Journal of Systems Architecture 137 (2023) 102853

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Composable distributed real-time systems with deterministic network channels☆

Henrik Austad [a,*], Erling Rennemo Jellum [b], Sverre Hendseth [b], Geir Mathisen [b], Torleiv Håland Bryne [b], Kristoffer Nyborg Gregertsen [a], Sigurd Mørkved Albrektsen [a], Bjarne Emil Helvik [c]

[a] SINTEF Digital, Mathematics and Cybernetics, Strindv. 4, Trondheim, 7034, Norway
[b] NTNU, Department of Engineering Cybernetics, O.S. Bragstads plass 2D, Trondheim, 7034, Norway
[c] NTNU, Department of Information Security and Communication Technology, O.S. Bragstads plass 2B, Trondheim, 7034, Norway

ARTICLE INFO

Keywords:
Real-time networks
Time-flow-graph
Distributed
CPS
IIoT
TSN
Linux

ABSTRACT

A system that needs to interact with the physical world in a timely manner is called a *real-time system*. When such a system is composed of multiple subsystems, or nodes, each of which is a geographically separate system, such a system of systems is called a *distributed* real-time system. The computation at each node must adhere to the timing requirements, and the connecting communication channels must never cause delays that trigger further timing violations. In this paper, we introduce *Deterministic Network Channels*, a network construct using Time Sensitive Networking QoS mechanisms that add reliable and deterministic communication for distributed tasks. Introducing such network channels as a construct allows designers to focus on higher-level primitives when building distributed systems. We describe our reference implementation and evaluate it by extending Timed C with network channels. Building on this, we also perform a thorough performance evaluation to determine practical bounds for both Linux and TSN under heavy workloads and adverse network conditions to show how the proposed reference implementation performs in real-world scenarios. In our tests, we can synchronize two separate machines running commercial off-the-shelf hardware to within 15 µs of each other under severe internal and external interference.

1. Introduction

In a real-time system, the correctness of a computation is not only dependent on its logical output, but also on the time at which the computation is ready. Meeting the temporal requirements of a system is a challenging task as most hardware architectures are not deterministic to time. Neither are most programming models that leave the programmer to fight temporal demons such as enforcing deadlines and handling timeouts on their own. Both require a deep understanding of both the problem domain and the hardware used for the system and often result in subtle, yet devastating timing bugs. A common technique is to configure timers to interrupt the program at specific points to prune away some of this non-deterministic timing. This increases overall complexity by exposing minute details about hardware capabilities, -configuration, and interrupts. It also shifts focus away from the core application being developed to the system on which it runs.

The C programming language continues to be the most popular language for embedded systems. C is a small language, it has few keywords, a limited set of standard libraries, and, by allowing direct memory reference, exposes the underlying hardware to developers. This is a tremendous expressive power — and danger, causing many to consider C an unsafe language. What is more, as the design of C lends itself naturally to compilers and hardware, writing a *C implementation* (i.e., "compiler") is a relatively easy task, which is another reason why C is normally the first language supported on a new architecture. This makes C a common language for small and embedded systems.

Timed C is an extension of the C programming language developed at the KTH Royal Institute of Technology [1]. It adds timing primitives to the programming language and thus abstracts away the configuring of timers and interrupts with a portable API. Timed C brings time as a

☆ This work was funded by the Norwegian Research Council under grant 323340 via SINTEF, under grant 327538, and by the Centre of Excellence NTNU AMOS via grant 223254.

* Corresponding author.
E-mail addresses: henrik.austad@sintef.no (H. Austad), erling.r.jellum@ntnu.no (E.R. Jellum), sverre.hendseth@ntnu.no (S. Hendseth), geir.mathisen@ntnu.no (G. Mathisen), torleiv.h.bryne@ntnu.no (T.H. Bryne), kristoffer.gregertsen@sintef.no (K.N. Gregertsen), sigurd@albrektsen.net (S.M. Albrektsen), bjarne@ntnu.no (B.E. Helvik).

https://doi.org/10.1016/j.sysarc.2023.102853
Received 20 September 2022; Received in revised form 21 February 2023; Accepted 22 February 2023
Available online 2 March 2023

"first-class member" and facilitates communication between local tasks. Currently, it does not support network transport, preventing Timed C from extending to distributed systems design. One reason for Timed C's "non-distributedness" is largely the network itself. The uncertainty in both delivery (i.e. packages being lost in transit) and delivery jitter (i.e. variance in transmission delay through the network) in ordinary packet-switched networks (PSN) must be properly addressed before such a network can be safely used in real-time systems.

As the world progresses towards "Industry 4.0" with Industrial IoT (IIoT) and Cyber–Physical Systems (CPS), the complexity and size of industrial systems continue to grow and expand. This shift to distributed architectures can effectively mitigate complexity as encapsulating each function in a separate node lends itself well to fault tolerance where critical functions are physically separated. However, with this distribution also comes increased complexity in managing the network connection between the nodes.

In this paper, we extend the Time Flow Graph formalism first described in Timed C with primitives for setting up and using deterministic *network channels*. We then describe a reference implementation and implement this on top of Time Sensitive Networking (TSN) streams and Linux before we show how this can be used to extend the capabilities of Timed C.

This simplifies building scalable and composable distributed real-time systems (DRTS) in C. We then evaluate the real-time performance of such network channels running on a GNU/Linux system.

The rest of this paper is organized as follows: in Section 2 we list the specific contributions made by this work. Section 3 covers the background in real-time systems, TSN, and Timed C. In Section 4 we first extend the Time-Flow Graph formalism with network channels before we present our reference implementation, a federated architecture extension to Timed C. Section 5 describes the experimental setup for our reference implementation. The results of these experiments are shown in Section 6 before we conclude in 7.

2. Contributions

In this work, we add support for deterministic network channels that provide guaranteed delivery, bounded latency, and ease of use. We have extended Timed C, a coordination framework for real-time tasks, such that it can be used to create DRTS which we have called *"Federated Timed C"*. In summary:

- C1: We have extended the Time Flow Graph formalism from Timed C to handle network channels for distributed systems and defined the requirements for composability.
- C2: We have extended Timed C by bringing network channels as a composable construct to tasks, enabling a distributed system to be designed as a Timed C program.
- C3: We have specified and implemented reference code with a reliable, deterministic network transport construct, including bounded latency that allows reasoning about timing constraints in distributed systems.
- C4: We have quantified the network transport latency and Linux's real-time capabilities to further strengthen the confidence in the latency bounds provided by the network channels.

The developed code is available under an open-source license at GitHub [2]with a preconfigured TimedC for also available [3]. Finally, all the tools used for generating interference are also available under open licenses.

3. Background and motivation

Broadly speaking, a Distributed Real-Time System (DRTS) must contain at least 3 components to be able to exhibit what we call "real-time behavior". First, each node in the system must be a capable real-time system. Second, the network connection between the nodes must be both reliable and deterministic in that no data should be lost nor should it be unduly delayed. Last, all nodes must share a common understanding of time.

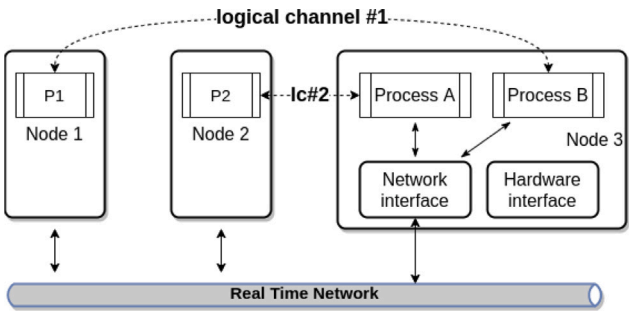


Fig. 1. Distributed real-time system with logical channels connecting nodes on 3 different hosts. In the expanded view of Node 3, we see that it contains two separate processes that both use the same network interface, but where the *logical channels* are distinctly between Process A and P2 and between process B and P1.

3.1. Real-time systems

The temporal requirements of the components of the real-time system are specified through deadlines. Depending on the consequence of missing a deadline, real-time systems are broadly divided into 3 categories: *(a) Soft real-time*, where a missed deadline is not fatal but the value of the result declines rapidly; *(b) Firm real-time*, where the value of a delayed result is worthless, but the system may yet recover; and *(c) Hard real-time* systems where a missed deadline results in a total system failure. Where soft real-time systems need care and attention, hard real-time systems often require a combination of rigid and well-structured development processes, formal verification, thorough testing, and certified toolchains and operating systems.

A real-time system is typically composed of several modules or subsystems where each serves a distinct role. Not all parts may have the same real-time requirements, and requirements may not fall firmly into one category, but rather be "somewhere in between". It is quite common to have sections with soft to no real-time requirements whereas others have more stringent requirements. In such mixed-criticality cases, it is essential to have a clear separation of dependencies and execution. The system may run in multiple threads in the same process, as separate processes located on isolated CPU cores — or something in between. Common for all such cases is the need for a methodical design approach and a robust and deterministic run-time.

3.2. Distributed real-time systems

A DRTS is a "system of systems" where each subsystem itself is a real-time system. The subsystems are interconnected via a real-time network [4]. Fig. 1 shows a construed DRTS. In each node, there are one or more processes that run the node's part of the system. Between the nodes are *logical channels* through which the processes pass control signals, data, or both. Depending on the abstraction level, these channels appear either as any other synchronization (or data sharing) primitive, a raw network socket, or something in between. These logical channels are ultimately handled by the network layer and are thus dependent upon the real-time behavior of the underlying network.

There are several reasons to choose a distributed architecture for a real-time system. First of all, the physical layout of the system may be distributed and thus lend itself naturally to a distributed architecture. Second, we mitigate a major drawback of single-host architecture, namely *decreasing composability*. When a "single-host system" grows, it becomes increasingly more difficult to contain the side effects of

CPU load, available memory, traffic on shared data buses, changes in scheduler timing, and so on. In other words, it is difficult to shield some parts of an application from others on such a system. Attempting to handle this may in turn lead to increased complexity when new components are added, making the quest for subsystem shielding even more difficult.

Extending a distributed architecture can be straightforward as long as shared resources and data dependencies are avoided. It can be as simple as adding a new node to the system. Fault tolerance can also be easier to implement in a distributed architecture because different functions and resources are already cleanly separated. This naturally lends itself to a design where critical functionality can easily be duplicated on multiple nodes.

The challenge lies in extending a DRTS with overlapping functionalities. The moment one node is dependent upon signals or data arriving from another node, the real-time behavior of the other nodes *and* the network becomes relevant. Thus, the delay *between* the nodes directly affects the system behavior.

3.3. Composability and composable DRTS

A system is *composable* to a property if, once the property has been established for a subset of the nodes, the addition or removal of nodes does not affect the property [5]. For DRTS, the timing and network latency are relevant properties. In a non-composable system, the timing behavior of an individual component could change if new components were added to the system. An example of such a system is a multi-threaded program running on a single-core CPU managed by a non-real-time operating system. When each component is in a separate thread, any additions will change the timing. In general, the question of composability arises in systems with resource sharing. In the multi-threaded example CPU, memory and storage are other examples of shared resources. For a DRTS, the network must be managed as another shared resource.

A DRTS can only achieve composability over a shared medium if it is possible to properly shield critical network traffic from other, unrelated traffic. For some systems, the sensitivity to disturbance can be so low that *arrival guarantees* are adequate. For other applications, such as industrial protocols and control loops, higher demands are placed on the delivery latency provided by the network. A common technique for reducing the variance in the delay to a minimum is to use a time-triggered communication system [5] to properly shield critical traffic.

