Modular Network Stacks in the Real-Time Executive for Multiprocessor Systems

Vijay Banerjee, Sena Hounsinou, Harrison Gerber, Gedare Bloom
Department of Computer Science
University of Colorado Colorado Springs
Colorado Springs, Colorado, USA
(vbanerje, shoueto, hgerber, gbloom)@uccs.edu

***Abstract*—Real-Time Executive for Multiprocessor Systems (RTEMS) is a real-time operating system used by the Experimental Physics and Industrial Control System (EPICS) open-source software for high-precision scientific instruments such as particle accelerators and telescopes. EPICS relies on the networking capabilities of RTEMS for microcontrollers that need to meet real-time constraints. However, the networking available in RTEMS either lacks the necessary drivers to be fully operational or lacks security features required in modern networks. In this paper, we introduce a modular networking architecture for RTEMS by separating the network software implementation and device drivers from the RTEMS kernel to provide them as a static library for applications to use. This networking-as-a-library concept provides application developers with better capabilities to select the network features needed for their target application and to keep their networking software updated and secure.**

***Index Terms*—RTEMS, EPICS, TCP/IP, Networking, lwIP, Modular Networking Stack, FreeBSD, libBSD**

I. Introduction

The US Department of Homeland Security (DHS) Cybersecurity and Infrastructure Security Agency (CISA) defines over a dozen Critical Infrastructure sectors. Each sector regroups a set of systems and assets that support communities. For instance, the Information Technology (IT) sector comprises systems that support information-based society. This sector also supports other critical infrastructure sectors because it encompasses the industrial control systems (ICSs) that monitor processes across cyber-physical infrastructures. As such, ICSs play a vital role across domains like healthcare, manufacturing, production, and research and development. These systems rely on the security of the networks that connect their components, which sometimes are located in different geographic areas. Recent events such as the Colonial Pipeline cyberattack [1] have shown that a single security breach can impact an entire region of the United States.

The Experimental Physics and Industrial Control System (EPICS) is an open-source scientific cyberinfrastructure that is used in particle physics research and development. Specifically, EPICS enables creating distributed real-time control systems for scientific instruments such as particle accelerators, telescopes, and other large scientific experiments. As in other ICSs, secure communication between different nodes is important to EPICS overall security posture. EPICS depends on the networking implementations provided by the OS. One of the OSs used by EPICS is the Real-Time Executive for Multiprocessor Systems (RTEMS) [2].

This work is supported by NSF OAC-2001789 and CO State Bill 18-086.

As a POSIX-compliant real-time OS (RTOS), RTEMS can support multiple *network stacks*. A network stack is the implementation of the set of networking protocols that handle the transfer of data across connected devices. The network stack, sometimes called the TCP/IP stack, follows a four-layer model [3] where the application is at the top layer and the device specific network drivers are at the bottom layer, which interfaces the hardware, network protocols such as the Internet Protocol (IP), and the OS.

Traditionally, the network stack implementation is a part of an OS that handles the networking tasks and the respective drivers for a network interface controller (NIC). The TCP/IP stack implementation of RTEMS historically also resided in the kernel along with the user-level application programming interface (API) declarations that RTEMS provides through the *Newlib* C library.

Prior to our work, the RTEMS networking stack implementation, which we now call the *legacy stack*, faced the following challenges: first, the legacy stack lacked several features that have become fundamental in modern networks. For example, the legacy stack is based on an earlier, less secure version of the Internet Protocol (IPv4), which is no longer suitable for many systems supported by RTEMS. This is particularly important for US federal agencies and research centers (including national laboratories using EPICS), which have been mandated to transition to a more secure network by 2023 [4]. However, it is challenging to update the legacy stack because it is integrated in the RTEMS kernel source code. Therefore, updating the network stack requires updating the entire RTEMS kernel. Although a newer FreeBSD-based stack known as the *libBSD stack* is equipped with more features (including security-related ones), it lacks drivers for commonly used EPICS microcontrollers. Moreover, the size of the applications linked to the libBSD stack is unsuitable for some memory-constrained hardware supported by RTEMS.

