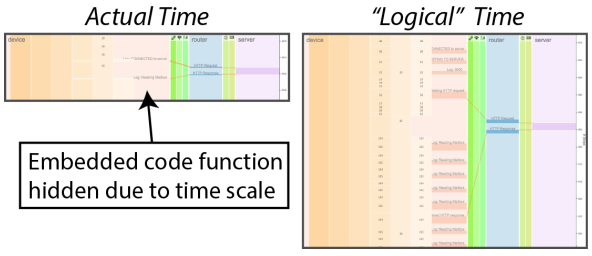

Figure 4. WiFröst makes it possible for users to switch between visualization in actual time for overall context and a discretized "logical" time to view detailed program behavior through buttons in the interface.

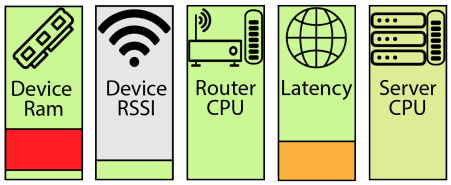

Figure 5. "Health" conditions are monitored periodically (e.g. device RAM), with the value then mapped to a color range and displayed alongside the relevant events. Domain boundary and resource conditions are displayed in their logical locations between the relevant domains.

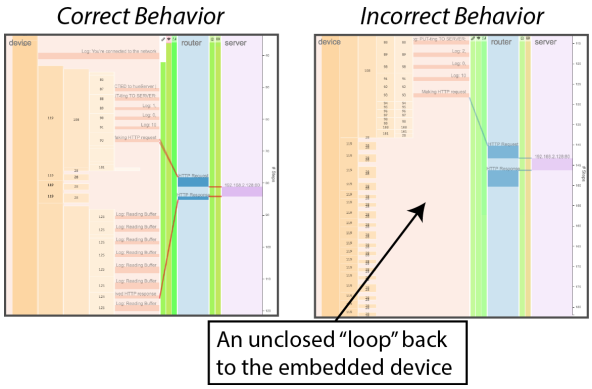

Figure 6. Visual comparison of traces containing both expected and unexpected behavior can yield insight into where the transaction broke down. Here, we can quickly see that although the HTTP request response made it back to the router, it did not get to the device. Closer inspection shows that in the incorrect behavior trace the user has forgotten to read from the receive buffer; the call to that function is missing in the embedded software stack visualization.

logs and offering concrete tips on how to fix these behaviors. Every time a new event is queued in the event stream, WiFröst re-runs a suite of error checkers on the history of events; see Fig. 7 for a list of the currently implemented checkers. Errors are often localized to a single event. For example, one checker reports an error whenever it observes a server response containing a `401` code. Other checkers inspect sequences of events across devices; to detect an incorrect WiFi network password, one checker searches for failure codes returned by the WiFi library, followed by a router health record where a connection is unauthorized. Some checks detect rising edges; this is especially important for health checks, where the data is collected periodically, but only a subset of those events should

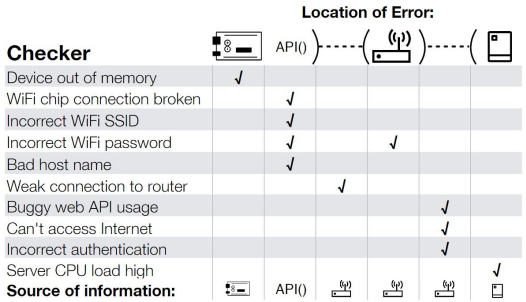

Figure 7. WiFröst monitors the data collected from the device, WiFi API, router, and server to detect common errors. This plot shows the checkers implemented in WiFröst, and the data streams that are monitored.

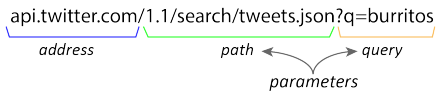

Figure 8. A web API call contains multiple parts. Through the instrumentation of the router WiFröst can not only identify that an API call is an error source in the system but also identify the specific *part* of the call that has caused the problem; for example, an incorrect address versus a set parameter not found or outside an allowable range.