In general, you can differentiate between event-triggered and time-triggered communication protocols. In an event-triggered protocol, the control of the timing of the transactions lies with the individual nodes of the DRTS whereas in a time-triggered protocol, the control resides within the communication system. The available throughput is then often multiplexed in the time domain. Traditional PSNs are event-triggered systems, and must therefore have attributes that allow them to behave in a time-triggered manner.

3.4. Real-time networks and TSN

Unless each node has a dedicated link to every other node, the connecting fabric must provide a way to reduce interference from unrelated traffic. Time Sensitive Networking (TSN) is a set of IEEE standards that are a continuation of Audio/Video Bridging (AVB) [6]. TSN enables time-triggered communication over Ethernet and provides strict Quality-of-Service (QoS) guarantees for time-sensitive or critical traffic. Initially built on top of IEEE 802.3 Ethernet, QoS is governed by IEEE 802.1Q Bridges and Bridged Networks [7] standards. It can also use other packet-switched transport protocols such as 802.11n, MoCA v2, and ITU-T G9960.

TSN uses the concept of Bridges and End Stations to describe network entities that send streams of regular, periodic traffic from one Talker to one or more Listeners. Talkers and Listeners are both End-Stations, Bridges are network entities with 2 or more ports (i.e., switches and routers) that forward traffic through the network. A Talker first announces an available stream before one or more Listeners can subscribe by requesting the necessary capacity through the network. If the network can accommodate the request with available buffers and bandwidth, the reservation succeeds. The Listener then receives data with extremely low packet loss[1] and bounded latency. An administrative upper reservation bound is typically set to 75% of link capacity to ensure that Best Effort (BE) traffic can flow. Any reserved link capacity that is left unused can be freely used by any BE traffic class.

TSN uses shapers to form traffic, and the first shaper introduced was the Credit Based Shaper (CBS, [7, Ch. 35]) that provides a bounded end-to-end (E2E) latency of 2 ms (for "class A") over a maximum of 7 network hops in a 100 Mbps network. Class B streams provide a 50 ms bounded latency guarantee and include the possibility of 2 wireless links in the path. CBS is a class-based shaper and is designed for constant bitrate, periodic traffic. By actively working to reduce traffic bursts on all bridges, total network burstiness is managed. These guarantees can be given due to the admission control provided when reserving resources. The Time Aware Shaper (TAS) [9] provides time-deterministic messaging by employing Time Division Multiple Access (TDMA) through the *gate control lists*. Each item on the list specifies if the gates for the priority queues should be open or closed. By carefully aligning *gateOpen* events for only scheduled traffic across the path, TAS can guarantee a 100 µs bounded latency (over 5 hops in a 1 Gbps network). For more sporadic traffic, TSN has a third shaper, the Asynchronous Traffic Shaper (ATS, [10]). ATS is based on the Urgency Based Scheduler [11], which does not require tight time synchronization of all bridges. ATS is thus more scalable than TAS and can handle sporadic traffic with less reservation overhead than CBS.

This comes as a contrast to other QoS schemes such as Integrated Services (IntServ, [12]) and Differential Services (DiffServ, [13]). The former is often called "Hard QoS" as it is based on reserving capacity for individual streams, and, as with TSN, a reservation may fail if inadequate resources are available. Due to the per-stream reservation, IntServ has scalability problems when the number of streams grows and the network increases in size. DiffServ is a class-based QoS and thus avoids many of the scalability problems IntServ faces, but at the cost of sacrificing QoS guarantees for individual streams. A combination of both was shown by Harju and Kivimaki [14] to provide both good QoS and scalability for large-scale IP-based networks.

This combination of IntServ and DiffServ is one of the approaches taken by IETF Deterministic Networking (DetNet). The goal of DetNet is to "provide a capability to carry specified unicast or multicast data flows for real-time applications with extremely low data loss rates and bounded latency within a network domain" [15]. Unlike TSN, DetNet operates on routable network traffic and does not provide explicit technology recommendations but instead, states desired behavior. A DetNet can be realized over a TSN but is not limited to TSN only. Nor is it required to be homogeneous in the sense that a DetNet can traverse both TSN and Multiprotocol Label Switching (MPLS) networks, and even hops over non-DetNet aware bridges. DetNet targets large Wide Area Networks (WAN) with explicit real-time demands but does not aim to meet as high demands as TSN which can be as low as 100 µs. The exact capabilities depends on the network configuration, what DetNet provides is a standardized way to configure and provision large network and express timing requirements.

[1] TSN guarantees no packet loss due to buffer congestion, but traffic can still be lost if the link itself is disrupted or the bridge becomes unavailable. For this, frame replication [8] can be used to create redundant paths.

3.5. Precision time protocol

For a component in a system to be able to reason about information from any other component, a shared understanding of time is required. Comparing two sensor readings becomes pointless if the capture time is unknown. This is normally solved in a single system by having a local source of time, e.g., a CPU clock. With a DRTS it is no longer possible to directly share a clock in this manner. In Lamport's seminal paper [16] about time and order of events in systems, he lay the foundation and need for time domains and network time protocols in distributed systems.

In 2002, IEEE published the first version of the Precision Time Protocol (PTP). One of the main goals of PTP was to enable an accurate time-keeping architecture for distributed systems operating in environments where access to high-quality time signals (e.g., atomic clock, Global Navigation Satellite System, GNSS) was either impossible or too expensive. PTP operates with the notion of *clocks* and treats both network infrastructure and endpoints as such. Network equipment is either *transparent clocks* or *boundary clocks* and endpoints are typically *ordinary clocks*. During normal operation, the clocks in the network select a single clock to act as the authoritative source of time, the *Grand Master (GM)*. The GM periodically sends updates, and all other clocks then update the offset and phase difference between their local clock and the GM clock. PTP specifies both a software- and a hardware-driven approach. In the software-driven case, the accuracy is greatly affected by how fast the system can react to new PTP messages. Any uncertainties in how tightly the local and remote times are coupled decrease the clock accuracy. Some network cards are capable of intercepting PTP sync messages and associating a local timestamp with incoming messages. This all but eliminates local variations in reacting to the messages and greatly improves the clock accuracy.

In a system where PTP is supported by both network switches and receiving hardware, the error between the GM and any clock is typically much less than 1 µs, and with careful configuration, PTP is capable of even higher accuracy. As an example, Project White Rabbit [17] achieved *sub-nanosecond* accuracy by using FPGAs and Synchronous Ethernet (SyncE) to reduce the remaining jitter throughout the network. In 2008, PTP underwent a large revision PTPv2 is the version supported by most network infrastructures and endpoints today. 2019 saw a new update to what is known as PTPv2.1 [18]. PTPv2.1 is backward compatible with v2 profiles, but v1 is not compatible with v2.

3.6. Linux, PREEMPT_RT and rt-tests

The Linux kernel [19] is a general-purpose operating system (GPOS) kernel, that, when bundled with system libraries, many of which are maintained by GNU [20], creates an open and free POSIX-compliant OS ("GNU/Linux"). PREEMPT_RT [21] started as a series of patches to modify the Linux kernel to create a deterministic real-time operating system (RTOS). It does this by making large portions of the kernel preemptable, most notably system calls (syscalls) and interrupt-handlers. By changing many of the interrupt handlers into kernel threads, they can be scheduled as regular threads. As we will see in Section 6, using PREEMPT_RT can improve the determinism significantly when the system operates under heavy load.

`cyclictest` is part of the rt-test suite [22] and was originally developed to measure the accuracy of the reworked timer infrastructure in the Linux kernel. It has since proved to be a very effective tool for both profiling the real-time performance of systems as well as pinpointing troublesome drivers and applications since it can direct the Linux kernel tracing subsystem [23].

3.7. Timed C

There exist several frameworks, dedicated languages, and extensions that address one or several of the critical elements for DRTS. One such candidate is Timed C and due to its clean design and small size, is imminently suited to extend and experiment with.

Timed C is an extension to the C programming language developed at KTH [1,24,25]. It consists of a set of constructs and primitives that allows a programmer to specify the intended timing of a program directly. It can be seen as a way of making time itself a "first-class citizen" of C. Timed C is meant to be portable and generates code for both POSIX and FreeRTOS.

Timed C uses the concept of *timing points*. Timing points allow the programmer to specify soft, firm, and hard real-time requirements. For example, the program should reach a certain point within a specified deadline. Semantically, timing points create a *logical* timeline for the program. Logical time only progresses at the timing points and program sequences between the timing points take zero logical time. When implemented as a physical system, the program will also exist on a *physical* timeline. A timeline violation occurs if the program arrives at a physical time that is greater than its associated logical time. A timeline violation can also be interpreted as a missed deadline.

The consequence of a missed deadline depends on the type of timing point, which represents the type of real-time guarantee it is associated with. A soft timing point does not change the program flow in case of a missed deadline but instead returns the amount of overshoot so that the application may deal with it. A firm timing point, on the other hand, interrupts the program flow when a deadline is missed and jumps directly to the timing point forcing the currently executing code to be aborted [26, III. B]. A hard timing point will behave like a firm timing point, but instead of jumping to a timing point, it will move to a dedicated error-handling routine. The exact implementation of timing points is outlined in [26]. Currently, hard timing points are not enforced in Timed C on x86 architecture.

Listing 1 shows a contrived example where timing points can be inserted into a system such that a sensor can be read regularly at 50 Hz. If `process()` should exceed the available time, the program will be interrupted and the control flow can be diverted to a rescue routine. In Fig. 2 a timing diagram for initialization and two iterations of the loop can be found. `sdelay()` implements a soft timing point, `fdelay()` a firm.

```
1  int main(void) {
2    init();
3    sdelay(10, ms);
4    while (1) {
5          read_sensor();
6          process();
7          fdelay(20, ms);
8    }
9  }
```

Listing 1: Timed C program using `sdelay()` and `fdelay()` to insert timing points in a system that reads and processes a sensor value. After the initial setup, the main loop repeats every 20 ms (50 Hz).

Timed C also contains the concurrent construct *task* which is a thin wrapper around the system multithreading library (pthreads in the case of Linux). Tasks can communicate through *fifochannels* and *lvchannels*. The former gives buffered non-blocking write and blocking read synchronization while the latter is a shared variable without synchronization. Listing 2 shows a system with 2 parallel tasks communicating over a shared fifochannel. No run-time schedules the different tasks according to the specific timing points, instead Timed C relies on the underlying RTOS for scheduling. Tasks can be assigned to different scheduling policies; the priority among the tasks is determined by static analysis of the timing points found in each task (see lines 3 and 12 in

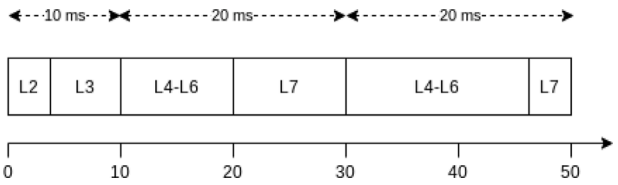


Fig. 2. Timing diagram of code in Listing 1. Note that L3 and L7 are of variable length, if L2 completes after 4 ms, then L3 adapts to spend 6 ms. The loop should run at 50 Hz, hence the 20 ms timing point. In the first iteration, the lines L4–L6 consume 10 ms, and in the second iteration, 16 ms. L7 then adapts to ensure a full 20 ms loop.

