Real-Time Bridging of I/O and Network Buses in Cyber-Physical Systems

Anton Njavro[0000−0003−2610−8020], Zhiyuan Ruan[0009−0008−2616−3162], and Richard West[0000−0001−5100−0666]

Boston University, Boston MA 02215, USA
{njavro,zruan,richwest}@bu.edu

Abstract. Cyber-physical systems (CPS) increasingly require real-time, high bandwidth data communication and processing. To address this, Time Sensitive Networking (TSN) provides latency-bounded data transmission at one or more gigabits-per-second throughput. However, it does not commonly connect directly to I/O devices, such as sensors and actuators. In contrast, Universal Serial Bus (USB) is ubiquitous for device I/O, but has yet to be widely adopted for host-to-host networking.
This paper considers the use of a common USB software stack for both device I/O and host-to-host communication. We compare against a system using USB for device I/O and TSN for host-level networking. Our findings show that a unified approach using USB results in reduced software complexity, simplified bus coordination, and more effective mitigation of priority inversion when transferring data across multiple bus segments. Experiments show that end-to-end latency is within expected delay bounds, and is reduced if the same USB software stack is used for all communication with a given host. This suggests that bridging challenges exist in current systems, which are solved by either extending a high-bandwidth bus such as TSN to support device I/O, or enhancing USB with improved networking capabilities.

Keywords: Cyber-Physical Systems · Universal Serial Bus · Time Sensitive Networking

1 Introduction

Increasingly complex cyber-physical systems (CPSs) [49,36] are emerging in domains such as automotive, industrial control, avionics and robotics. These systems often require high bandwidth, low-latency sensor data processing and control. This involves data collection from arrays of input devices (e.g., LiDARs and cameras, in the case of autonomous vehicles), and real-time processing on one or more compute nodes. Different compute nodes may interface with different devices needed for input and output, and then coordinate their data processing tasks by sharing information across a network.

Controller Area Networking (CAN) [31,14] has proven to be a dependable solution for information transfer between microcontrollers, but lacks the bandwidth

to meet the demands of modern cyber-physical systems. Ethernet [16,4,43,11], supports higher bandwidth but does not provide the tight timing guarantees required for real-time communication. Consequently, Time-triggered Ethernet [34] and now Time-Sensitive Networking (TSN) [20,40] have been developed as ways to guarantee communication delay constraints.

Originally applied to audio-video bridging (AVB), TSN is now being considered as a standard for use in automotive [10], aerospace [55] and industrial control applications. While it supports clock synchronization, bandwidth reservation, and predictable host-to-host networking, it does not provide a commonly adopted approach for communication with input/output (I/O) devices. Chip manufacturers have suggested the use of medium access control devices [17] as a means to connect microcontrollers to networks such as Ethernet and TSN. Until this technology is widely adopted, the problem of integrating high-bandwidth, low-latency sensors and actuators to compute nodes in emerging CPS domains is left to other bus technologies.

USB is a ubiquitous approach to connect I/O devices to hosts, although it is less commonly used for host networking. This is in part because it was not originally envisioned as a peer-to-peer technology. Hence, it would appear that TSN is preferred for host-to-host networking, while USB is better suited to device I/O. However, the use of two different buses leads to potential bridging problems at the host-level, to ensure traffic is able to disembark one bus in time to be ready for when the next bus is available for use. Coupled with the need for two different protocol stacks and device drivers, different packet formats, bus speeds, and arbitration protocols, there is often a delay moving traffic from one bus to another. This paper addresses the question: "if we used the same protocol stack for device I/O and host-based communication, would we be able to simplify coordinated data transfers, reduce latency and potentially improve throughput?" Consequently, we consider using USB as a unified solution for *both* host-to-host networking and device I/O, with the opportunity to schedule communication transactions across different buses.

Contributions. We use USB's debugging capability (xDBC) found in the xHCI specification, to achieve predictable host-to-host communication, while carefully synchronizing with the transfer of I/O device data. By applying our xDBC implementation to Linux, we show that a *USB-only* system exhibits *lower* worst-case end-to-end transfer latency than the best-case scenarios using *both* TSN for host networking and USB for device I/O. We then implement xDBC in the Quest real-time OS [53] to show how integrated USB networking and device I/O is able to reduce the end-to-end latency even further.

We believe that current cyber-physical systems would benefit from a single bus solution for coordinated time-sensitive, high bandwidth networking and device I/O. This suggests that either a bus such as TSN should be extended to communicate directly with sensors and actuators, or a device-centric bus such as USB should be enhanced with better networking features. For complex topologies spanning longer distances, combining USB and TSN might be preferred. USB could be used for clusters of nearby hosts close to devices, and TSN used

for back-end networking. This would allow for real-time data processing and control tasks to coordinate operations, with tighter time bounds, on hosts that are short hops away from sensors and actuators.

The rest of the paper is structured as follows: Section 2 provides background information. Section 3 then describes the challenges of a dual-bus design for device I/O and networking, and explains the proposed use of USB xDBC to solve these problems. The section is completed with a description of the latency models used to estimate the worst-case performance of unified (USB-only) versus dual (USB and TSN) bus communication. Section 4 presents our experimental evaluation for both unified and the dual-bus approaches. Related work is then discussed in Section 5, followed by conclusions and future work in Section 6.

2 Background

2.1 Universal Serial Bus

USB is a master-slave protocol that connects a host computer (the master) to one or more peripheral devices (the slaves). As of USB 3.0, a device operates at one of four possible communication rates: *low*, *full*, *high*, or *super* speed, with maximum throughput of 1.5 Mbps, 12 Mbps, 480 Mbps and 5 Gbps, respectively. Recent advances with USB 3.2 now increase bus bandwidth up to 20 Gbps, with USB 4 going as high as 40 Gbps. The effective bandwidth for a device depends on its speed, and how the system software programs the host controller to schedule USB packets on the corresponding bus instance.

Each USB device is defined by a set of descriptors that are readable by the host. These descriptors correspond to the *device*, its *configurations*, *interfaces* and supported *endpoints*. A device descriptor defines the number of configurations supported, amongst other information. A configuration descriptor defines the number of interfaces supported, as well as the maximum bus power consumed by the physical device. Only one configuration for a given device is active at any time. An interface descriptor specifies the number of endpoints as well as the function those endpoints serve. For example, a USB keyboard with an integrated mouse will have two interface descriptors, one each for the keyboard and the mouse. An endpoint descriptor specifies a maximum packet size, transfer type, transfer direction (input or output) and polling interval for periodic transfer of the endpoint.

There are four endpoint transfer types: (1) *Bulk transfers* - Used for lossless transmission of non-real-time data. A USB thumb drive is an example of USB device with a bulk endpoint; (2) *Isochronous transfers* - Used for loss-tolerant real-time data transmission. A USB camera typically comes with isochronous endpoints; (3) *Control transfers* - Used for lossless transmission of device configuration data; (4) *Interrupt transfers* - Used for lossless real-time data transmission. This type of endpoint is found on devices such as keyboards.

Bulk and control transfers are asynchronous, while isochronous and interrupt are periodic, with a defined interval between transactions. The specification

guarantees that high-speed periodic transfers are limited to 80% of a microframe, leaving at least 20% to asynchronous transfers. USB 3 superspeed periodic traffic occupies up to 90% of the microframe time, guaranteeing at least 10% for bulk and control data.

A USB host stack consists of a *host controller* and the software. Software encompasses device-specific drivers, a host controller driver, and various operating system interfaces for communication between devices and applications. USB transfers are always initiated by the host. Peripheral devices only respond to host requests. A transfer request from software on the host to a device might be carried out by several USB bus transactions. The number of transactions needed is decided by the maximum packet size of the endpoint and the number of bytes requested by the software. A transfer request of 1024 bytes, to an endpoint with maximum packet size of 512 bytes, requires two data transactions. Transactions are scheduled on the bus in frames of 1 ms with low and full speed devices, and in microframes of 125μs with high (USB 2.0) and super-speed (USB 3.x) devices. A transaction will not be scheduled by the host controller if it cannot be completed in the same frame or microframe.

2.2 USB Extensible Debug Capability (xDBC)

USB 3.x supports host-to-host communication using the xHCI built-in debug capability known as xDBC. This capability, originally designed as a debugging mechanism, allows for two USB host machines to communicate using a pair of super-speed bulk (IN and OUT) endpoints. *Software* techniques are able to reserve bandwidth for bulk transfers by restricting the traffic associated with periodic endpoints [69]. Alternatively, newer chipsets equipped with an eXtensible Device Controller Interface (xDCI) allow for dual role (host and device) controller switching, and customizable endpoint configurations for host-to-host communication. As we are yet to properly investigate xDCI capabilities, we restrict this paper to studying xDBC versus TSN.

2.3 Time Sensitive Networking (TSN)

TSN describes a set of standards to extend IEEE 802.1Q [2] Ethernet with clock synchronization and timing guarantees for message transfers. From a high-level perspective, the TSN standards of concern in this paper include: (1) *Traffic shaping and scheduling*, and (2) *Time synchronization*. TSN provides a Time Aware Shaper (TAS) and Credit-Based Shaper (CBS) to model and shape the traffic. Clock synchronization across all hosts in a network is achieved using the IEEE 802.1AS [3] Generalized Precision Time Protocol (gPTP).