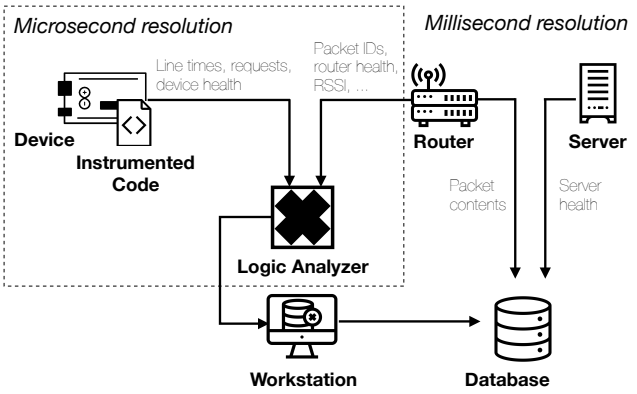

Figure 9. Dataflow through the WiFröst network components. Logs are collected from instrumented code, device, router, and server. The method of data collection varies by the time resolution required to preserve causation between events from neighboring components.

be flagged as "errors". For example, one checker for flagging weak connections between the device and router will only report an error when the RSSI falls below a threshold.

Calls to remote APIs can fail for a variety of reasons (Fig. 8), but visibility into these reasons is limited for a typical developer of an embedded device. The WiFröst instrumentation of the core WiFi library lets us parse and display relevant packet information as it passes through the instrumented router, giving visibility into *why* and *when* the API call failed.

IMPLEMENTATION

WiFröst relies on software and hardware instrumentation to capture a program's trace across the network stack, and a data pipeline to clean, align, and display this information. In this section, we describe the high-level design of the system (Figure 9), and the technical insights that allowed us to collect and display information for networked application debugging.

Collecting a Network Trace from Hardware and Software

Instrumenting the User's Application Code

To link network events and console output to concrete lines of code, and to report device health information, application code must be instrumented to collect line execution times and device health. WiFröst instruments a user's application source code to automatically collect this information, by running a custom ANTLR-generated [32] parse tree walker over the code. The walker initializes serial output at the start of the program's `setup` function. Before every statement in the program, the walker introduces a `Serial.write` statement to report the line that is about to execute. Inside the `loop` function, the walker adds code to query a utility library (in our case, "MemoryFree" Arduino library [2]) for the available RAM left on the device, and writes the result to the serial port. Generalizing this instrumentation approach to non-Arduino platforms is discussed in Limitations.

Instrumenting a Standard Networking Library

The WiFröst interface shows an event whenever a device makes a request and whenever it receives a response. It also flags whenever WiFi initialization has failed and detects the causes of these failures. To collect information for these events and checks, we instrumented the standard networking library for Arduino—"WiFi101" [1]. The library was extended to log all calls to its initialization and request functions, and to log error codes handled in these functions. Future tools like WiFröst could apply similar instrumentation to networking libraries through wrapping or source code modification.

Instrumenting a Maker's Router

Many confusing problems in a networked application manifest outside the device where a request originates, like failures due to weak signal strength and errors on a server. To detect such problems we introduce an instrumented wireless router; in our implementation, a Raspberry Pi. The instrumentation continually logs all incoming requests, server responses, signal strength (RSSI) with connected devices, router authorization failures, and the round-trip time of a ping to a well-known DNS server. While some of this information could be collected on a device (e.g., HTTP requests and responses), reporting this information from the router has the added advantage that it does not slow down a user's application code, it only introduces a small amount of network latency.

Cross-Device Transaction Detection

In the WiFröst interface, a user can visually trace a network "transaction" from initiation in the client API code, to handling in the router and server, to its reception back on the device. It is not possible to reliably link events into transactions using event timing alone—devices often make multiple overlapping requests, which can be fulfilled out of order. We further instrumented the networking library and router to associate a persistent transaction ID with each event in a transaction. When a user initiates an HTTP request using the networking library, the library appends a unique transaction ID to the headers. The router passes this header onto the server. When the server returns a response, the router automatically associates the response with the request, tagging the two with the same transaction ID. The router updates the headers of the response returned to the device to include this ID.