In this work, we separate the legacy stack in its own module outside RTEMS to facilitate switching the network stack without requiring heavy changes to user applications or the RTEMS kernel. This modularization uses extant Newlib header files to allow RTEMS users to build and link their applications to their network stack of choice, much like they can select among several different scheduling algorithms depending on the application needs [5]. This modular *Networking-as-a-Library* framework also provides users with an option to build their own implementation of a network stack for RTEMS instead of depending on the legacy implementation provided by the OS. This approach provides two major benefits to EPICS. First, applications are not restricted in terms of choice of networking features, because they can link to a different network library without changing much of the code. Second, the network library provides an opportunity to upgrade to more secure, modern network stacks.

Our contributions are summarized as follows:

- We create a standalone network stack for the legacy networking implementation that was a part of RTEMS.
- We create a modular network stack library for lwIP TCP/IP stack.
- We develop a *net-services* submodule that consists of the common network services that any networking module can use.
- We demonstrate the working of a user application with the modular network stack to show that the modular approach does not change the application layer. Hence, our approach does not burden existing users with the need to change their application.
- We evaluate the stacks in terms of Round Trip Time (RTT) to provide insights on performance implications of each network stack.
- We compare the size of the binary images from each of the stack and present the *rtems-lwIP* stack as suitable alternative to legacy network stack.

To the best of our knowledge, these contributions make RTEMS the first full fledged RTOS that allows the flexibility in choosing among multiple networking stacks. This flexibility makes RTEMS one of the most adaptable RTOS for real-time system developers by providing them the option to select network features that are specifically targeted toward their needs, hence enabling tradeoffs in performance (memory consumption, bandwidth, latency) and security.

The remainder of the paper is as follows: we next provide background about RTEMS and its network stacks in Section II. Then we describe the modular network stack framework in Section III. In Section IV we provide an analysis of the network stacks, and in Section V we discuss the current work related to modular network stacks. Finally, we conclude the paper in Section VI where we also briefly discuss our ongoing and future efforts.

II. Background

RTEMS is an open source real-time OS. As such, systems built using RTEMS have both temporal and logical correctness requirements. In addition, because the RTOS supports various size target platforms across different architectures (e.g., ARM, Motorola 6800, and SPARC), developers have endeavored to use code that can suit embedded and resource-constrained devices. In the early years of RTEMS, such code has been ported from the lightweight C library Newlib [6], which is another open source project focused on providing POSIX-compliant cross-compiled software which is widely used in embedded system projects. Specifically, RTEMS adopted several enhancements provided by Newlib (e.g., floating point support, and math library [7]) including header files [8] for which RTEMS added its own implementation. Some of these header files were used as a foundation for porting RTEMS network stacks. We provide more explanation regarding this in Section III.

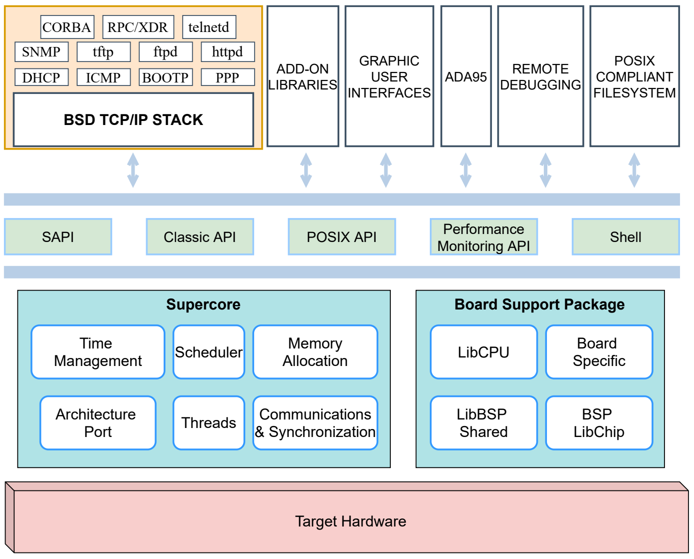

Fig. 1. RTEMS High Level Architecture

The RTEMS kernel (depicted in Figure 1) is composed of four main blocks: the Supercore, the board support package (BSP), the application programming interface (API), and the Services. The Supercore is the heart of the kernel. It provides the real-time functionalities of the OS. As the name indicates, the BSP is responsible for providing all the necessary support to integrate the different hardware targets supported by RTEMS. The API block allows RTEMS users to access the functionalities of the Supercore. RTEMS user-level services range from enabling programming in multiple languages to accessing additional libraries, including the newly added RTEMS network stack.