This paper focuses on the IEEE 802.1Qbv Time-Aware Scheduler (TAS) [1]. TAS is based on time-division multiple access (TDMA), to allocate network bandwidth shares across different time slots. TAS supports three type of classes: A, B, and lowest-latency control data traffic (CDT) class. Three bits of the Priority Code Point (PCP) field in VLAN tags are used to assign protocol priorities. This limits the number of priority classes to 8, whereas in USB it is possible to

support up to 1024 [54]. TSN may, however, implement more service classes in software, within each of the 8 priority classes assigned using VLAN tags.

Figure 1 shows a schedule where each cycle contains both timing-critical and best effort traffic, each in its own time slice. To prevent best effort traffic from interfering with critical traffic, TSN uses what are called *Guard bands*. These are dedicated time windows before timing-critical scheduling periods, in which initialization of new best effort transmissions are prohibited. Guard bands ensure high-priority traffic is isolated from best effort transfers. However, they result in lost bandwidth, since the time used for a guard band is not used by any traffic class. The derivation of a guard band depends on knowledge of the largest packet size. For a 1Gbps Ethernet connection and a maximum transmission unit (MTU) size of 1500 bytes (1542 bytes on the wire), the guard band is set to: $\frac{1542bytes}{125\times10^6 bytes\times\frac{1}{s}} = 12.3\mu$s.

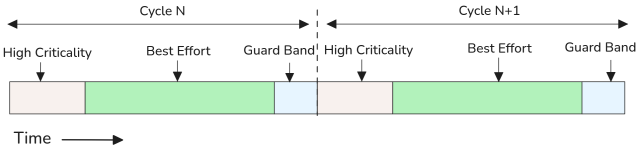

Fig. 1. Time Aware Scheduler Example.

Guard band problems are partially mitigated in one of two ways: (1) using knowledge of the packet length, a TSN scheduler determines its transmission time, and allows the packet to be sent only if it does not interfere with high-priority traffic, or (2) frame preemption is used to halt the transmission of lower-priority traffic when high-priority data is ready to be sent. Both approaches require complex hardware support, which is not yet enabled in Linux.

3 Technical Overview

3.1 Challenges of Non-Unified I/O and Networking

We now focus on the challenges faced by systems using different bus architectures for I/O and networking. Figure 2 shows the problem of moving data between USB (e.g., for device input) and TSN, as it traverses host-level stacks. High-priority outgoing data via a TSN-based i210 Network Interface Card (NIC) misses its TDMA window and waits for best effort traffic, due to lack of synchronization between USB 125μs microframes, user-level processing, and TSN.

Figure 3 presents an example of optimal alignment between USB and TSN subsystems. Two assumptions are made in this case. First, we assume the following delays are time-bounded: (1) the system latency for submitting a USB read request from user-level to the kernel (Δ_{OS_1}), (2) the time to return the data from the device to user-level (Δ_{OS_2}), (3) the user-space processing time (Δ_{user}), and (4) the system delay to forward outgoing data through the networking stack to the TSN-capable network controller (Δ_{OS_3}). As we will see later, this is a challenging assumption to make without support from a finely-tuned RTOS. Second,

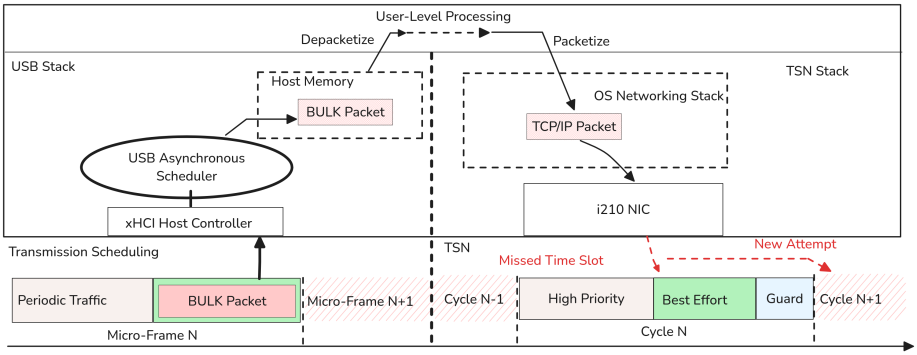

Fig. 2. Complete Datapath of a Dual-bus System.

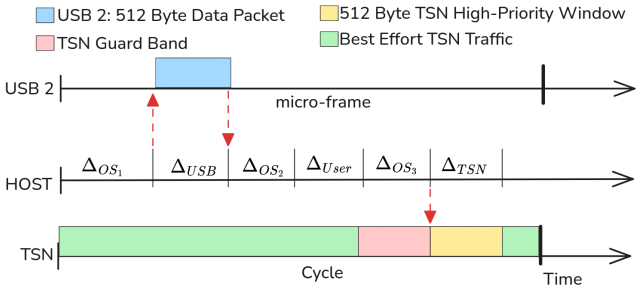

Fig. 3. Optimal Alignment of USB and TSN.

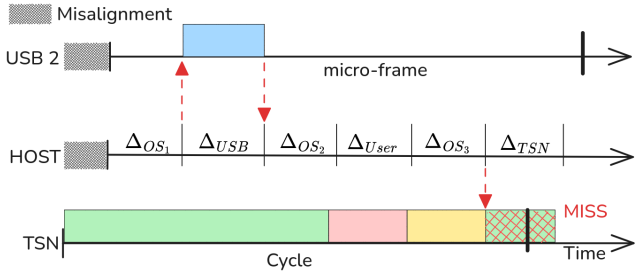

Fig. 4. Clock Misalignment of USB and TSN.

and more improbable, is the assumption that the TSN scheduling cycle aligns perfectly with when the processed USB data is ready for forwarding.

Figure 4 shows how clock misalignment results in high-priority data missing its allocated TSN time slot, causing it to wait for best effort traffic. The increased latency for high-priority data may result in a scheduling avalanche [58] in downstream nodes, or loss of critical information if buffers overflow. Mitigating this problem requires a coordination mechanism that works on top of the I/O and

networking subsystems. Two possible approaches include: (1) introducing more buffering, or (2) imposing more pessimistic constraints on the TSN scheduler.

Increased buffering would allow more I/O packets to be stored in system memory until their scheduled release time on TSN is reached. While this lowers the probability of losing packets, it potentially increases latency, as I/O packets wait for their release. Aside from buffering, the high-criticality traffic time window could be increased in the TSN time-aware scheduler. This, however, may result in noticeable throughput degradation for any low-criticality or best effort traffic class.

Besides the scheduling and coordination challenges faced by dual-bus systems, fundamental design differences between USB and TSN also limit their combined capabilities. One problem is that TSN has fewer traffic priority classes than USB does. Priority inversion is then possible, as I/O packets of different criticality levels may need to share the same priority-class window in TSN.

3.2 Unified USB Stack Solution

Having shown the challenges of a non-unified architecture, we now present a unified networking and device approach using USB. We focus on USB 2 high-speed device I/O as it is still more commonly used than USB 3.x. However, for host-to-host networking, we consider xDBC. xDBC allows for 5 Gbps peak bus bandwidth, which surpasses the commonly used 1Gbps, and even the newer 2.5Gbps, TSN networks.

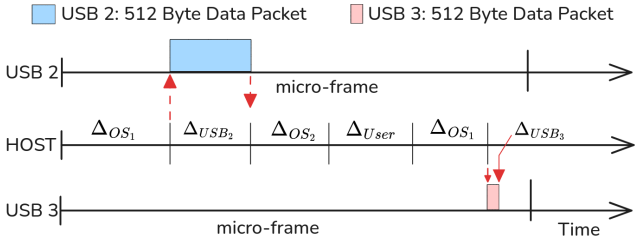

Fig. 5. Unified I/O and Networking Transfer over USB.

Figure 5 depicts an example where high-priority sensor traffic is read from an I/O device, processed in a user-level context, and sent over an xDBC network connection. In this example, both I/O reads and xDBC writes use a common USB software stack to submit requests. Using knowledge of active device transfers and their importance allows the USB stack to reserve bandwidth for when time-sensitive (high-criticality) data needs to be forwarded to a remote host. Thus, the stack is able to reserve time on the xDBC bus precisely after information is processed in user-space and needs transmitting. This is possible by having the USB stack use information about the time to process user-level data, Δ_{User}, for a given traffic priority class. Our stack is made aware of this information,

and, combined with awareness of active USB request blocks (URBs), it avoids assigning best effort or low-priority traffic to the xDBC bus when it predicts it is needed by high-priority data. Some prior works have shown that it is possible to implement such real time scheduling on USB [41,69].

We have implemented host-level networking support using xDBC in both Linux and the Quest RTOS. A Debug Target machine uses an xDBC driver to expose itself as a USB device, as shown in Figure 6. A Debug Host machine enumerates the target via a USB debug device driver. The xDBC capability allows the target to expose two Bulk endpoints for both IN and OUT communication. An example data path using the IN communication endpoint starts with the user-level call to *usb_write*() on Host 1. This results in a system call, which is re-directed to the xDBC driver. This immediately invokes the *xdbc_write*() function, which queues a TX event and notifies the USB host controller once the data is ready to be sent. The xDBC driver also polls the host controller regularly for xDBC-related events that need to be handled, such as completion of sending a packet by the host controller. On the receiving side, the USB debug device driver is responsible for submitting the read requests. Once those requests have been submitted, the debug device driver proceeds to wait until the Host Controller invokes an IRQ handler, which ends up transferring the data to user-level. Information flow from Host 2 to Host 1 is possible by swapping the *usb_read/write*() operations, and using the OUT communication endpoint.