Listing 2). Soft timing points are just translated into sleep statements, while firm timing points also start a countdown timer which would interrupt the program when the deadline is missed. Timed C is, as such, not a complete framework for implementing real-time systems. It is, however, a capable coordination framework in which one can implement systems with explicit timing constraints.

```
1  int fifochannel(fifo);
2  task reader() {
3     spolicy(FIFO_RM);
4     int val;
5     while (1) {
6        cread(fifo, val);
7        printf("fifo: %d\n", val);
8     }
9  }
10
11 task writer() {
12    spolicy(EDF);
13    int w = 0;
14    while(1) {
15       cwrite(fifo, w);
16       fdelay(100, ms);
17          w++;
18    }
19 }
20
21 void main(){
22    cinit(fifo, 0);
23    reader();
24    writer();
25 }
26
```

Listing 2: Two periodic tasks with different scheduling policies exchanging data through a shared fifochannel.

A formalism called Time Flow Graphs [26] is a way of representing a Timed C task. Timing points and the code fragments in between timing points are represented as nodes and the dependencies are represented as edges. A methodology for verifying the temporal properties specified in a Timed C program is also proposed. TFG is further discussed in Section 4.1.

3.8. Related work

Ptides [27] is a programming model for distributed systems based on the relationship between model time and real-time. In Ptides, known bounds on network latency and execution times enable distributed synchronization with little communication. Lingua Franca [28] is a coordination framework for distributed real-time systems based on the Reactor model of computation (MoC) [29]. The Reactor model is a timed, discrete-event MoC. It includes a notion of logical time which is related to Timed C but has a more formal definition of time between synchronization points, as well as the simultaneity of events. In Lingua Franca, software components, called reactors, communicate with time-tagged signals through named ports. The time tags are drawn from the logical timeline and logical time does not elapse during computation. Reactors react to events in time tag order and this ensures determinism at each logical instant. Distributed execution is achieved either with centralized coordination or decentralized coordination through Ptides. The distributed execution builds on sockets and while this is compatible with TSN, it leaves the challenging and error-prone task of correctly setting up the network for the user.

In "Temporal issues in Cyber–Physical Systems" [30], Broman et al. discuss how accurate clocks can and must be included in cyber–physical systems and how PTP can be used to accurately distribute time. Stanton [31] discusses how distributed coordination of tasks should target a future time instead of reacting to a message which then triggers an immediate action. This reduces the impact of network jitter but is vulnerable to inaccuracies in the time domain.

Gutiérrez et al. [32] showed by daisy chaining TSN nodes that the timeliness of the forwarding and shaping of TSN is well suited for running the control signal network for robotic systems. They did not test this with interfering traffic so no evaluation of stream protection was performed.

For large-scale industrial automation, the Open Platform Communication Unified Architecture (OPC-UA, [33]) is becoming the prevalent solution. The main focus of OPC-UA is device interoperability and is a device-centric, platform-independent architecture to integrate sensors and controllers. Although OPC-UA is primarily a centralized architecture, with amendment 14 [34] a publisher–subscriber ("PubSub") is specified. This approach allows OPC-UA to scale to very large systems since most of the data can flow directly between the devices and not through a central aggregator. One drawback with OPC-UA has always been its complexity and as a system grows, it becomes increasingly difficult to configure the system to achieve desired latency whilst simultaneously not overloading the network and losing traffic as a result [35]. To improve the reliability and determinism of PubSub, Bruckner et al. [36] investigated the usability of TSN and OPC-UA, and a recent joint effort by IEEE and IEC has started the standardization of TSN for Industrial Automation [37] where OPC-UA is part of the standardization effort. Recently the Open-Source Automation Development Lab (OSADL) has begun development to integrate TSN in the Publisher-subscribe profile [38] and provide a ready-made solution for industrial applications.

Another popular architecture is the Distributed Data Service (DDS, [39]), a data-centric, decentralized model that is purely publisher–subscriber. Each node can subscribe to a set of topics and the architecture orchestrates the delivery of data to the right recipients. The updated Robot Operation System (ROS2 [40]) has chosen DDS to handle the data handling. Agarwal et al. [41] used TSN to complement DDS and the simulations showed high message rates with very low latency for real-world data obtained from a wind farm. Their results show that TSN and DDS are a good match for real-world applications.

Communicating Sequential Processes (CSP) was first described by Tony Hoare in 1978 [42] and is a process algebra for specifying and verifying concurrency in systems by using message-passing and channels as synchronization primitives. Modeling concurrent systems suffer from a state space explosion when the size grows. To model and understand how these systems interact at the communication system level, a mathematical framework such as CSP is needed. CSP has successfully been used to verify critical systems such as avionics software at the International Space Station [43]. Messages and channels lend themselves naturally to distributed systems.

Whitney et al. [44] created a Go library called Gluster to support distributed applications in Go. Go is made for programming concurrent applications and is based on CSP. Gluster uses a Master/Worker model where a Master node schedules goroutines on the Worker nodes in the distributed system.

Both OPC-UA and DDS define a protocol format for exchanging data whereas Timed C, which we focus on in this paper, is more of a coordination framework for real-time tasks. In a sense, both Lingua

Franca and Timed C could have used DDS or OPC-UA as the data integration protocol, and compared to our proposed network channel, both DDS and OPC-UA could in theory use this network channel as a transport layer. As this will add several layers of complexity for very little gain, we choose to connect Timed C and *netchannels* and evaluate the result.

To the best of our knowledge, no other work defines similar network channels with the same level of quantitative evaluation to demonstrate how deterministic networking can be used to build composable network primitives for distributed real-time systems.

4. Deterministic network channels and Federated Timed C

We extend Timed C with primitives for inter-task communication via TSN as building blocks to what we informally refer to as "Federated Timed C". These communication primitives are called *netchannels* as they are semantically similar to the existing *fifochannels*. While *fifochannels* can deliver time deterministic inter- and intra-process task communication, netchannels extends this to network task communication.

This section starts out by defining the formal semantics of both types of channels using Time Flow Graphs (TFG), before presenting the reference implementation for deterministic channels in Timed C.

4.1. Formalization

The Time Flow Graph (TFG) formalism introduced in [26] can be used to reason about the composability of Federated Timed C programs. In TFG a Timed C task is represented as a graph $G = (V, E)$ where the nodes $V = (P \cup F)$ is the set of timing points P and code fragments between timing points F. The edges E represent the dependency relations between the nodes. A TFG can be analyzed at two levels of abstraction, at the program level or at the platform level. The highest level is the program level which lacks any information about the physical hardware on which it will execute. Analysis on the program level is purely in the logical time domain where code fragments have zero execution time. The second level is the platform level. It is a refinement of the program level where platform-dependent information about WCET of code fragments, release jitter, and trigger precision is included. Analysis on the platform level is on a refined logical timeline which we shall call the quasi-physical timeline. An operational semantics is also defined as a transition system.

A Federated Timed C program can be represented as a parallel composition of Timed C tasks communicating and synchronizing over asynchronous channels. We define the extended TFG $V_{ext} = (V \cup C) = (P \cup F \cup C)$, where C is a set of abstract communication channels. For Federated Timed C, $C = FC \cup NC \cup IC$, where FC is the set of fifochannels, NC is the set of netchannels and IC is the set of IP channels. IP channels are logical channels of best-effort traffic that connect two separate nodes and are not the primary target in Federated Timed C.[2] They are only added to benchmark the netchannels. Notice that lvchannels are not represented as they are not a message-passing construct, but rather a shared variable. The endpoint of a channel c_i is either a receiver or a sender.

Consider the Federated Timed C program in Listing 3 and its corresponding TFG in Fig. 3. Channel endpoints $c \in C$ are represented by hexagons and the arrows denote the reader and writer. Timing points $p \in P$, are represented by circles and code fragments $f \in F$ by squares. The subscript on the components is a reference to the line number in Listing 3.

[2] As is shown in Section 6, using netchannels *without* stream protection can be detrimental to overall system performance-

```
1  task sensor() {
2    int val;
3    while(1) {
4      val = sense();
5      chan_write(val);
6      fdelay(10, ms);
7    }
8  }
9
10 task processer() {
11   while(1) {
12     int val = chan_read();
13     process(val);
14   }
15 }
```

Listing 3: A simple Federated Timed C program composed of two tasks communicating over an abstract channel and can be realized as either fifochannel, netchannel or IP channel.

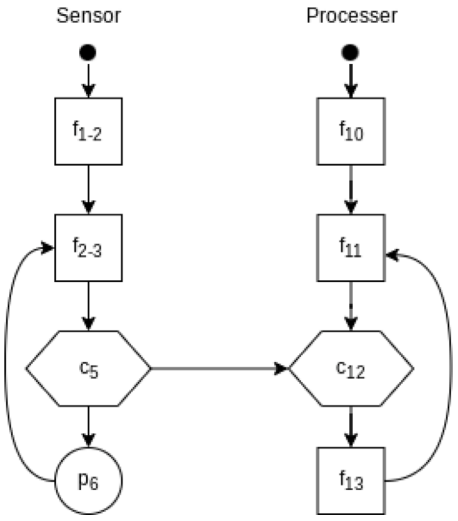


Fig. 3. TFG of Listing 3 for Sensor and Processer communicating over an abstract channel.

We extend and redefine the program property for *arrival time* such that $t_A : V_{ext} \rightarrow \mathbb{T}$. Here $\mathbb{T}$ represents time, it is either a relative time, i.e., a duration, or, in the case of t_A an absolute time. The arrival time of a node is the logical time when it *starts* executing. We also define the *release* time of a node as $t_R : V_{ext} \rightarrow \mathbb{T}$. The release time is the logical time when a node *finishes* executing. Lastly, we define the *latency* of a channel as $t_L : C \rightarrow \mathbb{T}$. The latency of a channel, $t_L(c)$ is the communication delay from the writer to the reader. From [26] $t_D(v)$ is the relative *deadline* of node v.

To formally define arrival and release times we also define the following functions.

- $pTP(v)$ is the set of previous/upstream timing points that has no other timing points between themselves, and the node v. Consider the TFG in Fig. 4. Here $pTP(v) = \{p_7, p_6\}$.
- $pC(v)$ is the set of upstream channels that has no other channels between themselves and the node v. Again consider Fig. 4, in this case, $pC(v) = c_8$. Notice that the cardinality of this set is never greater than one.
- $k : C \rightarrow \{writer, reader\}$ returns the type of channel end. In the case of Fig. 4 $k(c_8) = reader$.
- $o : C \rightarrow C$ returns the other connected channel end. In the case of Fig. 4 $o(c_9) = c_8$.

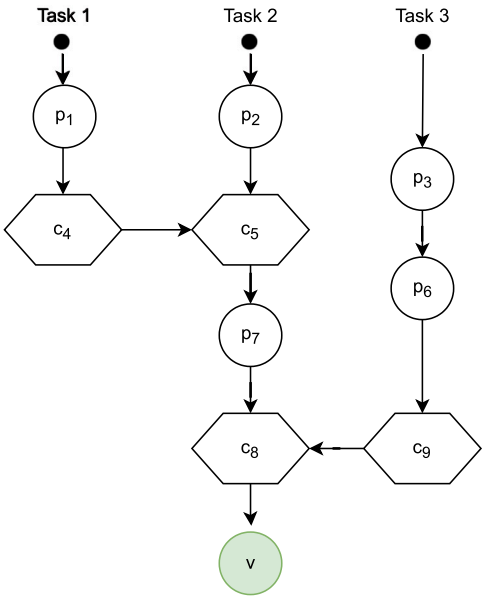


Fig. 4. Example TFG with multiple tasks, timing points, and channels.

Table 1
Arrival and release times for multiple iterations based on from Fig. 3 and Listing 3.