RTEMS provides a legacy implementation that is built into the RTEMS kernel. This legacy stack has been ported from the earlier versions of FreeBSD and has been a part of RTEMS since the late 1990s. FreeBSD integrated DARPA's TCP/IP stack [3] in its early network stack implementation. The stack generically regroups the OSI communication model into four layers: the application layer of the stack (which combines the session presentation and application layers of the OSI reference model) transmit user application data to the transport layer using sockets API. The transport layer is implemented using TCP and UDP protocols over the IP (or Internet) layer. The Network access layer (which integrates the data link and physical layers of the OSI model) is the lowest layer of the network stack. It handles the physical hardware and protocols that are required to deliver the data across a physical network. This handling is done through the device drivers in RTEMS, which are responsible for initializing and operating the embedded hardware's NIC.

A network application uses the networking APIs, like the socket API, to make system calls with the appropriate protocol headers, which triggers the network drivers to send physical signals to the hardware to carry out the requested action. Traditionally, the networking implementation is a part of an OS that handles the networking tasks and the respective drivers for a NIC. The implementation also provides user-level header files that contain the declarations for the user APIs.In RTEMS the POSIX networking API signatures are provided to the applications through the *Newlib* C library, and the implementation of the TCP/IP stack along with the NIC drivers, were a part of the RTEMS legacy stack.Although the RTEMS legacy stack has been used by multiple targets for a long time, it did not evolve at par with the developments in the FreeBSD stack due to the following reasons: first, making changes inside the kernel requires significant time and expertise. Next, making any change to the legacy network stack was essentially a change to the RTEMS kernel, which involves a lot of regression testing.

In addition to Newlib, RTEMS also makes use of FreeBSD's code base, similar to several other well-known OSs. FreeBSD [9] is also another open source OS, known for its high performance in modern systems. A RTEMS repository named *rtems-libbsd*, or the libBSD module was built by RTEMS developers to port the required codes from FreeBSD, which also includes the API implementation for the Newlib header files. The libBSD module uses a git submodule to track the upstream FreeBSD source code. LibBSD uses Python scripts to port specific files from this FreeBSD submodule as follows: first, a block of FreeBSD source code is imported from the submodule. Then, the necessary files are copied locally to the RTEMS-libBSD repository, and adapted to work with RTEMS through the scripts which not only imports the code, but also adds RTEMS specific header files to them, in order to properly connect the FreeBSD drivers to the RTEMS kernel. (We refer to this approach as the libBSD framework in the remainder of the paper.)

In recent years, the RTEMS developers have used the libBSD framework to import the FreeBSD's TCP/IP stack, providing the users with an option to use a modern and secure FreeBSD network stack with their RTEMS applications. The LibBSD, which uses the FreeBSD network stack, has a complete IPv6 support along with robust security features [10]. The modern features present the libBSD stack as a great upgrade option to a more modern stack. One caveat to having libBSD as the only alternative to the legacy stack is that some targets have very limited available memory and are not capable of running the libBSD stack. Thus, we prepared an adapter version of the lwIP [11] stack as an alternative to the libBSD. lwIP is an independent project targeted towards embedded systems with strict memory constraints. The features of lwIP stack are comparable to that of libBSD but size and memory requirements are much smaller than that of an application linked with libBSD. Some of the important highlights of the lwIP stack are the much required IPv6 support and the support for IPSEC, which has been studied and added by other independent projects [12].

In the following section, we present a modular network stack approach that decreases the reliance on the legacy stack and provides additional network stack options. This approach also gives RTEMS users the ability to develop more suitable network stacks without the need to modify the entire RTEMS kernel.

III. Modular network stacks

As stated in Section I, RTEMS users currently face the following challenges related to the implementation of the network stacks: (1) difficulty upgrading the legacy stack, (2) inability to fully utilize each of the existing network stacks (legacy and libBSD) because of a lack of appropriate drivers, (3) lack of security support in the legacy stack. To address the first challenge, we separated the components of the legacy stack from the current RTEMS kernel into its own standalone repository (see Section III-A).

To resolve (2), we have also separated the drivers from the RTEMS kernel and added them as a part of the networking module. In addition, we created a standalone submodule called *rtems-net-services*, that can be added to any RTEMS network stack to use networking services like the File Transfer Protocol (FTP) and the Trivial FTP (TFTP). These services are available for use by any network stack module (see Section III-D).