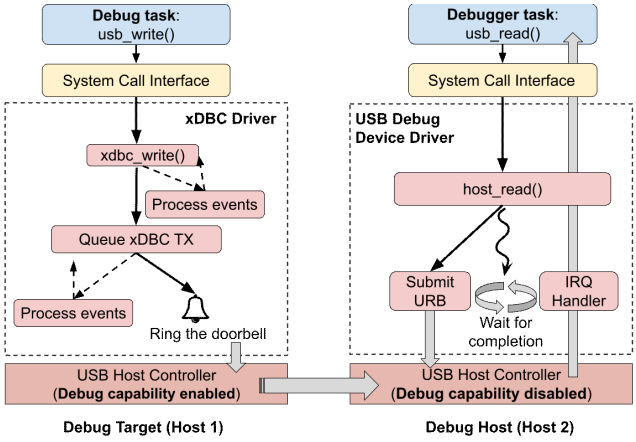

Fig. 6. Example USB xDBC Datapath.

3.3 Latency Model

This subsection presents a latency model for both unified and dual-bus solutions. Our analysis considers a system with a sensory device and two hosts, where a user application on one host (Host 1) reads from the sensor over USB 2, and

forwards data to a second host (Host 2) using either USB 3 or TSN. The end-to-end latency, Δ_{e2e}, is defined as the time between the initiation of a sensor read request on Host 1 and the reception of data on Host 2.

In our analysis, we make the following assumptions: (1) whenever a request is made to the sensor, data is immediately available for transfer between the USB sensor device and the host, (2) a user-level task predictably transfers one data packet from the I/O device onto the network every period, (3) OS-level latencies are bounded, and (4) a software time-aware TSN traffic shaper has a 125μs scheduling cycle and a window for high-criticality traffic of size b_i bytes.

From Figures 3 and 4, we define the following terms:

- Δ_{OS_1}: Represents the worst-case system overhead to issue a USB read from user-space to the USB host controller.
- Δ_{USB2}: Represents the worst-case time for the host controller to transfer a USB 2 packet.
- Δ_{OS_2}: Represents the worst-case system overhead for a USB packet to be returned to user-space from the USB host controller once the data is available.
- Δ_{User}: Represents the worst-case time that the user program takes to process USB data.
- Δ_{OS_3}: Represents the worst-case time to submit a packet from user-space to the network controller.
- Δ_{TSN}: Represents the worst-case TSN packet transmission time.
- Δ_{USB3}: Represents the worst-case USB 3 packet transmission time using xDBC.

We categorize Δ_{e2e} into host and communication delays for ease of analysis. For the host delay, we consider $\Delta_{USB_{RX}} = \Delta_{OS1} + \Delta_{OS2}$ to be the total system time to complete a USB read request. This is followed by Δ_{User} as defined above. Then, $\Delta_{USB_{TX}} = \Delta_{OS_3}$ represents the system latency to send a USB 2 packet onto xDBC, whereas $\Delta_{TSN_{TX}} = \Delta_{OS_3}$ is the latency to send an equivalent Ethernet packet to the TSN NIC. We extend Figures 3 and 4 with the overhead of receiving data on Host 2. This is represented by $\Delta_{USB_{RX}}$ when using xDBC, and $\Delta_{TSN_{RX}}$ when using TSN.

For the total communication delay, we define Δ_{BusArb} to be the bus arbitration delay. This represents the time from when the data arrives into the device queue until it is transmitted, which we consider negligible. $\Delta_{Transfer}$ is the worst-case transfer latency for b_i bytes of data over the respective medium, which corresponds to Δ_{USB2}, Δ_{USB3} and Δ_{TSN}. We define $\Delta_{Transfer\_USB2}$, $\Delta_{Transfer\_USB3}$ and $\Delta_{Transfer\_TSN}$ to differentiate transfers across the respective buses. Finally, $\Delta_{Propagation}$ is the time needed to propagate signals across a bus, which we assume is negligible.

Therefore:

$$\Delta_{e2e} \leq \begin{cases} \Delta_{USB_{RX}} + \Delta_{Transfer\_USB2} & \text{if using} \\ +\Delta_{User} + \Delta_{USB_{TX}} & \text{USB} \\ +\Delta_{Transfer\_USB3} + \Delta_{USB_{RX}} & \\ & \\ \Delta_{USB_{RX}} + \Delta_{Transfer\_USB2} & \text{if using} \\ +\Delta_{User} + \Delta_{TSN_{TX}} & \text{TSN} \\ +\Delta_{Transfer\_TSN} + \Delta_{TSN_{RX}} & \end{cases} \tag{1}$$

$\Delta_{Transfer}$ depends on the specific bus protocol, but is represented as follows:

$$\Delta_{Transfer} = h + 8\alpha p + (\alpha \times \lfloor O_{bits} + 8\gamma b_i \rfloor) \tag{2}$$

In the equation, h denotes the time required for the host to prepare a transmission request. In this paper, we consider peak bandwidths of 480 Mbps for high-speed USB 2, 5 Gbps for superspeed USB 3, and 1 Gbps for TSN. For the USB host controller, h is typically 5 nanoseconds, and no more than 304 nanoseconds for a 1Gb i210 TSN-enabled NIC [30]. p represents the protocol overhead. This includes the Ethernet address and VLAN tags, among others, for TSN, and synchronization transfers (SYNC) and CRC checks in the case of USB. For USB 2 high-speed bulk transfers this value is 55 bytes, while for the USB 3 it is 48 bytes. Protocol overheads for TSN come to 42 bytes per transfer. α is the latency of transferring one bit of data over the corresponding bus. In the case of high-speed USB 2, α is 2.08 nanoseconds, while for superspeed USB 3 and for 1 Gbps TSN, it is 0.2 and 1 nanoseconds respectively. O_{bits} accounts for worst-case bit stuffing and we consider it only used in USB 2 transfers, where it has a fixed value of $19/6 \approx 3.17$. The final variable, γ, is also a USB 2 exclusive overhead and it is the cost of host-device bit-level synchronization on the packet payload. Its value is $7/6 \approx 1.17$ for USB 2. We assume γ to be 1 for TSN and USB 3 since they do not rely on bit stuffing for their synchronization.

The transfer delay formula in nanoseconds for USB 2 high-speed data is:

$$\begin{aligned} \Delta_{Transfer\_USB2} &= h + 8\alpha p + (\alpha \times \lfloor O_{bits} + 8\gamma b_i \rfloor) \\ &= 5 + 8{\times}2.08{\times}55 \\ &\quad + (2.08{\times} \lfloor 3.17 + 8{\times}1.17{\times}b_i \rfloor) \\ &= 921.7 + (2.08{\times} \lfloor 3.17 + 9.36{\times}b_i \rfloor) \end{aligned} \tag{3}$$

The transfer delay formula in nanoseconds for USB 3 superspeed data is:

$$\begin{aligned} \Delta_{Transfer\_USB3} &= h + 8\alpha p + (\alpha \times \lfloor O_{bits} + 8\gamma b_i \rfloor) \\ &= 5 + 8{\times}0.2{\times}48 \\ &\quad + (0.2{\times} \lfloor 0 + 8{\times}1{\times}b_i \rfloor) \\ &= 81.8 + (0.2{\times} \lfloor 8{\times}b_i \rfloor) \end{aligned} \tag{4}$$

Finally, the transfer delay formula in nanoseconds for 1 Gbps TSN is:

$$\begin{aligned} \Delta_{Transfer\_TSN} &= h + 8\alpha p + (\alpha \times \lfloor O_{bits} + 8\gamma b_i \rfloor) \\ &= 304 + 8{\times}1{\times}42 \\ &+ (1 \times \lfloor 0 + 8{\times}1{\times}b_i \rfloor) \\ &= 640 + \lfloor 8{\times}b_i \rfloor \end{aligned} \tag{5}$$

4 Evaluation

This section presents our analysis of two systems, one based exclusively on USB and another involving TSN, with the goal of comparing their networking characteristics. We conduct two set of experiments, one with two connected nodes and another with four nodes connected in a ring topology. Our host nodes are Cincoze DX1100 embedded PCs, each equipped with an Intel Core i7-8700T 2.4GHz CPU. All USB connections to the DX1100 machines are made using USB 3.2 Gen1 ports. These host machines also come equipped with Intel i210 network interface cards (NICs), which provide native TSN support. In addition to the two host machines, we also use a single Teensy 4.1 controller board to mimic the sensor input of a cyber-physical system that sends fixed-size packets. The Teensy 4.1 features an NXP iMX RT1062 System-on-Chip (SoC), with a 600 MHz Arm Cortex M7 processor, and a USB 2.0 high-speed bus. The two host machines in our USB experiments are connected via a crossover cable through an xDBC port of the USB Host Controller for networking. Host-to-host connectivity for the TSN experiments uses a standard 1GbE cable.

In our Linux-based experiments, both hosts use kernel version 6.9.0, patched with PREEMPT_RT [50], version rt5. Comparison experiments are also performed with the Teensy connected to Host 1 running the Quest RTOS [13], which in turn uses xDBC to connect to Host 2, similarly running Quest. We developed a custom CDC-ACM driver for USB2 for both Linux and Quest to ensure similarly predictable read latency from the Teensy to Host 1.