Putting it Together: Cross-Device Time Synchronization

WiFröst's capture infrastructure faces a unique challenge: code-triggered events occur at the resolution of microseconds, so events on the device and router must be synchronized with microsecond-level precision to capture the causality of events. Typical techniques for synchronization like NTP [3], only ensure synchronization to a precision of milliseconds, even when two computers are directly connected over a network link.

To synchronize log data with microseconds-level precision, all log data from both the device and the router is transmitted over

megabaud-rate serial connections to a single Digilent Digital Discovery 2 logic analyzer. The logic analyzer is read via a USB connection to a local workstation and then the data is written to the WiFröst database. Network packets (e.g., HTTP requests and responses) are often kilobytes in size – too large for real-time processing by the logic analyzer. Therefore, whenever the router registers a network event, it sends a packet ID over serial to log the packet's timing, and then uploads the packet's payload asynchronously to the database over HTTP.

For an instrumented web API server, the milliseconds-resolution synchronization provided by NTP with the device and router is sufficient since the network delays are much larger than the clock error.

A Framework for Error Localization and Description

A WiFröst checker is defined by four rules, and can often be implemented in around 30 lines of code. First, a checker needs a rule that iterates over all events and returns locations of events including errors. Second and third, a checker includes a rule to describe the error, and potential fixes, which can be parameterized with event and line number data. Fourth, a checker can define how highlight and zoom to relevant events when the error is selected from the issues browser.

LIMITATIONS

Hardware Limitations:
Our current implementation of WiFröst only supports the Arduino Zero and its default WiFi library to connect to our software-defined router in order to get man-in-the-middle access to network traffic. However, our ANTLR instrumentation could easily be modified to process C or C++ code used by other embedded platforms, so long as the particular chip has a spare hardware UART to use as a output channel. Likewise, our on-device networking instrumentation code is only around 200 lines and could be adapted to other libraries where the `read` and `write` primitives can be wrapped or modified.

We rely on a commercial logic analyzer to simultaneously collect trace information from both the embedded device and software router in order to precisely synchronize their time domains. This requires physical connections to both devices, which limits where devices can be placed during testing.

Software Limitations:
In our current implementation, we only display a single embedded device communicating to a router and then the internet. IoT device developers are often interested in the flow of information from many user devices to a backend service. From an instrumentation perspective, it is straightforward to instrument and capture instruction streams from multiple devices simultaneously. However, it remains an open question how to visualize multiple related traces in an effective manner. Although it wouldn't be difficult to allow a user to select a single pair of devices of interest to display among several captured in the current UI, this is still point to point and wouldn't allow the user to easily see how groups of devices were interacting.

WiFröst currently only focuses on HTTP(S) transactions to ports 80 and 443 and REST interfaces. While this captures a great number of current IoT use cases, two logical next steps would be to add support for raw TCP/UDP socket communication and for additional protocols like MQTT. While the network capture and protocol parsing components would require changes, the core visualization approach conceptually supports these protocols.

Our instrumentation impacts runtime performance, which could negatively impact the behavior of time-critical code. We have observed a maximum 3x slowdown on the Arduino Zero for line execution logging. The slowdown is dominated by the imbalance between the rate that the processor fills up the UART output buffer and the rate it is emptied (2MBaud).

Trade-offs in Checks:
Our suite of general-purpose checks rely on relatively simple heuristics. Most APIs will respond with a 400 class error in the case of an client error, but some APIs will respond with a `200 OK` regardless of call success. We generalize by looking for the keyword "error" in the JSON response, but this does not cover all cases for all APIs. It is possible for users to author their own more precise application or API-specific checks by writing code using our framework. However, we expect a GUI-based checker authoring system would enhance usability.

Since WiFröst tags and visualizes communications across the network, the visualization is more sparse when communication fails to be sent at all. However, so long as any code is running on the device, the "Device" pane of the UI will be populated. In this way a user can observe their program's flow and get insight into why certain code paths might not have been run.