Iteration	$t_A(c_5)$	$t_R(p_6)$	$t_A(c_{12})$	$t_R(c_{12})$
1	0	10	0	0
2	10	20	0	10
3	20	30	10	20

The arrival time of a node is equal to the maximum of the release times of the set of upstream timing points and channels. This can be expressed formally as:

$$t_A(v) = \max\{t_R(u), \forall u \in (pTP(v) \cup pC(v))\} \tag{1}$$

The release time of a node depends on its type. The release time of a code fragment $t_R(f)$ and writer channel $t_R(c_w)$ is per definition equal to their respective arrival times. The release time of a timing point $t_R(p)$ is given in its function argument. E.g., $t_R(p_6) = 10$ ms in Fig. 3 and Listing 3. The release time of a reader channel $t_R(c_r)$ is either its arrival time or the arrival time of its corresponding channel end plus the channel latency. It can be expressed as.

$$t_R(c_r) = \begin{cases} t_A(c_r), \text{ if } t_A(c_r) \geq t_A(o(c_r)) + t_L(c_r) \\ t_A(o(c_r)) + t_L(c_r), \text{ otherwise} \end{cases} \tag{2}$$

$$\forall c_r : k(c_r) \in \{reader\} \tag{3}$$

The channel latency depends on the channel type.

1. Fifochannels are modeled as without latency, i.e., $t_L(c \in FC) = 0$.
2. IP channels have unknown, possibly unbounded latency. $t_L(c \in IC) = t \in \mathbb{T}$
3. Netchannels have unknown but bounded latency. $t_L(c \in NC) = t \in \mathbb{T} : t \leq 2$ ms.

With these additional constructs, we can reason about the logical timing of the Federated Timed C program in Fig. 3. We assume that $c_5, c_{12} \in FC$, i.e., the tasks are communicating through a fifochannel. The arrival and release times must be calculated at each iteration of the loop are calculated in Table 1.

At the first iteration of the loop both *Sensor* and *Processer* will reach their respective channels, c_5 and c_12 at logical time $t = 0$.

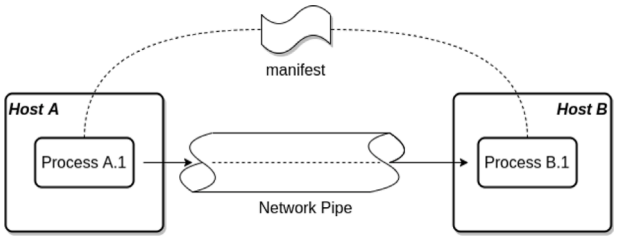


Fig. 5. Two hosts communicating through a network channel with a shared manifest describing the channel.

They perform synchronous communication and are released without advancing logical time. *Processer* contains no timing points and reaches c_{12} in the second iteration while logical time is still 0. *Sensor* contains a timing point p_6 which incurs a 10 ms delay and will not reach c_5 until 10 logical ms. *Processer* therefore blocks on c_{12} for 10 ms. In this way, the channel synchronizes the logical time of the two tasks. Timing information for 3 iterations of Listing 3can be found in Table 1.

A Federated Timed C program is composable to its logical timing if the timing is unchanged by introducing other, independent, Federated Timed C tasks. Formally, a program G_1 is composable when $t_A(v)$ and $t_R(v)$ are bounded or deterministic for all $v \in V$. Under these conditions. it can be composed with other programs, e.g., $G_1 \parallel G_2$ without changing its logical timing.

Per the definition, the only constructs introducing potentially variable timing properties are the channels. A program composed with only fifochannels is deterministic as fifochannels have bounded logical latency. Netchannels have strictly bounded latency and guaranteed delivery provided by the TSN QoS, and thus also satisfy this condition. IP channels, on the other hand, have best-effort delivery (i.e., no QoS from the network) and with a possibly unbounded latency, the channel latency is non-deterministic.

From this, we see that *without* bounded latency guarantees, which is only achievable by reserving transmission slots *and* buffer capacities, a network channel can never be composable. Conversely, by using QoS mechanisms such as TSN that shields critical traffic and provides bounded latency, network channels can be treated as composable components.

Note that composability is a property of our *model* of the program. In actual execution, the zero-delay execution of code fragments would naturally not hold. However, with a time-triggered architecture, such as an RTOS employing TDMA scheduling, execution times would remain deterministic and the overall program, composable.

4.2. Reference implementation overview

We now turn our attention to the implementation example, which is informally referred to as *netchannels* and when used in combination with Timed C becomes *Federated Timed C*. As previously stated, extending a real-time system with reliable networking support complicates code and configuration files (e.g., error handling, the configuration of network interfaces, correct addresses must be set, and QoS parameters specified). A *netchannel* is the implementation of a logical channel and is mapped to a TSN stream. Streams and channels are used somewhat interchangeably in the text. In essence, a *stream* indicates the actual TSN link, whereas a *channel* means the logical channel between two nodes that *may* be a TSN stream (Fig. 1 in Section 3.2).

In Listing 3, we present a trivial example for a network channel which can be shown in Fig. 5. We now adapt this to two Timed C tasks communication using a network channel. In Table 2 we have listed a partial API and C macros that allow us to write very clear and precise code from within a Timed C program.

By adding simple primitives for a reliable channel that follows the same design principles as *fifochannel* in Timed C, our design makes it

Table 2
Brief FTC API summary, for simplicity, only parameters passed to the macros are listed. For full reference, see [2]. *chan* is the *manifest label*, which the macros use to inject a predictable variable name in the code alongside the appropriate function call. Listing 4 describes a channel in a small manifest.

Macro name	Corresponding C-functions	Comment
`NETFIFO_RX(chan)`	`pdu_create_standalone()`	Rx-end of a net-channel
`NETFIFO_TX(chan)`	`pdu_create_standalone()`	Tx-end of a net-channel
`CLEANUP()`	`nh_destroy_standalone()`	De-init and free all PDUs and nethandlers.
`WRITE(chan,val)`	`pdu_send_now()`	Write data to channel and return
`READ(chan,val)`	`pdu_read()`	Reading from the channel will block
`WRITE_WAIT(chan,val)`	`pdu_send_now_wait()`	Write and wait a pre-determined time
`READ_WAIT(chan,val)`	`pdu_read_wait()`	Read and wait a pre-determined time.

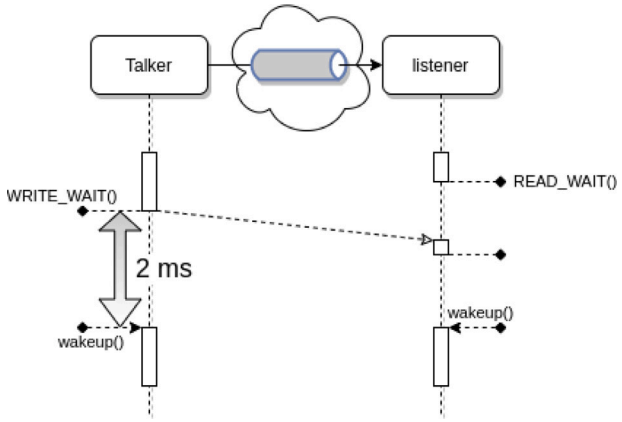


Fig. 6. Two hosts running simultaneous wakeup using `READ_WAIT()/WRITE_WAIT()`-pair with an AVB class A stream providing 2 ms upper bound on E2E delivery latency.

as easy to use a network channel as a normal fifo. The parameters are specified in a manifest file and streams in the network must use globally unique *Stream ID* attributes. A single file can be used for the entire application without fear of "ID collisions". This not only avoids errors where two nodes use almost the same parameters for a stream, but it also opens up the possibility for automated tools to verify network configurations at compile time.

The network channel also has temporal delivery guarantees such that once a network stream has been established it reduces the problem known as the *Byzantine Generals Problem* [45], by ensuring that no frames are unexpectedly lost *during transit*.[3] The macros lend themselves to convenient and clean code, it is important to note that these are simple "glue-ins" for functions whose argument is inferred by the pre-processor and the manifest.

Network channels should be indistinguishable from *fifochannels* in that a sender *writes* to a channel and the receiving end performs a blocking *read*. The example of a DRTS in Fig. 1 shows a set of logical channels connecting the processes running on the different nodes. In Fig. 5 we see an example of the extension made to Timed C, where a network channel is described in a shared manifest and acts as such a logical channel. The channel is currently single-writer/single-reader, but nothing is preventing such a pipe from being multiple readers as it relies on multicast addressing and AVB/TSN stream reservation. Fig. 6 depicts the timing characteristics of such a logical channel where two hosts perform a simultaneous wakeup. This particular scenario is evaluated in Section 6.5.

The code itself does not contain any dependencies to Timed C allowing it to be used as either a standalone system (example tools can be found in the repository [2]) or included in other frameworks. The core system is bundled as a set of header files and libraries and made available through a clear C-API with an accompanying set of macros. In our example tools, we utilize the API provided by the macros as this resulted in the cleanest code and implementation from within Timed C.

For stream protection, the SRP client code from AvNU's OpenAVB project [46] can be found in the srp/ subfolder. The code has been slightly modified to allow for multiple streams and easier integration with our system but is otherwise unchanged. It links to a separate archive and it is marked in the repository where our reference implementation is available [2].

4.3. Detailed architecture

To better explain the underlying architecture, it is useful to expand the example from Fig. 6 to include more tasks in each node. This shows how multiple logical channels are handled by a single socket pair between the two hosts. In Fig. 7, we see that Task A.1 uses the provided WRITE() macro to send data. For the task itself, everything else is abstracted away. Likewise, the listener (task B.1) uses READ() which is a blocking call where the caller waits for data on the declared network pipe. Until Task A.1 writes a value to the pipe, Task B.1 will wait. This pipe is described in the shared manifest (Listing 4). Similarly, multiple tasks on Host A can communicate over *their own logical channels*.

Tasks can also synchronize execution in the temporal domain by ensuring that both tasks continue exactly at the same time using the WRITE_WAIT() and READ_WAIT() pair. For a class A channel, both tasks will wait for 2 ms *after the value was initially captured* ensuring that the sender will wait for exactly 2 ms and the receiver will wait for 2 ms - "network latency". This gives an intuitive primitive for synchronizing two tasks without complex logic. By pairing up two such channels, it is trivial to construct a rendezvous mechanism where nodes wait until both are present before continuing at the same time. Looking back at the requirements for composability in Section 4.1, we see that this channel pair is also composable. In Section 6 we evaluate the accuracy of the simultaneous wakeup.

We split the system into 3 parts: (a) core, (b) writer, and (c) reader.

4.3.1. Core

The core orchestrates the flow of data leaving one writer and data entering destined for a reader. The core is not meant to be used directly by a Timed C task but rather provides the needed infrastructure for the sender and receiver. Core data, such as active network card, PTP clock file descriptor, etc are kept by a `nethandler` container. The core handles resource management which is instrumental in how we can offer a set of simple macros. It also contains a linked list of all active outgoing and incoming channels and finally a map that connects a stream_id to the correct listener task. This last part is how individual streams are handled and is described in Section 4.3.3.

Each network channel is further described internally by a struct detailing (network) address, stream_id, stream reservation handler (from AvNU's OpenAVB project, [46]), callback function for incoming data and buffer space for the latest frame traversing the channel. It also contains a Linux *pipe* pair used to communicate data to/from Timed C

[3] Granted, if parts of the network for some reason are removed, then data will be lost. We will not cover stream duplication and elimination in this paper.