Linux/USB and Quest/USB experiments use USB for both device I/O and networking, as shown in the test setup of Figure 7. Linux/TSN experiments use USB for device I/O and TSN for host-to-host communication, as shown in Figure 8. The USB read and write functions in Linux represent transfers via TTY device file descriptors. Similarly, the `sendto` and `recvfrom` socket functions interact with the Linux networking stack. For Quest, `usb_read` and `usb_write` system calls transfer data between the host and the specific end-point associated with a target bus.

While our experiments represent relatively simple host-to-host communication topologies, future work will focus on more scalable networks such as 3- and 4-dimensional hypercubes. A series of latency experiments read 5,000 512-byte packets from the Teensy, at a rate of 1 every 2.5ms on Host 1. These are processed in user-space and then forwarded to Host 2.

Both USB xDBC and TSN-based experiments in Linux consist of sender and receiver user-level programs, which run on isolated CPU cores. For Quest, user-

level programs on each host run with a real-time Sporadic Server [60] budget of 88μs every 125μs, to match the period of a USB microframe. Time measurements in Linux are performed using clock_gettime, while in Quest using the RDTSC feature of x86 processors. We have confirmed in Linux that both techniques are adequate by comparing them head to head.

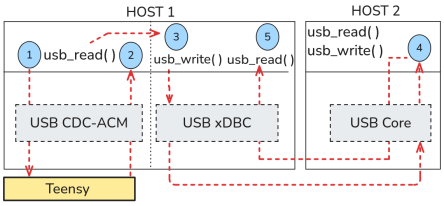

Fig. 7. Unified USB Testing Setup.

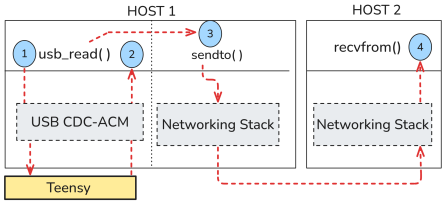

Fig. 8. USB+TSN Dual Bus Testing Setup.

4.1 TSN Test in Linux

A Time-Aware Shaper (TAS) is currently supported in mainline Linux through the Time-Aware Priority Shaper (TAPRIO) queue discipline (qdisc). The TAPRIO qdisc supports the mapping of Linux networking priorities onto the NIC hardware queues, which are additionally supported by a Gate Control List (GCL) scheduling state machine that provides predictable traffic egress. In addition to the qdisc support offered by Linux, we also utilize the Linux gPTP daemon to synchronize the clocks between two hosts.

After successfully synchronizing clocks between the two host machines, the end-to-end latency experiments begin by recording the start-time, t_s, of a USB read request to the Teensy. Once a 512 byte packet is successfully received from the Teensy, it is forwarded to a network socket, with t_s added to its payload. Similarly, the arrival time, t_a, is recorded on Host 2, when the packet is received in user-space. Δ_{e2e} is then measured to be $t_a - t_s$.

Four TSN experiments with varying traffic window sizes assess the variance caused by uncoordinated I/O and networking transfers. These experiments respectively use 8, 16, 62 and 112 microsecond transmission windows for high-priority traffic. The smallest window of 8 microseconds is more than sufficient to transfer 512 bytes at 1 Gbps. Due to space constraints, full results for two of the tests are recorded in this paper (see Figures 9 and 10), while summary statistics for all of the experiments are shown in Figure 13. In all cases we kept the TAPRIO cycle size equal to 125 μs to mimic the USB microframe, and allowed PTP packets throughout all of the 125μs. However, the high priority traffic from the Teensy is only allowed to be transmitted from Host 1 during its exclusive time window.

Between all the transmission window sizes, the 8μs window results in a highest observed latency and variance. As explained in Sections 2 and 3, missing the allocated window slot results in packet buffering, which if done at high frequency causes the backlog in the NIC queues. Additionally, packets could remain queued in the kernel, and if they miss their transmission window on the first host, they

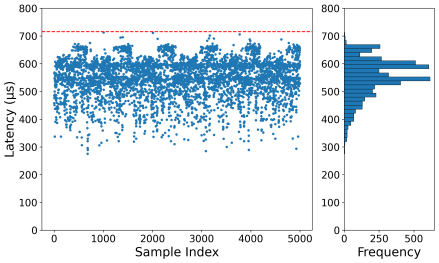

Fig. 9. Linux/TSN End-to-End Latencies with 62μs Window.

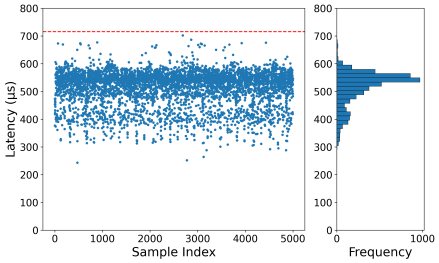

Fig. 10. Linux/TSN End-to-End Latencies with 112μs Window.

may cause a chain of misses down the line. As of yet, extending temporal knowledge of GCL up to the user-space is a challenging process that requires very detailed profiling of the kernel transmission latencies in order to correctly anticipate arrival of the packets to the NIC. With the inclusion of the I/O stream from a different bus, and without any low-level mechanism to synchronize the two, it is not feasible to operate tight control loops across the I/O and networking subsystems needed for real-time control.

We proceed to increase the transmission window size to 16μs, 62μs, and 112μs. The last window size of 112μs represents the largest window possible that still includes a guard band of 13μs. This guard band is based on the largest possible 1500 byte Maximum Transmission Unit (MTU) and 42 byte Ethernet packetization overhead, which requires 12.344μs transmission latency on a 1 Gbps line. As seen in Figure 13, broadening the transmission window results in lower variance by eliminating timeslot misses.

It is worth noting that txtime-assist mode was not used for the TSN experiments, even though it is intended to more precisely ensure the egress of data meets its GCL schedule. Synchronization issues with PTP and TAPRIO are known to exist when using txtime-assist mode, as discussed in [51]. Mechanisms to solve this problem involve using multiple i210 ports, but those were not available with our setup.

We now combine our modeling analysis from Section 3 with our empirical observations made through measurements. We first benchmark the reading of 512 byte packets off the Teensy in order to observe the combined latency of Δ_{OS_1}, Δ_{USB2}, and Δ_{OS_2}. On our Linux system, we observe a maximum Teensy read latency to be: 354.45μs. The Δ_{User} maximum observed latency is 3.32μs. The observed maximum latency for $\Delta_{TSN_{RX}}$ is 199.518μs. When it comes to $\Delta_{TSN_{TX}}$ we use SO_TIMESTAMPING functionality in Linux to measure the OS overhead of network transmission. The maximum outgoing latency is 28.679μs.

It is important to remember that for the worst-case modeling we must also include a potential 125μs penalty a packet might experience if it misses its transmission window. And finally, we refer to Equation 5 to obtain the $\Delta_{Transfer\_TSN}$ for 512 byte packets, which is 4.736μs. Adding all of these values results in a theoretical upper bound of 715.703μs for end-to-end latency over TSN using Linux on Hosts 1 and 2. This is noted in Figures 9 and 10 with a red line.

4.2 USB Test in Linux

To the best of our knowledge no RTOS that we know of has support for both TSN and real-time USB stacks. Since we are interested in doing faithful comparisons of both approaches, we have opted to test single-bus USB designs both in real-time Linux and in the Quest RTOS. As shown in previous works [69,41,22,54] and in Section 2, USB offers a great potential to unify the I/O and networking buses in real-time systems. Even though its USB stack is not explicitly designed with real-time characteristics in mind, PREEMPT_RT-enabled Linux offers an opportunity to compare USB and TSN on the same OS platform, mitigating any major differences that might arise from varying OS implementations. By comparison, the Quest RTOS will then be used to demonstrate the benefits of a unified real-time USB stack for multiple bus instances.

As for TSN, our USB tests in Linux run on isolated CPU cores in order to avoid system disturbances where possible. We have opted to not use the SCHED_FIFO or SCHED_RR policies since our measurements show they result in degraded performance on our test setup. Linux exposes xDBC functionality as a runtime serial communication mechanism. The Linux Debug Host is able to enumerate a Debug Target and access it as a **ttyUSB** device through the TTY layer, while the Debug Target communicates with the host via a **ttyDBC** interface. We disable the TTY-layer buffering and flow control mechanisms to prevent latency variability.

USB is a master-slave protocol that performs synchronization with the host at the hardware level. It does not natively support nor require a PTP-like mechanism for clock synchronization. To ensure USB and TSN measurements are as comparable as possible, we keep the gPTP daemon in Linux to have synchronized clocks across our two hosts. Besides the host-to-host link being USB xDBC, our Linux USB test is similar to that involving TSN. Once again, Host 1 records the initial timestamp, t_s, right before issuing a read request to the Teensy. After Host 1 receives the Teensy data, and verifies its correctness, it forwards it to xDBC. A user-level program on Host 2 reads the ttyUSB interface to receive the data from xDBC. Once the program has obtained the packet, it is inspected to ensure it has the correct signature, and its arrival time t_a is recorded.

Figure 11 shows the obtained results of our experiment. As we can see, the Linux USB measurements fall within our modeling bound and tend to average around 382μs, with a maximum delay of 470μs. These measurements shows that the Linux USB solution provides comparable performance to that achieved with TSN when it has a wide transmission window. However, in the USB case we do not need to statically allocate large portions of a microframe and still achieve similar or better performance.