CONCLUSIONS AND FUTURE WORK

A structured set of debugging tasks could provide insight into UI usage patterns and help us assess the overall effectiveness in helping programmers successful debug their applications. A more open-ended development task or workshop could guide the design of features beyond affordances for localizing errors.

In some cases, the end user may be developing a server back-end in conjunction with the embedded device. Instrumenting back-end server code in a similar way to how we currently instrument embedded software could yield more insight into failures that occur as a result of server-side problems. This approach would result in an even higher amount of information density in the user interface and likely require exploring new visualization and filtering techniques to enhance clarity.

Our system spans from user embedded code to communication with a networked server. Incorporating the additional domain of physical measurements alongside user code (as in Bifröst [28]) would let users perform root cause analysis of errors manifesting at the highest levels of abstraction, e.g., a server database value entry, down to the lowest level of behavior, e.g., measurement noise in a voltage on a pin. Such a tool would completely bridge the gap between physical electronics and web applications, dramatically changing the development and debugging workflow of Makers, students, and professional developers for the better.

ACKNOWLEDGEMENTS

This work was supported in part by NSF awards CNS 1505728 and IIS 1149799.

REFERENCES

1. Arduino – wifi. https://www.arduino.cc/en/Reference/WiFi. Accessed 04-04-2018.
2. Available memory – arduino playground. https://playground.arduino.cc/Code/AvailableMemory. Accessed 04-04-2018.
3. Ntp: The network time protocol. http://www.ntp.org/index.html. Accessed 04-04-2018.
4. Becker, R. A., and Cleveland, W. S. Brushing scatterplots. *Technometrics 29*, 2 (1987), 127–142.
5. Bederson, B. B., Meyer, J., and Good, L. Jazz: an extensible zoomable user interface graphics toolkit in java. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*, ACM (2000), 171–180.
6. Booth, T., Stumpf, S., Bird, J., and Jones, S. Crossed wires: Investigating the problems of end-user developers in a physical computing task. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ACM (2016), 3485–3497.
7. Burg, B., Bailey, R., Ko, A. J., and Ernst, M. D. Interactive record/replay for web application debugging. In *Proceedings of the 26th annual ACM symposium on User interface software and technology*, ACM (2013), 473–484.
8. Chanda, A., Cox, A. L., and Zwaenepoel, W. Whodunit: Transactional profiling for multi-tier applications. In *ACM SIGOPS Operating Systems Review*, vol. 41, ACM (2007), 17–30.
9. Chang, K. S.-P., and Myers, B. A. Gneiss: spreadsheet programming using structured web service data. *Journal of Visual Languages & Computing, http://www.sciencedirect.com/science/article/pii/S1045926X16300994* (2016).
10. Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. Pinpoint: Problem determination in large, dynamic internet services. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, IEEE (2002), 595–604.
11. Chiang, H., Hong, J., Kiningham, K., Riliskis, L., Levis, P., and Horowitz, M. Tethys–an energy harvesting networked water flow sensor. In *Proceedings of the Third International Conference on Internet-of-Things Design and Implementation*, ACM (2018).
12. Drew, D., Newcomb, J. L., McGrath, W., Maksimovic, F., Mellis, D., and Hartmann, B. The toastboard: Ubiquitous instrumentation and automated checking of breadboarded circuits. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, ACM (2016), 677–686.
13. Fonseca, R., Dutta, P., Levis, P., and Stoica, I. Quanto: Tracking energy in networked embedded systems. In *OSDI*, vol. 8 (2008), 323–338.
14. Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, USENIX Association (2007), 20–20.
15. Gay, D., Levis, P., Von Behren, R., Welsh, M., Brewer, E., and Culler, D. The nesc language: A holistic approach to networked embedded systems. *Acm Sigplan Notices 49*, 4 (2014), 41–51.
16. Gugerty, L., and Olson, G. Debugging by skilled and novice programmers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '86, ACM (New York, NY, USA, 1986), 171–174.
17. Handigol, N., Heller, B., Jeyakumar, V., Maziéres, D., and McKeown, N. Where is the debugger for my software-defined network? In *Proceedings of the first workshop on Hot topics in software defined networks*, ACM (2012), 55–60.
18. Handigol, N., Heller, B., Jeyakumar, V., Mazières, D., and McKeown, N. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *NSDI*, vol. 14 (2014), 71–85.
19. Hartmann, B., Klemmer, S. R., Bernstein, M., Abdulla, L., Burr, B., Robinson-Mosher, A., and Gee, J. Reflective physical prototyping through integrated design, test, and analysis. In *Proceedings of the 19th annual ACM symposium on User interface software and technology*, ACM (2006), 299–308.
20. Hibschman, J., and Zhang, H. Unravel: Rapid web application reverse engineering via interaction recording, source tracing, and library detection. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, ACM (2015), 270–279.
21. Industries, A. Adafruit io. https://io.adafruit.com/, 2018.
22. Kato, J., McDirmid, S., and Cao, X. Dejavu: integrated support for developing interactive camera-based programs. In *Proceedings of the 25th annual ACM symposium on User interface software and technology*, ACM (2012), 189–196.
23. Kato, Y. Splish: a visual programming environment for arduino to accelerate physical computing experiences. In *Creating Connecting and Collaborating through Computing (C5), 2010 Eighth International Conference on*, IEEE (2010), 3–10.
24. Ko, A. J., and Myers, B. A. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, ACM (New York, NY, USA, 2008), 301–310.
25. Ko, A. J., and Myers, B. A. Finding causes of program output with the java whyline. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2009), 1569–1578.