Moreover, as a part of our ongoing effort to expand support for all the network stacks, we have streamlined the workflow for adding support for a particular hardware platform. We demonstrated the workflow and experiments on an uCdimm ColdFire 5282 Microcontroller Unit (uC5282) which is widely used by EPICS for RTEMS-based projects.

To address the third challenge, we look to use a network stack implementation that provides modern security features such as IPv6. The lwIP network stack implementation [11] matches such a requirement. In addition to IPv6, lwIP can also be combined with other independent protocol implementations like embedded IPSec [13]. lwIP in combination with embedded IPSec has been evaluated with microkernel OS [12], showing that lwIP can be robust and versatile in adding security updates. Moreover, the lwIP network stack is targeted towards embedded systems with strict memory constraints. As such, we broadened the existing network stack options by fully integrating a third network stack module based on lwIP. The lwIP based networking stack, called, *rtems-lwip*, will enable the users to choose the network stack that provides the necessary security required for the application (see Section III-C).

As a result, a new architecture is obtained for the network stack library, as shown in Figure 2. In the following subsections, we describe how the stacks and the net-services module

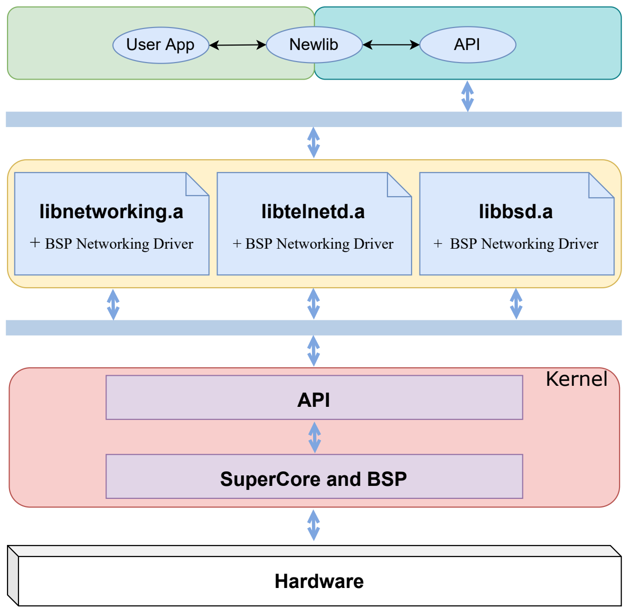

Fig. 2. RTEMS with Modular Network Architecture

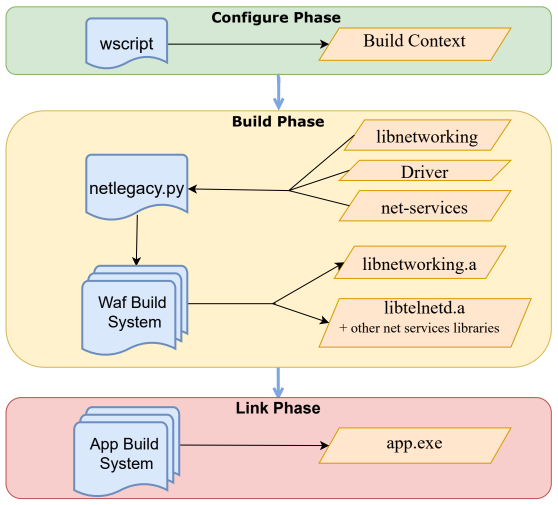

Fig. 3. Modular Network Stacks Build Process

were built to form the network stack library in further detail. Table I shows a comparison of the features of the network stacks.

TABLE I
COMPARING NETWORK STACK FEATURES

Feature	LibBSD	lwIP	legacy
TCP	✓	✓	✓
UDP	✓	✓	✓
IPv4	✓	✓	✓
IPv6	✓	✓	×
IPSec	✓	✓	×

A. *Legacy Networking Module*

As stated in Section II, any significant modification to the legacy stack requires extensive changes to the kernel. Thus, to ensure that any update still satisfies and preserves the current networking functionalities of RTEMS, and to make the legacy stack available to the projects that are actively using it (without any change in their project), we opted to separate the legacy stack from the core of the OS in the form of a static library (*libnetworking.a*). A static library allows us to treat the existing network stack as a separate unit in the OS, without requiring the code to be built along with the OS kernel. Building the static library requires that we slightly modify the current flow of the RTEMS Networking, and link the *libnetworking.a* library to the user application directly. In the new separate legacy stack, we followed the same directory structure that was maintained in RTEMS, to make it easier for the interested developers to maintain this stack separately, without having to adapt to a totally different organization of the same codebase.