We conclude the experiments involving Linux transfers over a unified USB network, by comparing to predicted results using the latency model from Section 3.3. We start by reusing the same maximum values for Teensy read requests ($\Delta_{USB_{RX}}$) and Δ_{User}, which are 354.45μs and 3.32μs, respectively. For $\Delta_{USB_{TX}}$, we measure the difference between when the user-level process issues a `usb_write()` and the lowest point in the USB xDBC stack on Linux before the

packet is split into Transfer Request Blocks (TRBs). The maximum of this value is measured to be 64.662μs. On the $\Delta_{USB_{RX}}$ datapath, the largest measurement is 78.664μs. Referring to Equation 4, we calculate the amount of time needed to transfer the 512 byte packet over xDBC. This is 0.901μs for superspeed USB. The upper bound for Δ_{e2e} is 501.997μs, as shown in Figure 11 once again by the horizontal red line.

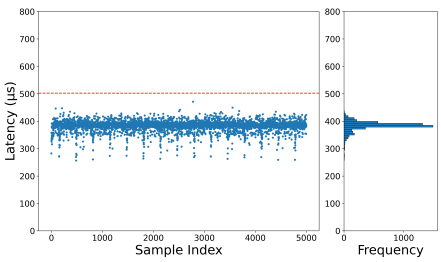

Fig. 11. Linux/USB End-to-End Latencies.

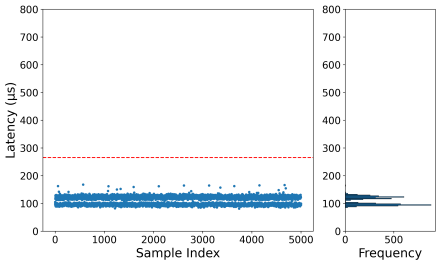

Fig. 12. Quest/USB End-to-End Latencies.

4.3 USB Test in Quest

For the purposes of analyzing USB in the context of cyber-physical systems with hard real-time constraints, we turn to the Quest RTOS. Throughout our testing process we set the budget to 88μs and period to 125μs respectively. The reasoning behind the period value is that we are interested in operating at the timing granularity of 125μs USB microframes. The budget value represents 70% of 125μs period, which gives the host enough time to perform meaningful processing on the packet and perform coordination between the device I/O and networking stacks. The USB test in Quest is similar to that in Linux. It consists of reading a packet, adding necessary signatures (or timestamps), and forwarding the data along to the egress stream.

As mention earlier, USB does not explicitly require clock synchronization mechanisms such as PTP, since the synchronization is embedded within the protocol hardware layer. This, along with the fact that Quest does not yet support gPTP functionality, requires the measurement of packet round-trip latencies, to collect all the timing data within a single clock domain. Our measurements in Quest use the RDTSC mechanism of x86 processors to measure time.

We again derive a latency bound for the USB-based Quest solution by obtaining the maximum latency measurements for varying subsections of our datapath. Starting with the Teensy read latency in Quest, we observe the maximum value of 38.03μs. Δ_{User} has a maximum value of 0.0225μs. Likewise, $\Delta_{USB_{TX}}$ has an upper bound of 55.238μs, while $\Delta_{USB_{RX}}$ is within 46.038μs. Equation 4 yields a worst-case latency for 512 byte packet transfers of 0.901μs. To calculate the worst case latency in Quest we also have to consider the possibility of a missed period, which in our case is 125μs. Thus, $\Delta_{e2e} \leq 265.23\mu$s.

Figure 12 shows the end-to-end latency measurements for Quest. Looking at the numerical statistics in Table 1, and the histograms in Figure 13, it is clear that USB communication in Quest outperforms all other forms of communication involving Linux. USB communication in Quest has a worst-case end-to-end latency that is more than 4 times lower than the best-case involving TSN for Linux. Notwithstanding, unified USB communication in Linux achieves lower worst-case end-to-end latency and delay variation than any TSN scenario.

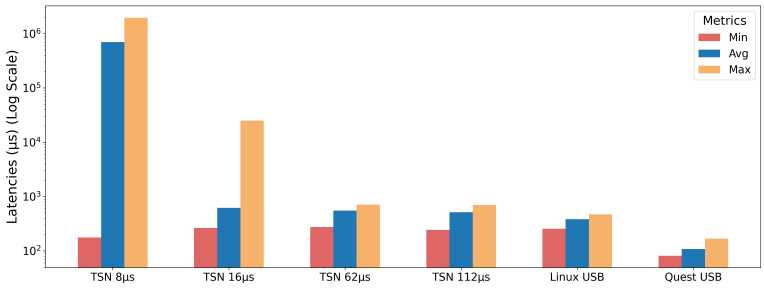

Fig. 13. Combined Statistics for All Experiments.

	Min	Max	Avg	Std
TSN 8μs	176.47	1962558.64	696740.65	301966.31
TSN 16μs	264.97	25105.43	618.75	859.83
TSN 62μs	275.50	712.90	550.21	70.71
TSN 112μs	243.62	701.97	514.75	61.75
Linux USB	256.35	470.93	382.05	19.62
Quest USB	81.55	168.07	108.61	14.57

Table 1. End-to-End Latency Statistics (μs).

4.4 Throughput Measurements

Finally, TSN throughput measurements are performed using iPerf3 on Linux. These measurements are for host-to-host communication, without involving the Teensy. Comparisons with the throughput for USB xDBC on Quest are recorded in Table 2, using different packet payloads from 512 to 2048 bytes. As is seen, USB is able to outperform a 1GbE TSN i210 NIC. This shows that USB xDBC is capable of satisfying high-throughput demands of modern cyber-physical systems, potentially better than TSN. Future work will consider optimizations to our USB communication stack, to take advantage of the higher bandwidths of USB 3.2 Gen2 (20Gbps), and newer buses.

	512 Bytes	1024 Bytes	2048 Bytes
USB:	1014.153	1010.859	1006.255
TSN:	936	934	934

Table 2. Throughput Measurements (Mbps) for Different Payloads.

Note that USB's peak bandwidth of 5Gbps compared to TSN's 1Gbps (in these experiments) does not significantly influence the latency benefits of a single-bus approach to networking and device I/O. Even accounting for the difference in protocol bits added to equal payloads of USB versus TSN packets, the transmission delay difference across each bus is less than a few microseconds. As USB and TSN measured throughputs are actually much closer, as seen in Table 2, the practical transmission delay differences of the two buses have even less of an impact on the end-to-end latencies shown in Figure 13.

4.5 Multi-hop Host-Based Networking

All experiments so far consider point-to-point communication. We now investigate USB networking performance using multiple hosts. We limit the network size to four hosts, arranged in a ring as shown in Figure 14. This might represent a zonal architecture used in an automotive software-defined vehicle, for example, where host-based processing is performed close to the sensors and actuators. Such configurations minimize communication latencies essential for real-time control, whereas larger network deployments introduce excessive delays that compromise system responsiveness.

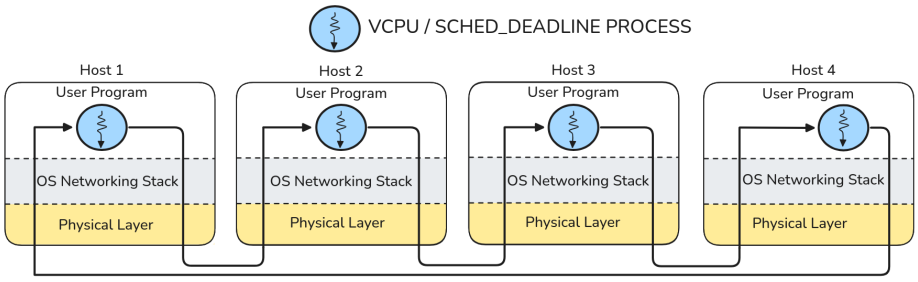

Fig. 14. Experimental four-node ring using DX1100 hosts.

We evaluate the following three scenarios: (1) **Ethernet-based Ring (Linux PREEMPT_RT)**, (2) **USB-based Ring (Linux PREEMPT_RT)**, and (3) **USB-based Ring (Quest RTOS)**. Case (1) considers standard Ethernet without TSN features enabled, as we focus on a single traffic class. This configuration represents optimal TSN performance where all gates remain open, eliminating scheduling constraints and guard band overhead. Case (2) replaces Ethernet links with USB connections using xDBC. Finally, Case (3) replaces PREEMPT_RT-enabled Linux with Quest.

For each of the three scenarios, above, we measure the throughput of 1024-byte packets, representative of sensor data, around the ring. A second experiment measures the round-trip latency of 1024-byte packets, transferred every 500ms from the first host, around the entire ring. The 500ms delay between each transfer represents a sampling interval to retrieve data from a sensor, but this has no bearing on the actual round-trip latency once a packet begins propagation.

Data traversing each host is forwarded by a user-level process. Linux configurations use a `SCHED_DEADLINE` policy, while Quest processes are associated with equivalent VCPUs [13], having 90μs runtime budget and 150μs deadline parameters. These parameters are sufficient to maximize user-level data-forwarding rates without hogging CPU resources.

Throughput results, shown in Table 3, demonstrate that Quest with USB networking achieves superior performance (221.00 Mbps), exceeding the Ethernet baseline by 13.3%. Linux with USB networking exhibits reduced performance (112.71 Mbps) due to TTY subsystem constraints. Originally designed for terminal I/O, the TTY layer introduces substantial overhead through line discipline processing, kernel workqueue scheduling, and context switching mechanisms unsuited for high-frequency data exchange.