26. Lieber, T., Brandt, J. R., and Miller, R. C. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*, ACM (2014), 2481–2490.

27. Marquardt, N., Gross, T., Carpendale, S., and Greenberg, S. Revealing the invisible: visualizing the location and event flow of distributed physical devices. In *Proceedings of the fourth international conference on Tangible, embedded, and embodied interaction*, ACM (2010), 41–48.

28. McGrath, W., Drew, D., Warner, J., Kazemitabaar, M., Karchemsky, M., Mellis, D., and Hartmann, B. Bifröst: Visualizing and checking behavior of embedded systems across hardware and software. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, ACM (2017), 299–310.

29. McGrath, W., Etemadi, M., Roy, S., and Hartmann, B. Fabryq: Using phones as gateways to prototype internet of things applications using web scripting. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ACM (2015), 164–173.

30. Mellis, D., Banzi, M., Cuartielles, D., and Igoe, T. Arduino: An open electronic prototyping platform. In *Proc. CHI*, vol. 2007 (2007).

31. Mellis, D. A., Buechley, L., Resnick, M., and Hartmann, B. Engaging amateurs in the design, fabrication, and assembly of electronic devices. In *Proceedings of the 2016 ACM Conference on Designing Interactive Systems*, ACM (2016), 1270–1281.

32. Parr, T. J., and Quong, R. W. Antlr: A predicated-*LL(k)* parser generator. *Software - Practice and Experience 25*, 7 (1995), 789–810.

33. Particle. Particle cloud. https://www.particle.io/, 2018.

34. Reynolds, P., Killian, C. E., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A. Pip: Detecting the unexpected in distributed systems. In *NSDI*, vol. 6 (2006), 9–9.

35. Sambasivan, R. R., Zheng, A. X., De Rosa, M., Krevat, E., Whitman, S., Stroucken, M., Wang, W., Xu, L., and Ganger, G. R. Diagnosing performance changes by comparing request flows. In *NSDI*, vol. 5 (2011), 1–1.

36. Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Technical report, Google, Inc, 2010.

37. Strasnick, E., Agrawala, M., and Follmer, S. Scanalog: Interactive design and debugging of analog circuits with programmable hardware. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, ACM (2017), 321–330.

38. Wundsam, A., Levin, D., Seetharaman, S., Feldmann, A., et al. Ofrewind: Enabling record and replay troubleshooting for networks. In *USENIX Annual Technical Conference* (2011), 15–17.

39. Zimmermann, H. Osi reference model–the iso model of architecture for open systems interconnection. *IEEE Transactions on communications 28*, 4 (1980), 425–432.