The implementation of the legacy stack resided inside the RTEMS *cpukit/* directory while the BSP-specific drivers for the legacy networking were located inside the *bsps/* directory for each of the hardware targets supported by RTEMS. To proceed as described above, we created the *rtems-net-legacy* repository [14] to hold the TCP/IP implementation file and BSP drivers.

At compile time, the user application statically connects to the libnetworking.a library which contains the legacy TCP/IP implementation along with the BSP networking drivers used by the legacy stack. To provide a simple and easy to integrate system for developers, we opted for the Waf build system [15]. The Waf system was selected majorly because it is written in a general purpose scripting language (Python). As such, a developer only needs to focus on the functional code, saving time on build system changes.

The new build process comprises three stages (Configuration, Build, Link) as shown in Figure 3. During the Configuration stage, Newlib provides the networking API header files used by the user application. Then a Waf script (*wscript*) collects the build context which consists of the target name, toolchain executable locations, build flags, and other environment variables. In the Build phase, a script (*netlegacy.py*), which we added to the repository, uses this build context and collects the required files to build. The selection of files is important in order to ensure that the correct driver gets linked according to the build context. The linked driver gets connected to the RTEMS kernel through the *bsp.h* header file, which is a BSP-specific header file present in all the RTEMS BSPs. This header contains the macro defines for *RTEMS_BSP_NETWORK_DRIVER_ATTACH* which declares the name of the driver attach function that the BSP will call, in order to initialize the network interface. The driver attach function is defined in the *libnetworking.a* library that is generated from the waf build. This library is then linked to the user application in the Link phase. Since the end product from the Build phase is a separate C library, the user can use their own build system to link to the library.

B. LibBSD Module

RTEMS uses this LibBSD framework to port the current TCP/IP network stack from the FreeBSD sources, which provides a full featured IPv6 [16] supported network stack. To accommodate this BSD stack addition to RTEMS without changing the RTEMS source code significantly, the TCP/IP API header files required for the current BSD stack are pushed into upstream Newlib repository under the directory *libc/sys/rtems/include/*. Similar to the legacy networking process, the LibBSD builds a static library *libbsd.a* from the FreeBSD ported codes. This separate library is especially interesting for the network stack, as the networking API declarations are provided by Newlib while the implementation is obtained from libBSD. Using this approach, any application that makes use of the latest BSD networking can simply link to *libbsd.a*.

C. liblwIP Module

The lwIP stack has been used in multiple embedded OS projects, such as FreeRTOS and HelenOS. In the RTEMS community, some users have also individually developed RTEMS drivers in order to support the lwIP TCP/IP stack for their projects. For example, the "uLan protocol for RS-485 9-bit network" project [17] has adapted the lwIP stack for RTEMS along with a driver for their target board ARM based TMS570. There are multiple such independent projects that are using the lwIP TCP/IP stack, but developments made on these projects are unavailable for an RTEMS user out of the box. To address the issue of scattered lwIP drivers, and to provide an alternative to the *libbsd* and *legacy* network stacks, we built a standalone networking module for RTEMS that can act as a centralized location for drivers, hardware abstraction layers (HAL), and adaptations that were developed by independent projects to use the lwIP stack with RTEMS.

The newly created *rtems-lwip* repository [18] also uses the Waf build system and has the same modular structure as the previous two stacks. We have also added the upstream *lwIP* repository as a submodule to *rtems-lwip*. This submodule tracks the upstream changes, making it easier to update to the latest changes without having to start a whole new self-hosted independent project. Along with our adaptation of the lwIP stack, we have added drivers developed by Texas Instruments for the ARM-based BeagleBone Black board to test the build process.

D. Net-Services Submodule

To further push for modular options, we moved some of the common net services (for example, *tftpfs* and *telnetd*) from the RTEMS kernel into a separate repository designated *rtems-net-services* that acts as a submodule to *rtems-net-legacy*. This new submodule builds static libraries (such as *libtftpfs.a* and *libtelnetd.a* for ftpfs and telnetd respectively) for the networking services.