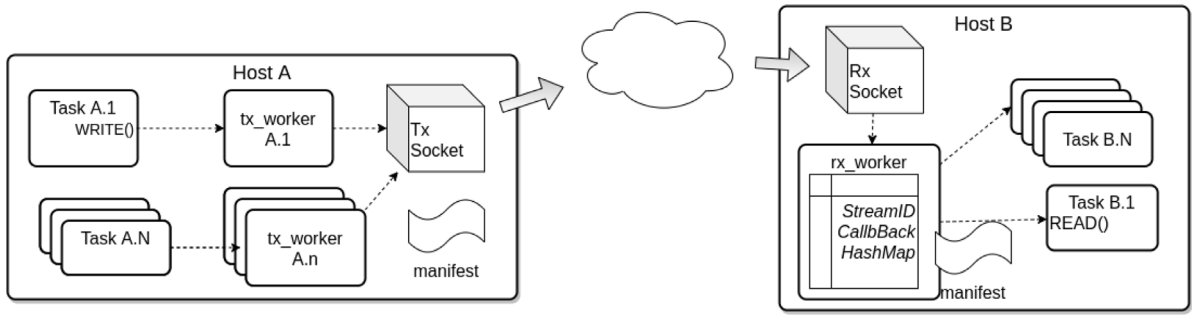


Fig. 7. `net_pipe` detailed architecture. To send a value from Task A.1 to B.1, A.1 first issues a `WRITE()` which feeds the data through a local pipe to a tx-worker. The data is then sent via the tx-socket to the corresponding Rx socket at Host B. B has a single Rx worker responsible for capturing all incoming traffic and forwarding it to the correct local task using the StreamID. Task B.1 receives the data by issuing a `READ()` which blocks on a local pipe awaiting incoming data.

tasks and this is where READ/WRITE terminates. The shared *manifest* is used to describe each network channel and configure it with the correct values.

```
1  struct net_fifo net_fifo_chans[] = {{
2          /* DEFAULT_MCAST */
3          .dst     = {0x01, 0x00, 0x5E,
4                      0x01, 0x11, 0x42},
5          .stream_id = 42,
6          .class   = CLASS_A,
7          .size    = 8,
8          .freq    = 50,
9          .name    = "mcast42"
10 }};
```

Listing 4: Manifest for a system with a single pipe sending 8 bytes of data every 20 ms (50 Hz). The label "mcast42" is used with both macros and functions to indicate which entry to use from the manifest and later, identify the correct object in the functions.

The manifest seen in Listing 4 is an array of `struct net_fifo` which contains:

dst Destination address, a multicast address associated with a specific stream.

stream_id The unique identifier for the stream.

class The AVB QoS class sets the expected observation interval and maximum E2E latency.

size The maximum amount of data each packet contains. In this particular example, we plan to send a single `uint64_t` value, which is 8 bytes.

freq How often data is sent This should be either the frequency of periodic data or an upper bound on how often sporadic data will be sent through this channel.

name a unique moniker used to find the correct entry in the table and also by the macros to inject variables and functions into the code (to properly hook into the core of Federated Timed C).

4.3.2. Writer

For a task to send data, it needs to declare a NETFIFO_TX (see Listing 5). The label used corresponds to an entry in the manifest. During the setup process, each declared channel creates and configures its outgoing socket, and creates a pipe pair where the writer part is made available to the task and the reader-end is handed to a dedicated thread. This thread is what allows us to implement the same abstraction as Timed C's `fifochannel` since the thread will wait for data to arrive and handle all the logic of sending a correctly assembled avtp-frame. If tasked with using stream reservation, proper values are declared via SRP/MRP and announced to the network during the setup phase. Finally, to take advantage of dedicated hardware queues and the Credit Based Shaper, we set the socket priority such that Linux and the mqprio Qdisc can direct the traffic to the correct queue. Our test system uses an Intel I210 Network controller which has 4 tx-queues 2 of which support Credit Based Shaper. A prerequisite for using traffic steering is that Linux Qdiscs are configured (this is shown in Listing 14 in Appendix B).

```
1  #include "manifest.h"
2  const int LOOPS = 1000000;
3  task writer() {
4       NETFIFO_TX(mcast42);
5       for (uint64_t i = 0; i < LOOPS; i++) {
6         WRITE_WAIT(mcast42, &i);
7         sdelay(20);
8       }
9       CLEANUP();
10 }
11
```

Listing 5: Code for talker, based on Fig. 6.

The talker based on the sequence in Fig. 6 is shown in Listing 5. The talker sends an incrementing value every 20 ms (50 Hz) using the channel described by the manifest in Listing 4. Behind the scenes, the core assembles an outgoing frame using the experimental type tag from IEEE 1722 [47], sets the presentation_time to the current PTP timestamp, and updates all relevant fields before sending the frame. It also uses the PTP timestamp from when the frame was created and sleeps for a total of 2 ms (or 50 ms in the case of Class B streams). The current version of netchannel supports Class A and B, TSN's Scheduled Traffic which provides a 100 µs E2E bound is listed as future work. The final CLEANUP() ensures that streams are unannounced (removed from the network) and that memory is freed.

4.3.3. Reader

The receiving end of the channel in Listing 5 is shown in Listing 6, line 6. When declaring a channel to be a receiver (using `NETFIFO_RX()`), the core machinery uses the `mcast42` label to retrieve the fields from the manifest and configure the receiver accordingly. This declares to the network stack that data should be received and delivered to the socket and configures a callback function for the corresponding *StreamID* so that data for this stream is delivered to the correct *fifo*. When the client code calls READ_WAIT(), it blocks in the correct queue until a frame with the expected StreamID arrives.

When a new frame is received for this stream, it is forwarded to the reader. If `READ_WAIT()` is used, the full PTP capture timestamp is reconstructed. The reader schedules a wakeup at the same time as the writer. In this way, the writer knows that the data *will* be received, and *when* the receiver will continue forward. By using the `WRITE_WAIT()/READ_WAIT()` pair, both tasks will continue forward together at the same time.

```c
#include "manifest.h"
task reader() {
  NETFIFO_RX(mcast42);
  uint64_t d = 0;
  while (1) {
    READ_WAIT(mcast42, &d);
    /* use d */
  }
  CLEANUP();
}
```

Listing 6: Code for listener example, based on Fig. 6.

4.4. Automatic stream reservation

From the manifest, the network subsystem extracts the required values to reserve bandwidth correctly all the way from sender to receiver. A talker (writer) first announces available streams and waits for new subscribers (readers) to connect. Note: once announced, the system waits for at least one listener to subscribe before continuing. This is so that the task knows that *someone* is receiving the data. Likewise, a listener waits for a matching announcement before subscribing.

4.5. Description of protocol format

We use a basic AVTP common data header [47, 4.4.3] without any extra subheaders. The type is marked as `EF_STREAM`[4] to signal to any other AVB-capable device on the network that this an experimental stream and most likely non-standard AVB traffic. The manifest specifies the amount of data that will be copied into the `stream_data_payload` from the fifo used by the core of federated Timed C. AVB is an L2 protocol, which means that everything is included directly in the payload field an ethernet frame (with VLAN extension). The protocol format is described in depth in the transport protocol definition [47, Sec. 4.4.4.1].

4.6. Integration with timed C

The core of *netchannels* is a small set of functions and macros that form the API. The code is contained in an archive and can be easily linked with any C/C++ application. Our modification to Timed C limits itself to adding relevant headers and libraries to the build system for Timed C/KTC. For convenience, a fork of KTC with added libraries and headers is published to github [3, branch: net_chan].

- A set of header files exposing required API and constants.
- 2 static libraries libtimedc_avtp.a and libmrp.a (the latter being a slightly reworked client library for AvNu's MRP service to connect and configure the mrpd daemon)
- Add references to timedc_avtp and MRP in Ktc.pm, the configuration file for the KTC compiler
- Add example code for writer and reader to demonstrate how *netchannels* can be used from within Timed C.

The total changeset to KTC is minimal [3] and it also shows that including *netchannels* to any other software system is fairly easy. In this paper, we have evaluated this in Timed C, so this is what we focus on for the rest of the paper.

[4] In truth, we use `AVTP_SUBTYPE_TIMEDC` which is a `#define` to the experimental value.

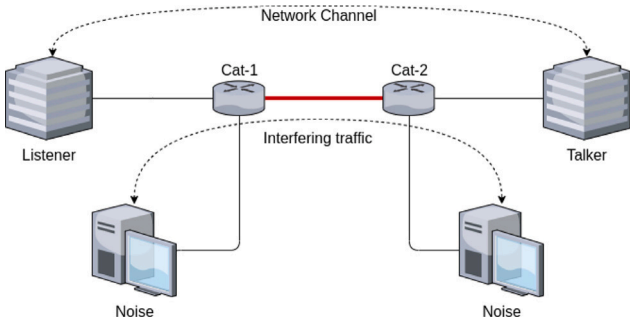


Fig. 8. Test system setup w/network setup. Critical traffic flows between Talker and Listener through a network channel. Additional noise is generated and introduced on the same physical link between the two core switches.

Table 3
Test-machines HW/SW configuration, Talker and Listeners are close to identical in HW configuration, Talker have 32 GB RAM.

	Test machines
Motherboard	ASUS P11C-I
CPU	Intel Xeon E-2224
Freq	3.40 GHz
Memory	16/32 GB
Storage	WD Black SN750 PCIe v3
NIC	Dual I210AT
OS	Debian 11
kernel	v5.16.2-rt19 & 5.10.0

5. Experimental setup and evaluation

In this section, we evaluate the reference implementation of our network channel concept and show how accurate this system can be using GNU/Linux and contemporary x86 hardware.

Fig. 8 shows the test scenario where the test machines were connected via 2 switches (Cisco Catalyst 3650 24PDM-S, with AVB, enabled). Additionally, two extra systems were used to generate a tunable load that could consume all available network bandwidth between the 2 core switches (bridges). This was done to investigate the core contribution of composable network channels so that they will not be affected by adverse network conditions. If TSN QoS guarantees perform as expected, any induced network load should not lead to excessive delay variations and dropped frames.

Table 3 describes the test machines used in the experiment, the only difference being the amount of available memory. As the scenarios are not memory-bound, this is not a relevant difference. Both were configured to run a standard Debian Linux kernel (v5.10.0-11-amd64) and a custom kernel with PREEMPT_RT enabled (v5.16.2-rt19).

5.1. Test scenarios

We used a slightly modified version of the talker and listener code presented in Listings 5 and 6. We enabled logging and needed to be able to adjust real-time measures and SRP behavior. The complete code can be found in Listings 11 and 12 (Appendix A). File I/O is a notorious problem for real-time applications since there are no safe ways to read or write to file in a critical path. Both the IO stack in the OS as well as a storage medium can introduce large, unpredictable delays. Instead, we exploited the available memory in our test machine and cached all the results in memory before finally writing it to disk at the end of the test run. We also allocated and locked all required memory before starting the profiling to avoid page faults during testing.

During testing, we logged the RMS error for the PTP clock reported by `ptp4l`. This allowed us to compound the PTP clock uncertainty with measurement noise from the test application used. If the PTP accuracy is poor (high RMS), we would have less confidence in the distributed system's accuracy.

5.2. Test tools

To quantify the e2e latency and wakeup latency, the code samples shown in Listings 5 and 6 are not well suited to profile the performance. The core code has instrumentation hooks that allow us to capture timestamps in a CSV format.

For every frame leaving or entering the system, timestamps were captured and saved to a buffer and written to a file *after* the test-run had completed. In particular, we captured when an outgoing frame was created (in `pdu_send()` used by `WRITE()`, `WRITE_WAIT()` macros) when the execution was passed to the Rx worker blocking on `recv_msg()` and finally we logged when the task itself was woken and selected to run (when we return from `READ()`, `READ_WAIT()`).

The logs from both sender and receiver were then merged during post-processing to reveal E2E latency and detect dropped frames. E2E was computed from the time the frame left the application to when it was received. This includes delay in the network stack, scheduling jitter, and network transmission delay.