Table 3. Average Throughput in 4-Node Ring Topology

Configuration	**Throughput (Mbps)**
Ethernet + Linux PREEMPT_RT	195.08
USB + Linux PREEMPT_RT	112.71
USB + Quest RTOS	221.00

Latency experiment results, shown in Table 4, reveal Quest with USB achieves the best average and maximum performance (1.091 ms, and 1.100ms, respectively), while Linux with USB demonstrates lower delay variation (0.004 ms standard deviation). Ethernet exhibits elevated latency and variability, reflecting the overhead inherent in conventional networking stack architectures.

Table 4. Round-Trip Latency in 4-Node Ring Topology

Configuration	**Min (ms)**	**Max (ms)**	**Avg (ms)**	**Std Dev (ms)**
Ethernet + Linux PREEMPT_RT	1.429	1.800	1.614	0.263
USB + Linux PREEMPT_RT	0.884	1.217	1.214	0.004
USB + Quest RTOS	0.999	1.100	1.091	0.025

The experiments in this section point to the viability of USB as a technology for host-based networking, where processing nodes are within a few hops of sensors and actuators. Results show competitiveness with Ethernet-based networks. For many cyber-physical systems, it seems that USB is a potential candidate for networking and device I/O. Where hosts must naturally perform data processing tasks, the avoidance of using switches is an added appeal of USB, as it keeps the necessary hardware, wiring and cost of networking to a minimum.

4.6 Discussion

The results presented above serve to show the challenges faced by host-level bridging. A host-level bridge links multiple bus segments so that data is processed and forwarded between compute nodes. While TSN provides support for real-time host-level networking, it does not solve the end-to-end problem, as it does not provide a means to communicate directly with input and output

devices. This paper serves to encourage the community to either extend TSN with more widespread support for I/O devices, or to consider supporting a USB standard for host-level networking. USB 4.0 already has support for host-to-host communication, but it has yet to be considered for industrial-grade real-time applications and networks. For small CPS networks, USB offers a lot of potential (e.g., in automotive domains where zonal architectures [39] are becoming a topic of interest, and researchers are attempting to build software defined vehicles and networks [24,71,47]).

The current limitations for USB concern the signaling strength, which limits bus segments to a few meters [67,68]. This distance can be increased using repeaters, powered hub-devices that allow up to 7 bus segments to be chained together, or USB powered cables that reach greater than 15 meters. However, a new USB standard is needed for links that span larger distances. An additional limitation of USB is that it is not a full peer-networking solution. As it operates in a master-slave configuration, it limits topologies to stars and trees [69] . Ring, hypercubic and arbitrary graph topologies require additional hardware [21] (e.g., active bridge cables [29], custom USB switches, or combinations of host and device controllers). Notwithstanding, these challenges are possibly easier to address than implementing a full end-to-end solution for TSN, which requires all devices to have a suitable medium access controller [17].

Now that many Intel processors feature embedded DWC3 [62,63] USB (xDCI) device controllers as a complement to xHCI, it is possible to build relatively complex USB networking topologies. This is possible by configuring each host to have one xDBC connection, one xDCI (device controller) connection, and up to 127 host controller connections to remote hosts configured as devices. The USB specification limits devices to 127 per host controller.

A 4-dimensional hypercubic topology is possible using up to 2 *device* connections comprising either xDBC or xDCI, while all other *host* connections employ xHCI per compute node. **Proof:** Let the device connections be designated *incoming* and the host connections be *outgoing*, although both types may actually be bi-directional using USB IN/OUT endpoints. We know a hypercube has 2 nodes in any dimension, and for an n-dimension hypercube, we have 2^n nodes. The total number of edges is $n{\times}2^{(n-1)}$. This is because each of the 2^n nodes has n edges (one per dimension), but they are each shared with one neighbor. As all edges are effectively directed (either being associated with xHCI, xDBC or xDCI), then there must be a total of $n{\times}2^{(n-1)}$ incoming edges. Any edge that is incoming to a node must also be outgoing from another node, so the total outgoing edges equals the total incoming edges, which equals the total edges. If any node has at most 2 incoming edges (for xDBC and xDCI), and there are 2^n nodes, then the total incoming edges in the entire graph is $2{\times}2^n$, but this must equal the total edges. Hence, $2{\times}2^n = n{\times}2^{(n-1)}$, implying $2{\times}2{\times}2^{(n-1)} = n{\times}2^{(n-1)}$. Thus, $n = 4$, for a max-sized hypercube of $2^4 = 16$ nodes.

While USB is limited to timing at 125μs (the size of a microframe), this serves well for many latency-sensitive applications. Researchers have already implemented a USB-CAN gateway controller [59], suggesting that USB is capable

of handling latencies incurred by CAN bus data. We believe that a combination of USB and TSN network solutions will be useful in the future, allowing for small-scale host-level networks to employ USB, while more complex topologies use TSN. Such a heterogenous approach would make for an interesting solution to traffic routing, and isolation of data of different criticality levels.

5 Related Work

Related work on timing-predictable cyber-physical systems [7] studies the construction of task pipelines to ensure end-to-end guarantees between the arrival of sensing data and the corresponding control output [22,23]. Systems such as Scout [44] schedule paths through as sequence of services that are treated as schedulable entities, but this differs from our focus on investigating the use of USB as a single-bus solution for networking and device I/O. Similarly, RAD-FLOWS [48] investigates a design framework for predictable data communication from a theoretical perspective, independent of technologies such as USB and TSN discussed in this work.

Other work has more generally addresses real-time bus communication in both theoretical scenarios [37,33,5] and physical implementations [15,72,35,38,65]. Tindell et al [65] investigate the worst case transmission delay due to blocking of higher priority tasks, taking jitter into account in their analysis. Davis et al [14] provide a revised study of CAN, correcting for earlier analytical flaws. Likewise, Zuberi and Shin [72] address the utilization problem of CAN-bus networks, using a mixed traffic scheduler that combines both Earliest Deadline First and Deadline Monotonic Scheduling. Others provide a scheduling analysis of CAN [15], assuming messages are scheduled using a FIFO rather than priority queue. In the realm of USB, Huang et al [27,28] attempt to provide QoS guarantees for USB 1.1 and 2.0, by modifying the endpoint descriptors within the host controller driver. In contrast, our work focuses on the use of USB as a combined bus solution for networking and device I/O, showing the performance capabilities in the presence of host-to-host communication based on xHCI's debug capability.

Work by Ruan et al [54] considers the use of USB interrupters to differentiate between I/O transactions for tasks of different priorities and timing requirements. Similarly, work on USB bus scheduling provides guarantees to bulk data transfers in the presence of isochronous and periodic transactions [69,41,22]. Common to these works is the use of software-based scheduling techniques to control the submission of USB transactions to the host controller, thereby shaping the means by which the hardware orders transfer requests.

We compare with TSN [19,57,46], since it stands to be one of the most significant networking technologies in cyber-physical systems. Others describe studies on TSN testbeds [52,8,9], to simulate real-world systems, and application usage in industrial automation [56,66], automotive [64,70,45,6], and robotics [25] research. More specific research aspects of TSN, include work on time synchronization [26,32], credit-based shaping [42,18], and scheduling [61,12].

6 Conclusions and Future Work

This paper investigates the bridging of buses to exchange data between devices and hosts. Bridging is necessary in cyber-physical systems, with task pipelines connecting sensors and actuators to a network of hosts. Empirical results show that using USB for both networking and device I/O is a viable solution for collecting, processing and distributing high bandwidth, low-latency data.

Linux experiments using USB host-to-host communication, in conjunction with USB CDC-ACM device transfers, achieve average and worst-case end-to-end latencies that are better than when TSN is involved in data transfer. This is true even for large TSN transmission windows, suggesting that USB multi-hop communication in Linux is a promising approach. In comparison, combined USB networking and device I/O in Quest achieves a worst-case end-to-end latency that is more than 4 times lower than the best case involving TSN, with 512-byte payload transfers. Additionally, our observations of the throughput with USB suggest it has the potential to handle data rates on par with, or higher than TSN, at least over short range, small-scale host-based networks.

Future work will investigate the use of USB xDCI (eXtensible Device Controller Interface), to form more complex USB or hybrid USB+TSN networks [69]. We are similarly interested in studying differentiated service support [54] to provide end-to-end quality-of-service guarantees to separate communication streams.