The creation of the *rtems-net-services* submodule demonstrates that placing the RTEMS networking services in a module is both effective and maintains usability and function as it does not add any exra layer of build process for the user. Users who rely on the RTEMS networking services can build only the services they need, reducing executable size since these services are no longer in the kernel.

IV. Evaluation

In this section, we present three evaluations of the RTEMS modular network stack framework presented in Section III. In the first experiment (in Section IV-A), we show that the modular networking framework does not require substantial effort from the user. To do so, we demonstrated the building of the the *rtems-net-legacy* module using the *waf* system.

In the second experiment (Section IV-B), we analyzed the memory requirements to implement each of the three network stacks (legacy, libBSD, lwIP). Specifically, we computed and compared the size of the binaries of the same application over the legacy and libBSD network stacks. This experiment highlights the memory requirements between the stacks, and shows the potential implications of switching from the legacy stack to the *rtems-libbsd* stack.

In the final experiment (Section IV-C), we evaluated the round trip times (RTT) of the *rtems-libbsd* and *rtems-net-legacy* stacks Although the lwIP network stack is fully integrated and adapted to RTEMS, the driver support is limited for an RTT analysis. Therefore, the lwIP stack has not been used in this experiment.

For all three experiments, we have selected the uCdimm Coldfire 5282 (uC5282) as our hardware target due to its wide use in projects that deploy EPICS and RTEMS for safety-critical applications. The uC5282 microcontroller module uses the Motorola MCF5282 microcontroller, that has an integrated 10/100 Fast Ethernet Card. The uCdimm platform has an on-board Synchronous Dynamic Random Access Memory of 16MB.

A. Building a Classic Application Using Net-Services

In this section we illustrate the process of building a simple Round-Trip Time application using the *legacy* network stack. The application building follows a two-step process as described in section III-A. The network stack is configured for the target hardware with the command shown in Figure 4, then built using the *'./waf'* command.

```
rtems-net-legacy $>./waf configure \
> --prefix=$RTEMS_PREFIX \
> --rtems-bsps=m68k/uC5282
```

Fig. 4. Command to configure and build *rtems-net-legacy* stack for uC5282

For rapid functionality testing of the network stacks on uC5282, we have also provided QEMU emulator [19] support for the uC5282 target. The current main branch of QEMU does not have support for the target board so we refactored an old patch [20] to make the board compatible with the current QEMU. We will contribute this added support to QEMU upstream.

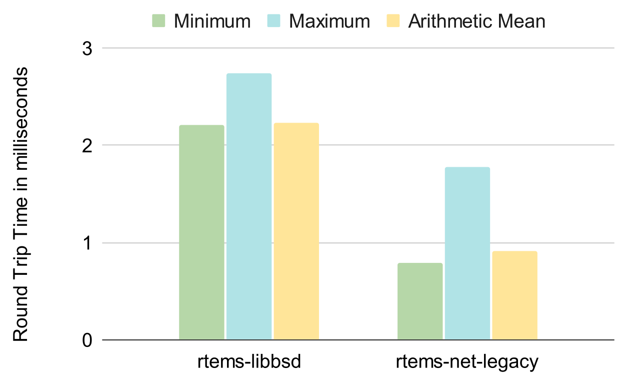

Fig. 5. Round trip time comparison of the RTEMS network stacks

B. Size Comparison of Binary Images

To compare the sizes of the binary images of the three network stacks, we used the same RTT application that we built in the experiment in IV-A. We used the GNU *objcpy* and *size* tools from the GNU toolchain for the *m68k* target, to get the binary images of the executable linked to different stacks, along with the size of *text*, *data*, and *bss* segments to understand the memory usage of the apps in each network stack.

Table II shows the results from the comparison of the size of the generated binary images. The size difference between the *libbsd* stack and other two stacks is significant. However, the size difference between the *lwIP* stack and *legacy* stack is much lower. The sizes of the *.text* segment (which in all three cases represents the largest portion of the executable) show that the libBSD brings in a much larger code, making the executable much larger compared to the other two. The *.data* segment shows a similar pattern where, interestingly, the lwIP stack has the lowest value. This low value is due to the optimized memory design of the lwIP stack which enables it to run on targets as low as 512kB of memory. The *.bss* segment in the lwIP stack can be reduced by allocating even lower memory to the lwIP configuration, which can be done in the *rtems-lwIP* repository through the *lwipopts.h* header. This similarity in the size of the lwIP and legacy stack shows that *rtems-lwip* can be a suitable alternative to the *legacy stack* for memory-constrained targets.