Note: both `WRITE_WAIT()` and `READ_WAIT()` uses `CLOCK_MONOTONIC`to suspend for the necessary time; using the PTP clock directly is not possible in `clock_nanosleep()`. `CLOCK_MONOTONIC` is a strictly increasing clock meaning that even when the local clock is adjusted for drift, it will never move backward. This provides a more stable clock reference, especially when sleeping for relatively short durations. Initial investigation showed that the relative error between `CLOCK_MONOTONIC` and PTP time was not significant for the duration of the sleep cycle as this was less than 2 ms and once `ptp4l` had stabilized, clock adjustments were done in small steps.

5.2.1. Local interference - CPU and file I/O

To simulate a heavily loaded system with a mixture of high CPU utilization and heavy file I/O interrupts, we ran a multi-threaded compilation job of a large software system. Our systems have a total of 4 physical cores, we started a total of 32 threads that would read one or more files, and run through various CPU-intensive compilation stages before writing the result back to disk.

Our experience shows that running a highly parallel compilation task will saturate both the CPU and create a high level of file I/O interrupts and be a highly effective “real-time demonstrator”. The effect is readily measured and will efficiently expose missing real-time measures (e.g., task priorities, task isolation, memory locking, interrupt shielding). We downloaded the latest Linux and ran Listing 7 while our test application was running.

```
make allyesconfig
for i in $(seq 1 100); do
  make clean ;
  make -j32 all > /dev/null;
done
```

Listing 7: Steps to induce local load. The effect of the interference can be observed in Tables 5 and 6 where the effectiveness of real-time measures is also demonstrated.

We also used cyclictest with the following configuration switches to establish a baseline. Results were captured for both vanilla and PREEMPT_RT kernel with the different load scenarios. The results are presented in Section 6.2 and were used to guide the decision for which kernel to use in the test scenarios.

```
cyclictest --duration=3600 -m -S -p90 \
      --policy=rr -i200 -h500000 -q
```

Listing 8: Cyclictest command options used to measure the real-time response of the kernel task scheduler and wakeup machinery.

5.2.2. Network noise generator

To induce a suitable amount of network noise that would trigger a worst-case scenario for (un)protected flows, we created a small tool, `noisegen` [48], that would saturate any link with UDP traffic of a configurable size. For an unprotected stream, any high-rate interfering stream will trigger dropped frames through the network. From an AVB perspective, large frames will induce the highest interference. For our network, UDP packages with a payload of 1470 bytes proved to cause the highest ratio of dropped unprotected frames. By default, the generator saturates the network completely, but it is also possible to reduce transmitted noise to approx. 20% of link utilization or enter a periodic cycle where it will fill a link for N seconds before suspending for another N seconds. (This latter mode is what we see in Fig. 10.) The code was run by specifying the target, payload size, and period (second granularity). The effective bandwidth was logged by the receiver every second, which meant that it did not correlate perfectly with the logs from the Timed C profiling application. The purpose was to induce a network load and quantify the effect of AVB QoS capabilities. We can never perfectly replicate interfering traffic in a network, so any observed jitter, delay, and dropped frames must be captured by the AVB client (our Timed C distributed system). The second granularity allows us to visually adjust the induced load with the plots of the network but will not be used in any rigorous analysis of system performance.

```
 src $> ./noisegen -I 192.168.10.1 -s 1470 -p 30
sink $> ./noisegen -I 192.168.10.1 -s 1470 -r > log
```

Listing 9: Running noisegen to introduce cross-link traffic ref. Fig. 8.

With this setting, an average of 67.700 UDP packets/s with 1470 bytes payload was sent back-to-back. Accounting for header size, checksums, and inter-frame gaps, this corresponded to 83.1% utilization of a 1 Gbps link. On a smaller network without VLAN trunking and service protection, this same setting resulted in 98.87% utilization. For unprotected streams, these conditions caused excessive frame drops. In both networks, unprotected traffic suffered severe packet loss with close to 100% dropped traffic. With a set of 0 received frames, it is not possible to draw any conclusions regarding E2E delay, jitter, or packet delivery. Instead, we used the cyclic mode to periodically apply noise and better illustrate the effect of stream protection.

5.3. 3rd party tools

5.3.1. SRP daemon setup

TSN uses the Stream Reservation Protocol (SRP, [7, Ch. 34]) to dynamically reserve resources for a path between Talker and Listener. The AvNU Alliance has implemented a server through its OpenAVB initiative to handle these configuration messages that run as a standalone service [46].

The daemon has not been modified, but we have extracted the MRP-client code and included it in our project to handle the interface between our application and the `mrpd` daemon. Originally written as a standalone binary, we had to perform some slight modifications to make it into a linkable library, the change is available under srp/ in our code repository [2]. To keep the separation clean, the SRP client code is now compiled into a separate library (`libmrp`) with corresponding headers residing under `include/srp` in the project catalog.

5.3.2. LinuxPTP - Timing setup and accuracy

On each machine, linuxptp [49] has been configured to run in 802.1AS mode. This is provided by a pre-made configuration files for the PTP daemon `ptp4l`. No changes to PTP priority fields were needed for the clients, causing one of the switches to act as GM for the network. As the network ran in isolation, this meant that the absolute time compared to a global reference source such as GNSS was wrong, but

internally in the time domain, all nodes agreed on the same source — which ultimately is what matters. `ptp4l` was run with SCHED_RR and priority 30 to remove most of the interference from other tasks, no other measures were taken to shield it. To reduce the impact of the network noise system running *locally* on the test machines, `ptp4l` was configured to use socket priority 4 in Listing 10. Setting up the network to provide a dedicated strict priority queue for ptp4l is described further in Listing 14.

```
chrt --rr 30 ptp4l -i enp2s0 \
   --step_threshold=1 -f gPTP.cfg \
   --socket_priority=4
```

Listing 10: PTP Setup.

5.3.3. Linux real-time and network configuration

In short, "real-time measures" are all the small and large steps taken to improve the system responsiveness and predictability as experienced by its real-time tasks.

To avoid unpredictable delays due to major page faults, all memory was locked in memory. This is done using `mlockall(MCL_CURRENT|MCL_FUTURE)` which ensures that dynamic memory will not be written to swap. It is important to note that this does not page the memory *in*, so care must be taken to touch all memory before the critical sections begin.

Another common source of execution jitter is the CPUs' aggressive measures to enter lower cstates. This is done to save power and the deeper a CPU transitions, the longer it will take to return. This introduces variations in the order of 100 s of µs, so by default it is disabled by default (to keep cstate untouched, `nf_keep_cstate()` can be used during initialization).

Our test machines have 4 *physical* cores and with the help of HyperThreading (HT) can run up to 8 threads in "parallel". Rather crudely put, HT provides extra virtual cores and will quickly change which thread is executing by swapping a set of hardware registers whenever the executing thread stalls on a memory access. This means that HT can provide several delays where the length is dependent upon the execution path of an unrelated thread. To avoid this, HT has been disabled in BIOS.

To reduce the effect of interference from other tasks and interrupts, a shielded set of 2 cores was created in which the application itself was run. All other movable tasks were moved away from these cores to reduce scheduler interference. It further used SCHED_RR scheduler and priority 80 to out-rank any standard kernel thread and avoid task preemption. All interrupt handlers that could be moved were further affined to other cores by adjusting the IRQ affinity. This is particularly useful when using `preempt_rt` as several interrupt handlers are moved to dedicated kernel threads.

To manage the network traffic, a mqprio qdisc was attached to the I210 network card and configured to use the default AVB VLAN parameters before a Credit Based Shaper Qdisc was attached to Tx-0 and assigned socket priority 3. This is shown in Listing 14. The design and operation of Linux Qdisc are complex and outside the scope of this paper.

A full example of local steps taken to improve RT behavior is presented in Appendix B.

5.4. Running the test scenarios

Once ready, we profiled both kernels to determine the accuracy of a standard and a "real-time tweaked" system. We tested these kernels under different scenarios:

1. *idle*: No other tasks were running other than standard background tasks.
2. *CPU Load*: A local task spawning multiple threads that all induced high CPU load and high IO activity (Section 5.2.1).
3. *Network noise*: High network load on the shared link causing the switches to drop unprotected frames (Section 5.2.2).

We then applied (a) no measures, (b) real-time measures (further described in Section 5.3.3) (c) real-time measures *and* performed stream reservation to protect the network stream. The latter was handled by the core part and the mrpd client library.

By splitting the load and measures this way, we aimed to quantify the different capabilities of the kernels and the real-time measures available in Linux. We also wanted to determine how TSN and stream reservation performed in practice under various loads.

Finally, two longer tests were run with the PREEMPT_RT kernel and all RT measures enabled. The first ran without enabling stream reservation, the latter with the stream being protected by the network as it moved from talker to listener. In addition, heavy network cross-traffic with a worst-case frame size was added to both these long-running tests to cause as much external interference to the traffic as possible.

6. Results

A key requirement for composability is to shield one part of a system from whatever happens in another part. This is done using traditional mechanisms available in Linux such as real-time priorities, CPU shielding, and memory. Expanding from this, a network channel should not be affected by unrelated traffic. The goal of using deterministic network channels as a composable building block is to be sure that traffic is not only delivered but delivered with a *bounded latency* regardless of other traffic. To evaluate this, we designed an experiment where the two test machines should wake up *exactly* at the same time and continue simultaneously with a cooperative task. In our case, this was merely logging the timestamp for later comparison but could be any task requiring some form of synchronicity. The simultaneous wakeup accuracy is therefore the main objective of the tests and should be independent of both local interference and network noise. We ran multiple tests profiling both gPTP accuracy as reported by `ptp4l`, network traversal time ("E2E-latency"), and wakeup accuracy using READ_WAIT() and WRITE_WAIT(). We tested the system as shown in Fig. 8. The entire set of logs obtained and used to generate the tables and figures are made available [50].

6.1. PTP accuracy

For the timestamps captured during testing to be of any value, we must start by quantifying the PTP accuracy. Normally a system will slave its system clock (CLOCK_REALTIME) to the PHC using phc2sys but using the system clock for wakeup can be dangerous when it is being continuously adjusted. Instead, we read the timestamp directly from the PTP Hardware Clock (PHC) to remove one extra service from the test and remove a potential source of errors. It is generally recommended to use CLOCK_MONOTONIC, and we then must use 2 different clock sources; using the PHC is therefore not much extra complexity and saves us an extra source of error.

Fig. 9 shows the PTP RMS accuracy reported by `ptp4l` for a long run (8hr 50 min) with heavy network cross traffic in the setup depicted in Fig. 8. `ptp4l` was run as specified in Section 5.3.2 on a Linux 5.16.2-rt19 kernel. The performance was representative of other measured high-load scenarios, we show the result for the PREEMPT_RT kernel used for the final tests. More details can be found in Table 4. The core switches have the AVB option enabled which means PTP sync messages are timestamped on both ingress and egress. Compared to the wakeup

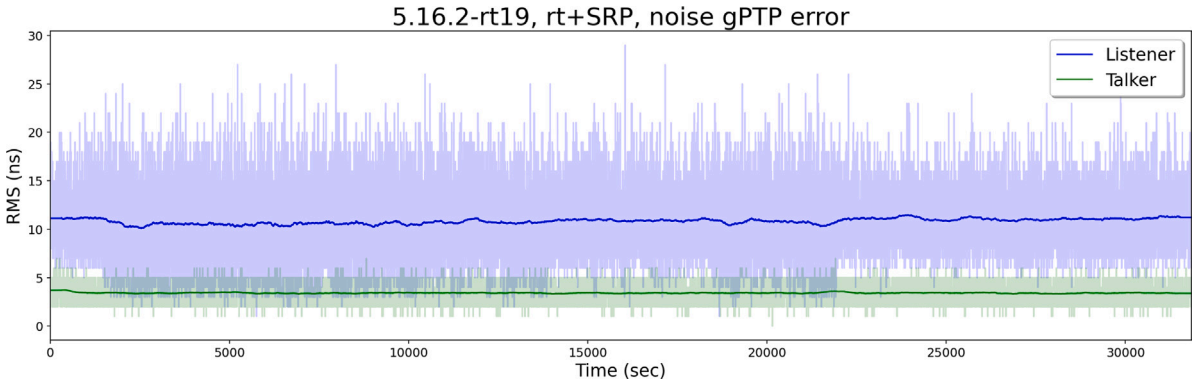


Fig. 9. gPTP performance evaluation, Cat-2 selected as PTP GM. RMS values as reported from gPTP on both hosts during excessive network interference. A 10-min length, simple moving average (SMA) window is overlaid on each dataset.