7 Acknowledgments

This work is supported in part by the National Science Foundation (NSF) under Grant #2007707. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

References

1. IEEE Standard for Local and Metropolitan Area Networks – Bridges and Bridged Networks - Amendment 25: Enhancements for Scheduled Traffic. IEEE Std 802.1Qbv-2015 (Amendment to IEEE Std 802.1Q-2014 as amended by IEEE Std 802.1Qca-2015, IEEE Std 802.1Qcd-2015, and IEEE Std 802.1Q-2014/Cor 1-2015) pp. 1–57 (2016). https://doi.org/10.1109/IEEESTD.2016.8613095
2. IEEE Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks. IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014) pp. 1–1993 (2018). https://doi.org/10.1109/IEEESTD.2018.8403927
3. IEEE Standard for Local and Metropolitan Area Networks–Timing and Synchronization for Time-Sensitive Applications. IEEE Std 802.1AS-2020 (Revision of IEEE Std 802.1AS-2011) pp. 1–421 (2020). https://doi.org/10.1109/IEEESTD.2020.9121845
4. Alves, M., Tovar, E., Vasques, F.: Ethernet Goes Real-Time: a Survey on Research and Technological Developments (2000), https://api.semanticscholar.org/CorpusID:54705346

5. Aras, C., Kurose, J., Reeves, D., Schulzrinne, H.: Real-Time Communication in Packet-Switched Networks. Proceedings of the IEEE **82**(1), 122–139 (1994). https://doi.org/10.1109/5.259431
6. Ashjaei, M., Lo Bello, L., Daneshtalab, M., Patti, G., Saponara, S., Mubeen, S.: Time-Sensitive Networking in Automotive Embedded Systems: State of the Art and Research Opportunities. Journal of Systems Architecture **117**, 102137 (2021). https://doi.org/https://doi.org/10.1016/j.sysarc.2021.102137
7. Axer, P., Ernst, R., Falk, H., Girault, A., Grund, D., Guan, N., Jonsson, B., Marwedel, P., Reineke, J., Rochange, C., Sebastian, M., Hanxleden, R.V., Wilhelm, R., Yi, W.: Building Timing Predictable Embedded Systems. ACM Trans. Embed. Comput. Syst. **13**(4) (mar 2014). https://doi.org/10.1145/2560033, https://doi.org/10.1145/2560033
8. Bosk, M., Rezabek, F., Abel, J., Holzinger, K., Helm, M., Carle, G., Ott, J.: Simulation and Practice: A Hybrid Experimentation Platform for TSN. In: 2023 IFIP Networking Conference (IFIP Networking). pp. 1–9 (2023). https://doi.org/10.23919/IFIPNetworking57963.2023.10186364
9. Bosk, M., Rezabek, F., Holzinger, K., Marino, A.G., Kane, A.A., Fons, F., Ott, J., Carle, G.: Methodology and Infrastructure for TSN-Based Reproducible Network Experiments. IEEE Access **10**, 109203–109239 (2022). https://doi.org/10.1109/ACCESS.2022.3211969
10. Brunner, S., Roder, J., Kucera, M., Waas, T.: Automotive E/E-architecture Enhancements by Usage of Ethernet TSN. In: 2017 13th Workshop on Intelligent Solutions in Embedded Systems (WISES). pp. 9–13 (2017). https://doi.org/10.1109/WISES.2017.7986925
11. Cardoso, J., Derler, P., Eidson, J.C., Lee, E.A.: Network Latency and Packet Delay Variation in Cyber-Physical Systems. In: 2011 IEEE Network Science Workshop. pp. 51–58 (2011). https://doi.org/10.1109/NSW.2011.6004658
12. Craciunas, S.S., Oliver, R.S., Chmelík, M., Steiner, W.: Scheduling Real-Time Communication in IEEE 802.1Qbv Time Sensitive Networks. In: Proceedings of the 24th International Conference on Real-Time Networks and Systems. p. 183–192. RTNS '16, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2997465.2997470, https://doi.org/10.1145/2997465.2997470
13. Danish, M., Li, Y., West, R.: Virtual-CPU Scheduling in the Quest Operating System. In: 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium. pp. 169–179 (2011). https://doi.org/10.1109/RTAS.2011.24
14. Davis, R., Burns, A., Bril, R., Lukkien, J.: Controller Area Network (CAN) Schedulability Analysis: Refuted, Revisited and Revised. Real-Time Systems **35**, 239–272 (02 2007). https://doi.org/10.1007/s11241-007-9012-7
15. Davis, R.I., Kollmann, S., Pollex, V., Slomka, F.: Controller Area Network (CAN) Schedulability Analysis with FIFO Queues. In: Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems. pp. 45–56. ECRTS '11, IEEE Computer Society, Washington, DC, USA (2011). https://doi.org/10.1109/ECRTS.2011.13, http://dx.doi.org/10.1109/ECRTS.2011.13
16. Decotignie, J.D.: Ethernet-Based Real-Time and Industrial Communications. Proceedings of the IEEE **93**(6), 1102–1117 (2005). https://doi.org/10.1109/JPROC.2005.849721
17. Automotive MAC-PHY devices connect low cost MCUs to Ethernet: https://www.embedded.com/automotive-mac-phy-devices-connect-low-cost-mcus-to-ethernet/ (September 2023)

18. Fang, B., Li, Q., Gong, Z., Xiong, H.: Simulative Assessments of Credit-Based Shaping and Asynchronous Traffic Shaping in Time-Sensitive Networking. In: 2020 12th International Conference on Advanced Infocomm Technology (ICAIT). pp. 111–118 (2020). https://doi.org/10.1109/ICAIT51223.2020.9315374
19. Fedullo, T., Morato, A., Tramarin, F., Rovati, L., Vitturi, S.: A Comprehensive Review on Time Sensitive Networks with a Special Focus on Its Applicability to Industrial Smart and Distributed Measurement Systems. Sensors **22**, 1638 (02 2022). https://doi.org/10.3390/s22041638
20. Finn, N.: Introduction to Time-Sensitive Networking. IEEE Communications Standards Magazine **2**(2), 22–28 (2018). https://doi.org/10.1109/MCOMSTD.2018.1700076
21. Frigerio, A., Vermeulen, B., Goossens, K.G.W.: Automotive Architecture Topologies: Analysis for Safety-Critical Autonomous Vehicle Applications. IEEE Access **9**, 62837–62846 (2021). https://doi.org/10.1109/ACCESS.2021.3074813
22. Golchin, A., Cheng, Z., West, R.: Tuned Pipes: End-to-End Throughput and Delay Guarantees for USB Devices. In: 2018 IEEE Real-Time Systems Symposium (RTSS). pp. 196–207 (2018). https://doi.org/10.1109/RTSS.2018.00037
23. Golchin, A., Sinha, S., West, R.: Boomerang: Real-Time I/O Meets Legacy Systems. In: 2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 390–402 (2020). https://doi.org/10.1109/RTAS48715.2020.00013
24. Gopu, G.L., Kavitha, K.V., Joy, J.: Service Oriented Architecture-based Connectivity of Automotive ECUs. In: 2016 International Conference on Circuit, Power and Computing Technologies (ICCPCT). pp. 1–4 (2016). https://doi.org/10.1109/ICCPCT.2016.7530358
25. Gutiérrez, C.S.V., Juan, L.U.S., Ugarte, I.Z., Vilches, V.M.: Time-Sensitive Networking for Robotics. CoRR **abs/1804.07643** (2018), http://arxiv.org/abs/1804.07643
26. Hu, L., Shou, G., Zhang, X., Liu, Y., Hu, Y.: Multi-domain Time Synchronization Model and Performance Evaluation in TSN. In: 2021 7th International Conference on Computer and Communications (ICCC). pp. 2028–2032 (2021). https://doi.org/10.1109/ICCC54389.2021.9674709
27. Huang, C.Y., Chang, L.P., Kuo, T.W.: A Cyclic-Executive-based QoS Guarantee over USB. In: The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings. pp. 88–95 (2003). https://doi.org/10.1109/RTTAS.2003.1203040
28. Huang, C.Y., Kuo, T.W., Pang, A.C.: QoS Support for USB 2.0 Periodic and Sporadic Device Requests. In: Proceedings of the 25th IEEE International Real-Time Systems Symposium. pp. 395–404. RTSS '04, IEEE Computer Society, Washington, DC, USA (2004). https://doi.org/10.1109/REAL.2004.45, http://dx.doi.org/10.1109/REAL.2004.45
29. Inc., P.T.: SuperSpeed USB 3.0 Host-to-Host Bridge Controller Datasheet (2019)
30. Intel Corporation: Intel® Ethernet Controller I210 Datasheet (2021), https://www.intel.com/content/www/us/en/embedded/products/networking/i210-ethernet-controller-family.html, revision 3.7
31. Road Vehicles – Controller Area Network (CAN) (2009), iSO 11898
32. Kerö, N., Puhm, A., Kernen, T., Mroczkowski, A.: Performance and Reliability Aspects of Clock Synchronization Techniques for Industrial Automation. Proceedings of the IEEE **107**(6), 1011–1026 (2019). https://doi.org/10.1109/JPROC.2019.2915972