TABLE II
SIZE COMPARISON OF BINARY IMAGES (ALL VALUES IN KB)

Network stack	.text	.data	.bss	Total Size
rtems-libbsd	1273	58.4	24	**1,332**
rtems-net-legacy	244.4	6	44	**250.5**
rtems-lwip	293	1.7	59	**294**

C. Round Trip Time Analysis

The RTT analysis shows the *latency* of the network, which gives an idea about how much time it takes for a packet to be transferred. A comparison of the RTT over the loopback device shows the latency from the network stacks only, without other factors that can affect the latency, like the wiring and routing overhead due to the connection between devices.

To eliminate any performance difference due to different hardware NIC speeds, we executed our application on the same uC5282 hardware with each network stack. Since the uC5282 lacked a driver for *rtems-libbsd*, we added this support by porting the legacy networking driver to the libbsd stack, which we will contribute to the upstream *rtems-libbsd* repository.

To compare the RTT, we created a lightweight application that sends a constant size packet over Internet Control Message Protocol (ICMP) using raw sockets. The ICMP header size is $28B$ and we added a *padding* buffer of $56B$ to send a total of $84B$. From the recorded data over 10 runs (see Figure 5), we noted that the LibBSD stack has a latency overhead which can be approximately double the average latency from the legacy stack. This observation gives an interesting insight that switching an application to the FreeBSD-based *libbsd* stack will have a performance overhead that can accumulate every time a packet is sent or received. This overhead might become critical in high precision industrial controllers where the latency of the network can impact the validity of observed values. The latency analysis reinforces the need for a lightweight network stack alternative, which will be available to the user through the rtems-lwip module.

V. Related Work

A modular network stack approach has been previously attempted on microkernel OSs like HelenOS [21] where each part of the network stack works as a server module for the microkernel proof-of-concept implementation. In contrast, our work is based on a Monolithic Real-Time kernel, where we have implemented the whole network stack as a separate library module that gets linked into one whole executable binary which is run on the target hardware.

NetBSD also uses a modular TCP/IP stack implementation through a rump Kernel TCP/IP [22] that virtualizes kernel functional units into clients. The clients can be one of three types: local, microkernel or remote. The local client type approach uses rump kernel as a library with rump API calls. Instead of adding a new API layer, our approach provides support for the common API calls for multiple stacks and an application does not require any change in terms of includes and API calls for working with an RTEMS networking library.

We have extended our unique approach to add a totally independent networking stack lwIP, which has been used in RTOSs before [23], [24], but our work differs in two ways. First, the independent networking implementation has not been integrated into the kernel in contrast with the FreeRTOS TCP/IP implementation which is part of the kernel. Second, the networking module provides a framework for adding and modifying any layer of the network stack without affecting the main kernel, which will pave the way for support on a wider range of architectures and NICs.

VI. Conclusion and Future Work

We have separated the legacy network stack from RTEMS and created a standalone modular network stack as a separate library that can be linked to user applications directly. We have also created a networking library module for the lwIP stack, which allows a platform to integrate independent lwIP drivers for use with an RTEMS-based application. The new modular networking architecture will facilitate switching from one network stack to another and provides an easier route to develop and use custom network stacks. Such stacks can seamlessly fit as modules on top of RTEMS without changing the kernel.

Our analysis of the three network stacks (legacy, libBSD, and lwIP) provides insights on their features and performance. The features of the libBSD and lwIP stacks are comparable and much ahead of the RTEMS legacy stack. However, the legacy stack shows lower latency when compared to the libBSD stack. The analysis of memory requirements show that for memory-constrained platforms the lwIP stack is a better alternative to the legacy stack than the libBSD stack. Future work can improve our benchmarking analysis by considering the throughput from each stack along with evaluating dynamic memory allocation for size-based comparisons among the network stacks.

To further facilitate the development of multiple network stacks without losing support for the common network services, we have created a networking submodule that contains popular network services including TFTP and telnetd. We have thoroughly tested these services with RTEMS. This submodule can be added to any networking stack thus providing a common framework of services and testing. Future work can add sample applications to the rtems-net-services submodule that can be used by any network stack.