Table 4
gPTP performance summary for a long test, data represents local time error from the master clock (PTP GM). GM selected by the network to run on switch closets to talker (Cat-2).

	Listener	Talker
duration	8 hr 50min	
max	29 ns	7 ns
avg	10.8 ns	3.4 ns
σ	3.3 ns	0.8 ns

Table 5
Cyclictest performance summary, task wakeup accuracy from `clock_nanosleep()`, 60-minute summary during heavy CPU load, aggregated results from all 4 threads. Wakeup deviation reported from cyclictest scaled to µs.

	5.10.0	5.16.2-rt19
samples	71 796 433	71 999 889
max	3092 µs	16 µs
avg	3.8 µs	1.1 µs
σ	24.5 µs	0.3 µs

accuracy (covered next), this is 3 orders of magnitude better and thus more than sufficient for our tests.

6.2. Kernel wakeup accuracy

To test the real-time capabilities of the selected kernels, we ran cyclictest for 1 h during heavy CPU and IO load (massively parallel make of the Linux kernel source code). This is a common strategy for testing the performance of new RT kernels and configurations. Except for running cyclictest with real-time priority, no other real-time measures were taken. The results are presented in Table 5.

Not only is the maximum latency reduced by a factor of 2 but the standard deviation is also significantly lower yielding improved determinism for the PREEMPT_RT kernel. With these results in mind, we felt comfortable moving forward using the PREMPT_RT kernel. These results also yield lower bounds on the accuracy of our system regardless of network performance on a Linux-based system. *E.g., we cannot expect to achieve better wakeup accuracy than* 16 µs.

Table 6
E2E package latency of 50 Hz stream, the system running without neither network interference nor SRP to reserve network resources. The top half shows an otherwise idle system, and the lower section shows delay values when running a CPU-intensive load.

E2E Delay	5.16.2-rt19	
	No RT	With RT
Idle		
max	697.4 µs	95.8 µs
min	134.7 µs	39.3 µs
avg	471.0 µs	56.5 µs
σ	51.3 µs	1.0 µs
CPU Load		
max	256.7 µs	131.6 µs
min	69.2 µs	59.1 µs
avg	108.4 µs	87.5 µs
σ	9.2 µs	3.8 µs

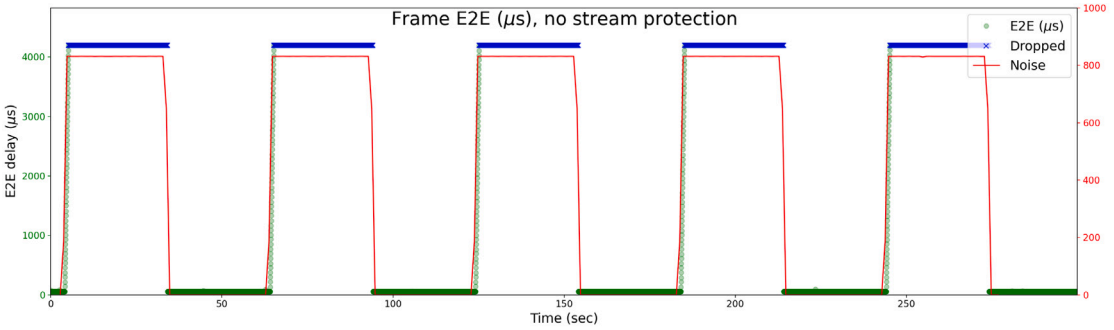
6.3. Package transmission delay

As the machines are running with `ptp4l` and clock accuracy is within 30 ns, we can safely use the PTP timestamps for when a frame was created at WRITE()/WRITE_WAIT() and received to gauge the total delay. The delay measured not only the network delay but also the network stack at both ends. In Table 6 we present the values for 5.16.2-rt19 with *no* network noise and no stream protection. The first test was without any real-time measures, the second using all the measures described in Section 5.3.3.

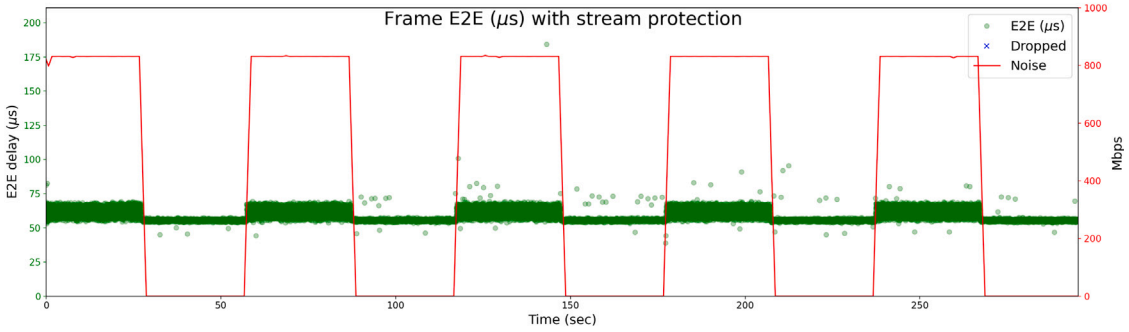
We see a clear improvement using real-time measures such as IRQ affinity, task shielding, real-time priorities, and locking memory to avoid page faults.

6.4. E2E delay with network noise

As shown in Fig. 8, we can apply cross-link noise to the system to evaluate the effect of reserving network capacity. This experiment is an almost exact replica of the package transmission delay except that we now add network interference. Having established the benefit of using real-time measures, we now use this for all tests and demonstrate how stream reservation helps avoid package loss and reduce jitter. In Fig. 10 we see a clear effect of protecting the stream with SRP. Running without protection, we observe a near 100% frame drop in Fig. 10(a) when the noise generator saturates the network. Using stream protection in Fig. 10(b), we can see that *no* frames from the critical stream are lost, yet we notice an increase in E2E delay jitter with approximately 20 µs. This is consistent with worst-case interference



(a) Frame E2E latency *without* stream protection, blue cross indicates dropped frame. 7240 of 15001 frames were lost. Delay increases steadily towards 4ms at which point the bridge's queues are overwhelmed



(b) Frame E2E latency *with* stream protection. 0 lost frames, the largest delay is 185μs.

Fig. 10. Frame E2E delay in a 5-min test, variable network interference in 30-s cycles. Blue crosses indicate lost frames, and green circles represent E2E delay for frames. The right y-axis (red) shows the consumed bandwidth of interfering traffic. See Table 7 for details. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

Table 7
E2E delay, Linux v5.16.2-rt19, the system is idle with periodic network interference (30-sec cycle of 0 to approximately 830Mbps traffic). The first column shows delays and packet loss without any network protection schemes; the second column shows results when SRP is used to reserve network resources. Fig. 10 provide a graph view of same dataset.

	No protection	With SRP
max	4196.9 μs	184.3 μs
min	39.1 μs	39.0 μs
avg	126.2 μs	58.9 μs
σ	437.8 μs	4.3 μs
loss	48.3%	0%

Table 8
E2E summary using RT+SRP, a heavy CPU load on both machines and variable network saturation (5 min noise period). For Listener and Talker, a negative error means waking up *after* targeted time. For relative error, values are absolute (which unit wakes up before the other is not as relevant as how far apart they wake).

	Listener	Talker	Relative error
max	−17.4 μs	−7.9 μs	15.3 μs
avg	−2.3 μs	−2.1 μs	0.25 μs
σ	0.70 μs	0.38 μs	0.80 μs

of large frames for a shared link[5] and internal forwarding delay on the bridge. Table 7 provides additional details where especially the max delay and standard deviation demonstrate the efficacy of stream protection in this case.

6.5. Network triggered wakeup accuracy

The final and most important metric to evaluate for our network channel is the coordinated wakeup accuracy. As described in Section 4, `WRITE_WAIT()` and `READ_WAIT()` will let each node continue at the same time; the goal is to have the talker and listener tasks waking up simultaneously regardless of local load and network noise. Where Fig. 10 shows the necessity of using SRP to protect critical traffic, we now look at the complete service, the accuracy of two systems running a coordinated wakeup with `WRITE_WAIT()` and `READ_WAIT()`. Fig. 11 shows the aggregated wakeup error between the two machines, Table 8 shows the maximum *relative* error in our setup.

The result in Table 8 shows that a shared signal between two hosts on a network can synchronize the execution flow to within 15.3 μs under heavy load and high network interference. When we compare this with the results presented in Table 5, we see that we are within the bounds found by the de-facto Linux real-time performance measuring toolkit.

Oliveira et al. [51] provide a thorough evaluation of PREEMPT_RT on comparable hardware to our test setup. Their load scenarios are more detailed than what we used in our evaluation, yet we see that for heavy loads, they present similar numbers. Similarly, for ARM-based devices, Adam et al. evaluate several flavors of Raspberry Pi (RPi) and kernel versions using cyclictest. The comprehensive summary ([52, Table 2]) they provide shows max latency numbers well below 100 μs for most of the models tested.

[5] A frame of 1470 bytes + headers will consume 12 μs in transmission delay.

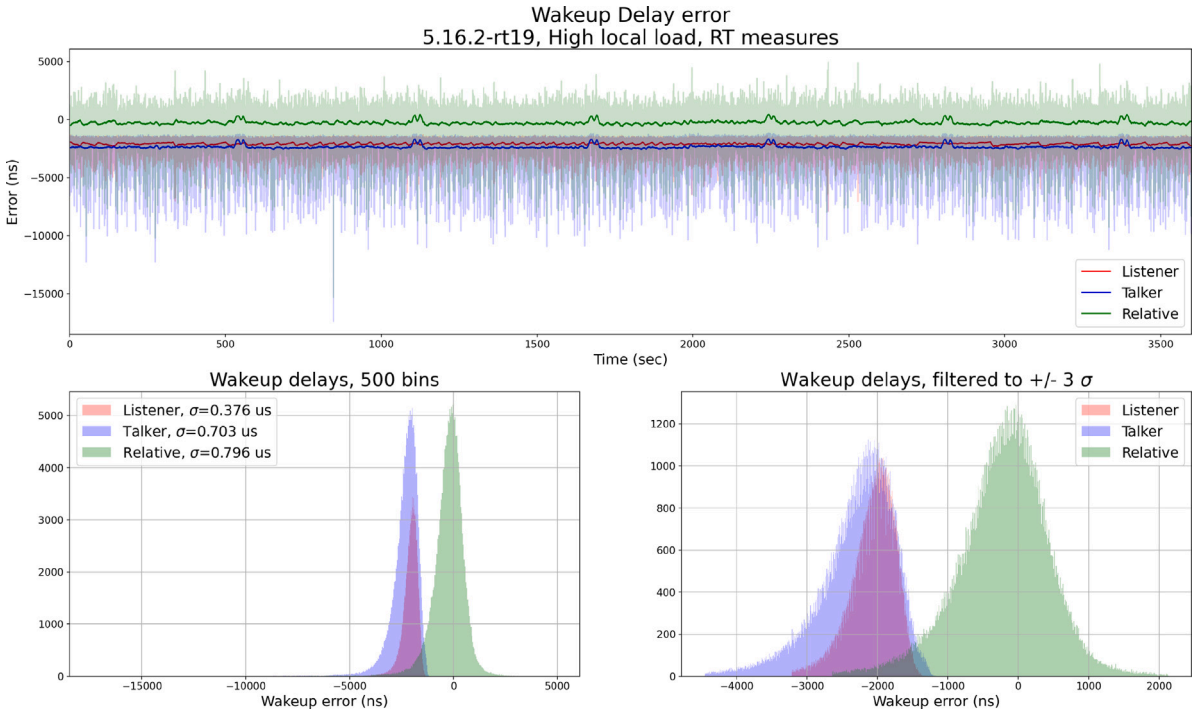


Fig. 11. Delay wakeup accuracy, RT+SRP measures under heavy CPU load and interfering traffic (Kernel 5.16.2-rt19). A negative value means wakeup *after* target. The lower right figure is the same as the left but restricted to $+/-3\sigma$.