33. Kettler, K., Lehoczky, J., Strosnider, J.: Modeling Bus Scheduling Policies for Real-Time Systems. In: Proceedings 16th IEEE Real-Time Systems Symposium. pp. 242–253 (1995). https://doi.org/10.1109/REAL.1995.495214
34. Kopetz, H., Ademaj, A., Grillinger, P., Steinhammer, K.: The Time-Triggered Ethernet (TTE) Design. In: Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05). pp. 22–33 (2005). https://doi.org/10.1109/ISORC.2005.56
35. Kopetz, H., Grünsteidl, G.: TTP-A Protocol for Fault-Tolerant Real-Time Systems. Computer **27**(1), 14–23 (Jan 1994). https://doi.org/10.1109/2.248873
36. Lee, E.A.: Cyber Physical Systems: Design Challenges. In: 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC). pp. 363–369 (2008). https://doi.org/10.1109/ISORC.2008.25
37. Lehoczky, J.P., Sha, L.: Performance of Real-Time Bus Scheduling Algorithms. In: Proceedings of the 1986 ACM SIGMETRICS joint international conference on Computer performance modelling, measurement and evaluation. pp. 44–53. SIGMETRICS '86/PERFORMANCE '86, ACM, New York, NY, USA (1986). https://doi.org/10.1145/317499.317538
38. Loeser, J., Härtig, H.: Low-Latency Hard Real-Time Communication over Switched Ethernet. In: Proceedings of the 16th Euromicro Conference on Real-Time Systems. pp. 13–22. ECRTS '04, IEEE Computer Society, Washington, DC, USA (2004). https://doi.org/10.1109/ECRTS.2004.16
39. Maier, J., Reuss, H.C.: Design of Zonal E/E Architectures in Vehicles Using a Coupled Approach of k-Means Clustering and Dijkstra's Algorithm. Energies (2023), https://api.semanticscholar.org/CorpusID:263285378
40. Messenger, J.L.: Time-Sensitive Networking: An Introduction. IEEE Communications Standards Magazine **2**(2), 29–33 (2018). https://doi.org/10.1109/MCOMSTD.2018.1700047
41. Missimer, E., Li, Y., West, R.: Real-Time USB Communication in the Quest Operating System. In: 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 11–20 (2013). https://doi.org/10.1109/RTAS.2013.6531075
42. Mohammadpour, E., Stai, E., Le Boudec, J.Y.: Improved Credit Bounds for the Credit-Based Shaper in Time-Sensitive Networking. IEEE Networking Letters **1**(3), 136–139 (2019). https://doi.org/10.1109/LNET.2019.2925176
43. Moreira, P., Serrano, J., Wlostowski, T., Loschmidt, P., Gaderer, G.: White Rabbit: Sub-nanosecond Timing Distribution over Ethernet. In: 2009 International Symposium on Precision Clock Synchronization for Measurement, Control and Communication. pp. 1–5 (2009). https://doi.org/10.1109/ISPCS.2009.5340196
44. Mosberger, D., Peterson, L.L.: Making Paths Explicit in the Scout Operating System. SIGOPS Oper. Syst. Rev. **30**(SI), 153–167 (Oct 1996). https://doi.org/10.1145/248155.238771
45. Patti, G., Bello, L.L., Leonardi, L.: Deadline-Aware Online Scheduling of TSN Flows for Automotive Applications. IEEE Transactions on Industrial Informatics **19**(4), 5774–5784 (2023). https://doi.org/10.1109/TII.2022.3184069
46. Pei, J., Hu, Y., Tian, L.: A Review on Key Mechanisms of Time-Sensitive Networking. In: 2021 International Conference on Advanced Computing and Endogenous Security. pp. 01–07 (2022). https://doi.org/10.1109/IEEECONF52377.2022.10013335
47. Pelliccione, P., Knauss, E., Heldal, R., Ågren, M., Mallozzi, P., Alminger, A., Borgentun, D.: A Proposal for an Automotive Architecture Framework for Volvo Cars.

In: 2016 Workshop on Automotive Systems/Software Architectures (WASA). pp. 18–21 (2016). https://doi.org/10.1109/WASA.2016.9

48. Pineiro, R., Ioannidou, K., Brandt, S.A., Maltzahn, C.: RAD-FLOWS: Buffering for Predictable Communication. In: 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium. pp. 23–33 (2011). https://doi.org/10.1109/RTAS.2011.11
49. Rajkumar, R.R., Lee, I., Sha, L., Stankovic, J.: Cyber-Physical Systems: The Next Computing Revolution. In: Proceedings of the 47th Design Automation Conference. p. 731–736. DAC '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1837274.1837461, https://doi.org/10.1145/1837274.1837461
50. Reghenzani, F., Massari, G., Fornaciari, W.: The Real-Time Linux Kernel: A Survey on PREEMPT_RT. ACM Comput. Surv. **52**(1) (feb 2019). https://doi.org/10.1145/3297714, https://doi.org/10.1145/3297714
51. Rezabek, F., Bosk, M., Carle, G., Ott, J.: TSN Experiments Using COTS Hardware and Open-Source Solutions: Lessons Learned. In: 2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops). pp. 466–471 (2023). https://doi.org/10.1109/PerComWorkshops56833.2023.10150312
52. Rezabek, F., Bosk, M., Paul, T., Holzinger, K., Gallenmüller, S., Gonzalez, A., Kane, A., Fons, F., Haigang, Z., Carle, G., Ott, J.: EnGINE: Developing a Flexible Research Infrastructure for Reliable and Scalable Intra-Vehicular TSN Networks. In: 2021 17th International Conference on Network and Service Management (CNSM). pp. 530–536 (2021). https://doi.org/10.23919/CNSM52442.2021.9615529
53. Richard West: The Quest Real-Time Operating System, http://www.questos.org/ (November 2024), http://www.questos.org/
54. Ruan, Z., Njavro, A., West, R.: USB Interrupt Differentiated Service for Bandwidth and Delay-Constrained Input/Output. In: 2024 IEEE 30th Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 42–54 (2024). https://doi.org/10.1109/RTAS61025.2024.00012
55. Sanchez-Garrido, J., Aparicio, B., Ramírez, J.G., Rodriguez, R., Melara, M., Cercós, L., Ros, E., Diaz, J.: Implementation of a Time-Sensitive Networking (TSN) Ethernet Bus for Microlaunchers. IEEE Transactions on Aerospace and Electronic Systems **57**(5), 2743–2758 (2021). https://doi.org/10.1109/TAES.2021.3061806
56. Seliem, M., Pesch, D.: Software-Defined Time Sensitive Networks (SD-TSN) for Industrial Automation. In: 2022 14th International Conference on Computational Intelligence and Communication Networks (CICN). pp. 1–7 (2022). https://doi.org/10.1109/CICN56167.2022.10008262
57. Seol, Y., Hyeon, D., Min, J., Kim, M., Paek, J.: Timely Survey of Time-Sensitive Networking: Past and Future Directions. IEEE Access **9**, 142506–142527 (2021). https://doi.org/10.1109/ACCESS.2021.3120769
58. Simon, C., Maliosz, M., Mate, M.: Design Aspects of Low-Latency Services with Time-Sensitive Networking. IEEE Communications Standards Magazine **2**(2), 48–54 (2018). https://doi.org/10.1109/MCOMSTD.2018.1700081
59. Sinha, S., West, R.: Towards an Integrated Vehicle Management System in DriveOS. ACM Trans. Embed. Comput. Syst. **20**(5s) (sep 2021). https://doi.org/10.1145/3477013, https://doi.org/10.1145/3477013
60. Sprunt, B., Sha, L., Lehoczky, J.: Aperiodic Task Scheduling for Hard Real-Time Systems. Real-Time Systems Journal **1**(1), 27–60 (1989)

61. Stüber, T., Osswald, L., Lindner, S., Menth, M.: A Survey of Scheduling in Time-Sensitive Networking (TSN). ArXiv **abs/2211.10954** (2022), https://api.semanticscholar.org/CorpusID:253734910
62. Synopsys: DesignWare USB-C 3.0 Controller IP. https://www.synopsys.com, observed May, 2025
63. Synopsys: Synopsys USB 3.0 Dual-Role Device Controller IP. https://www.synopsys.com, observed May, 2025
64. Testa, G., Canzonieri, G., Leonardi, L., Bello, L.L., Patti, G.: Comparative Assessment of Time-Sensitive Networking Transmission Schemes in Automotive Applications. In: 2023 AEIT International Conference on Electrical and Electronic Technologies for Automotive (AEIT AUTOMOTIVE). pp. 1–6 (2023). https://doi.org/10.23919/AEITAUTOMOTIVE58986.2023.10217195
65. Tindell, Hansson, Wellings: Analysing Real-Time Communications: Controller Area Network (CAN). In: 1994 Proceedings Real-Time Systems Symposium. pp. 259–263 (1994). https://doi.org/10.1109/REAL.1994.342710
66. Tschöke, S., Lynker, F., Buhr, H., Schreiner, F., Willner, A., Vick, A., Chemnitz, M.: Time-Sensitive Networking over Metropolitan Area Networks for Remote Industrial Control. In: 2021 IEEE/ACM 25th International Symposium on Distributed Simulation and Real Time Applications (DS-RT). pp. 1–4 (2021). https://doi.org/10.1109/DS-RT52167.2021.9576141
67. Universal Serial Bus Specification, 2.0 edn. (April 27 2000)
68. Universal Serial Bus Specification, 3.2 edn. (September 22 2017)
69. West, R., Golchin, A., Njavro, A.: Real-Time USB Networking and Device I/O. ACM Trans. Embed. Comput. Syst. **22**(4) (Jul 2023). https://doi.org/10.1145/3604429, https://doi.org/10.1145/3604429
70. Xu, Y., Shang, J., Tang, H.: Recent Trends of In-Vehicle Time Sensitive Networking Technologies, Applications and Challenges. China Communications **20**(11), 30–55 (2023). https://doi.org/10.23919/JCC.ea.2021-0888.202302
71. Zhu, H., Zhou, W., Li, Z., Li, L., Huang, T.: Requirements-Driven Automotive Electrical/Electronic Architecture: A Survey and Prospective Trends. IEEE Access **9**, 100096–100112 (2021). https://doi.org/10.1109/ACCESS.2021.3093077
72. Zuberi, K.M., Shin, K.G.: Non-preemptive Scheduling of Messages on Controller Area Network for Real-Time Control Applications. In: Proceedings of the Real-Time Technology and Applications Symposium. RTAS '95, IEEE Computer Society, Washington, DC, USA (1995), http://dl.acm.org/citation.cfm?id=526671.828330