With our ongoing efforts, we intend to add more device drivers to the lwIP stack to support a wider range of architectures, and seek to understand how best to integrate it with multicore targets [25]. We plan to utilize this modular architecture to extend support for secure network services like the Secure Shell Protocol (SSH), which is an important tool for securing communications especially in industrial Internet-of-Things (IIoT) [26] control systems. In addition, to facilitate the development and adaptation of a network stack over a wider range of targets, we plan to develop a library for networking driver modules.

References

[1] A. Hobbs, "The colonial pipeline hack: Exposing vulnerabilities in us cybersecurity," 2021.

[2] G. Bloom, J. Sherrill, T. Hu, and I. C. Bertolotti, *Real-Time Systems Development with RTEMS and Multicore Processors*. CRC Press, Nov. 2020. [Online]. Available: https://www.taylorfrancis.com/books/9781351255790

[3] B. Beranek, "A history of the arpanet: the first decade," *Technical report*, 1983.

[4] "Executive order on improving the nation's cybersecurity," 2021. [Online]. Available: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/

[5] G. Bloom and J. Sherrill, "Scheduling and Thread Management with RTEMS," *SIGBED Rev.*, vol. 11, no. 1, pp. 20–25, Feb. 2014. [Online]. Available: http://doi.acm.org/10.1145/2597457.2597459

[6] C. Vinschen and J. Johnston, "The newlib homepage," 2018. [Online]. Available: http://sourceware.org/newlib

[7] W. Gatliff, "Porting and using newlib in embedded systems."

[8] C. Johns, J. Sherrill, B. Gras, S. Huber, and G. Bloom, "FreeBSD and RTEMS, UNIX in a Real-time Operating System," *FreeBSD Journal*, 2016. [Online]. Available: http://issue.freebsdfoundation.org/publication/?i=330348&article_id=2557258&view=articleBrowser

[9] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson, *The design and implementation of the FreeBSD operating system*. Pearson Education, 2014.

[10] "Freebsd security." [Online]. Available: https://docs.freebsd.org/en/books/handbook/security/

[11] A. Dunkels, "Design and implementation of the lwip tcp/ip stack," *Swedish Institute of Computer Science*, vol. 2, no. 77, 2001.

[12] M. Hamad and V. Prevelakis, "Implementation and performance evaluation of embedded ipsec in microkernel os," in *2015 World Symposium on Computer Networks and Information Security (WSCNIS)*. IEEE, 2015, pp. 1–7.

[13] N. Schild and C. Scheuer, "Embedded ipsec, light weight ipsec implementation," *Diplome Thesis, Berne Univ, Switzerland*, 2003.

[14] "Rtems net-legacy." [Online]. Available: https://git.rtems.org/rtems-net-legacy/

[15] [Online]. Available: https://waf.io/book/

[16] [Online]. Available: https://docs.freebsd.org/doc/6.0-RELEASE/usr/share/doc/handbook/network-ipv6.html

[17] [Online]. Available: https://sourceforge.net/p/ulan/lwip-omk/ci/master/tree/

[18] "Rtems net-legacy." [Online]. Available: https://git.rtems.org/vijay/rtems-lwip.git

[19] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 41.

[20] T. Straumann, "qemu + uc5282," 2009. [Online]. Available: https://lists.rtems.org/pipermail/users/2009-September/021089.html

[21] L. Mejdrech, "Networking and tcp/ip stack for helenos system," 2010.

[22] A. Kantee, "The design and implementation of the anykernel and rump kernels," *Aalto university*, 2016.

[23] "Porting lwIP - FreeRTOS." [Online]. Available: https://docs.aws.amazon.com/freertos/latest/portingguide/porting-lwip.html

[24] G. Bloom and J. Sherrill, "Harmonizing ARINC 653 and Realtime POSIX for Conformance to the FACE Technical Standard," in *2020 IEEE 23rd International Symposium on Real-Time Distributed Computing (ISORC)*, May 2020, pp. 98–105, iSSN: 2375-5261.

[25] D. Cederman, D. Hellström, J. Sherrill, G. Bloom, M. Patte, and M. Zulianello, "RTEMS SMP for LEON3/LEON4 Multi-Processor Devices," in *Data Systems In Aerospace*, Warsaw, Poland, Jun. 2014.

[26] G. Bloom, B. Alsulami, E. Nwafor, and I. C. Bertolotti, "Design patterns for the industrial Internet of Things," in *2018 14th IEEE International Workshop on Factory Communication Systems (WFCS)*, Jun. 2018, pp. 1–10.