6.6. Final remarks

Although the results obtained during testing show a clear advantage of using protected streams, the amount of available hardware constrained the size of our experimental setup. We also only ran a single stream between the talker and listener to reduce the possible sources of errors. Due to the number of systems available for testing, both end stations, and network bridges, we have not been able to accurately quantify the effect of multiple reserved streams. The same limit applies to the number of network bridges between a talker and listeners.

7. Conclusion

In this paper, we have presented network channels, an extension to Timed C and Time Flow Graph that allows us to design, verify and implement robust distributed real-time systems. This has been rooted in a formal definition of TFG. A tool such as TFG is useful when describing large and complex systems, and by extending TFG and Timed C, to the network realm, it becomes possible to describe large, distributed systems with the same tools. For a component such as a network channel to be considered *composable*, the amount of temporal shielding the network can provide must be comparable to the real-time demands. For TSN using stream class A, an upper bound of 2 ms E2E latency is specified in the standard which is what we have evaluated in this paper. Whereas TSN also provides a 100 µs upper bound using the time-aware scheduler, this has not been evaluated in this work as we did not have access to supportive network equipment. A TSN network such as we have used in this paper can therefore only provide composable channels for systems that allow *up to* 2 ms delays.

A reference implementation has been described and evaluated on Linux. We saw that PREEMPT_RT greatly improves system determinism and that SRP and TSN ensure that no packets are lost regardless of network load. By using PTP, we can accurately reason about total transmission delay through both the network and the local network stack on both ends. The accuracy of the PTP clock synchronized during our test runs shows that PTP is 3 orders of magnitude more accurate than scheduler wakeup, allowing us to be confident that PTP timestamps can be used to synchronize task wakeups events.

We showed that our framework for network channels can share signals across a network domain and synchronize the execution of a distributed system both during adverse network conditions and extreme local loads. We show experimentally that the construct fulfills the requirements for composability for systems that accept a 2 ms delay. The API presents itself as clear and simple, leading to a clean design of DRTS. We demonstrated this inside a Timed C example program, and as noted in the beginning, including it in any other C program is trivial.

Finally, we discuss the limitations of our experimental setup and how a future experiment could be constructed to further strengthen the results.

8. Future work

We see maximum E2E delays of more than 100 µs in the system, but we suspect that a large portion of this is due to variance in the delay through the Linux network stack. Another interesting avenue is to look into Linux's Express Data Path (XDP) and move most of the network processing to user space to further reduce network delay and jitter.

As the current revision of the network channel uses class A and B from AVB, a logical next step is to enable the Time Aware Shaper for scheduled traffic and add support for sporadic traffic via the Asynchronous Traffic Shaper.

With the limits discussed in Section 6.6, we see a clear opportunity for running the test scenario with multiple reserved streams as well as adding additional network bridges. It would also be interesting to create multiple streams flowing both ways between the systems acting as talker and listener. This should show how well our reference implementation scales with the number of logical channels as well as adding further pressure on the network bridges.

Finally, as the shared manifest is a single, central place that specifies all streams for the DRTS, combining this with a known network topology to determine routing and time slots for TAS is a necessary next step.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Appendix A. Code examples

See Listings 11, 12 and 13.

```c
#include <stdio.h>
#include <cilktc.h>
#include <timedc_avtp.h>
#include "manifest.h"

static bool running = false;
void sighandler(int signum)
{
   printf("%s(): Got signal (%d), closing\n",
          __func__, signum);
   fflush(stdout);
   running = false;
}

task writer()
{
   printf("%s(): getting ready\n", __func__);
   NETFIFO_TX(mcast42);

   for (uint64_t i = 0; i < LOOPS && running; i++)
   {
         WRITE_WAIT(mcast42, &i);
         if (!(i%100))
            printf("%lu: written\n", i);
         sdelay(20, ms);
   }

   uint64_t stop = -1;
   WRITE_WAIT(mcast42, &stop);
   printf("Magic stop marker written\n");

   CLEANUP();
   return NULL;
}

void main()
{
   nf_set_nic(NIC);
   printf("Run for %d, using %s\n", LOOPS, NIC);

   printf("Using SRP\n");
   nf_use_srp();
   nf_set_logfile("netfifo_talker_rt_srp.csv");
   nf_log_delay();

   running = true;
   signal(SIGINT, sighandler);
   writer();
}
```

Listing 11: Talker example with profiling.

```c
#include <stdio.h>
#include <cilktc.h>
#include <timedc_avtp.h>
#include "manifest.h"

task reader()
{
   NETFIFO_RX(mcast42);
   /* Default, terminal value, unless changed by
    * remote, will terminate loop. */
   uint64_t d = -1;
   while (1) {
         READ_WAIT(mcast42, &d);
         if (!(d%100))
            printf("Counter received! -> %lu\n", d);

         if (d == -1) {
            printf("Magic terminator, stopping\n");
            break;
         }
   }
   CLEANUP();
   return NULL;
}

void main()
{
   printf("Using %s\n", NIC);
   nf_set_nic(NIC);
   nf_use_srp();
   nf_set_logfile("netfifo_listener_rt_srp.csv");
   nf_log_delay();
   reader();
}
```

Listing 12: Listener example with profiling.

```c
#pragma once
#define NIC "enp2s0.2"

#define LOOP_HZ 50
#define IT_MIN (50*60)
#define IT_HR (IT_MIN * 60)
#define LOOPS (5 * IT_HR)

#include <timedc_avtp.h>
struct net_fifo net_fifo_chans[] = {{
         /* DEFAULT_MCAST */
         .dst     = {0x01, 0x00, 0x5E,
                     0x01, 0x11, 0x42},
         .stream_id = 42,
         .class   = CLASS_A,
         .size    = 8,
         .freq    = 50,
         .name    = "mcast42"
}};
```

Listing 13: Manifest for profiling code.

Appendix B. Linux RT setup

See Listing 14.

```
tc qdisc replace dev enp2s0 parent root mqprio \
   num_tc 4 \
   map 3 3 1 0 2 2 2 2 2 2 2 2 2 2 2 2 \
   queues 1@0 1@1 1@2 1@3 hw 0
tc qdisc add dev enp2s0 parent 8001:1 cbs \
   idleslope 20000 sendslope -980000 \
   hicredit 30 locredit -1470 offload 1
ip link add link enp2s0 name enp2s0.2 \
   type vlan id 2 egress-qos-map 2:2 3:3
ip link set enp2s0.2 up
cset shield --cpu 2-3
cset shield -s -p $$

# move network interrupts to reduce
# interference from non-related network load
for irq in $(cat /proc/interrupts | \
   grep -E '(enp1s0|enp2s0$|enp2s0-TxRx-0|enp2s0-TxRx-1)'| \
   cut -d ':' -f1);
do
   echo "Setting affinty for irq ${irq} to: 1"
   echo "1"> /proc/irq/${irq}/smp_affinity
done

for irq in $(cat /proc/interrupts | \
   grep -E '(enp2s0-TxRx-2|enp2s0-TxRx-3)'| \
   cut -d ':' -f1);
do
   echo "Setting affinty for irq ${irq} to: 14"
   echo "1"> /proc/irq/${irq}/smp_affinity
done
```

Listing 14: Linux setup of Qdisc and shielding of 2 cores.

References

[1] S. Natarajan, D. Broman, Timed C: An extension to the C programming language for real-time systems, in: 24TH IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2018, IEEE, 2018, pp. 227–239, http://dx.doi.org/10.1109/RTAS.2018.00031, [ed] Pellizzoni, R., URL http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-235153.

[2] H. Austad, NetChannels, 2022, URL https://github.com/henrikau/net_chan.

[3] H. Austad, KTC fork with pre-built NetChannel extension included, 2022, URL https://github.com/henrikau/ktc/tree/net_chan.

[4] K. Erciyes, Distributed Real-Time Systems, Theory and Practice, first ed., Springer International Publishing, 2019, p. 341, http://dx.doi.org/10.1007/978-3-030-22570-4.

[5] H. Kopetz, Real-Time Systems - Design Principles for Distributed Embedded Applications, in: Real-Time Systems Series, Springer, 2011, http://dx.doi.org/10.1007/978-1-4419-8237-7.

[6] Standard for Local and Metropolitan Area Networks–Audio Video Bridging (AVB) Systems, IEEE Std 802.1BA-2011, 2011, pp. 1–45, http://dx.doi.org/10.1109/IEEESTD.2011.6032690.

[7] Standard for Local and Metropolitan Area Network–Bridges and Bridged Networks, IEEE Std 802.1Q-2018 (Revision of IEEE Std 802.1Q-2014), 2018, pp. 1–1993, http://dx.doi.org/10.1109/IEEESTD.2018.8403927.

[8] Standard for Local and Metropolitan Area Networks–Frame Replication and Elimination for Reliability, IEEE Std 802.1CB-2017, 2017, pp. 1–102, http://dx.doi.org/10.1109/IEEESTD.2017.8091139.

[9] 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), 2016, pp. 1–57, http://dx.doi.org/10.1109/IEEESTD.2016.8613095.

[10] IEEE, Asynchronous Traffic Shaping, IEEE Std 802.1Qcr-2020, 2020, pp. 1–151, http://dx.doi.org/10.1109/IEEESTD.2020.9253013.

[11] J. Specht, S. Samii, Urgency-based scheduler for time-sensitive switched ethernet networks, in: 2016 28th Euromicro Conference on Real-Time Systems, ECRTS, 2016, pp. 75–85, http://dx.doi.org/10.1109/ECRTS.2016.27.

[12] R.T. Braden, D.D.D. Clark, S. Shenker, Integrated services in the internet architecture: An overview, 1994, http://dx.doi.org/10.17487/RFC1633, RFC 1633. URL https://www.rfc-editor.org/info/rfc1633.

[13] D.L. Black, Z. Wang, M.A. Carlson, W. Weiss, E.B. Davies, S.L. Blake, An architecture for differentiated services, 1998, http://dx.doi.org/10.17487/RFC2475, RFC 2475. URL https://www.rfc-editor.org/info/rfc2475.

[14] J. Harju, P. Kivimaki, Co-operation and comparison of DiffServ and IntServ: Performance measurements, in: Proceedings 25th Annual IEEE Conference on Local Computer Networks, LCN 2000, 2000, pp. 177–186, http://dx.doi.org/10.1109/LCN.2000.891025.

[15] N. Finn, P. Thubert, B. Varga, J. Farkas, Deterministic networking architecture, 2019, http://dx.doi.org/10.17487/RFC8655, RFC 8655. URL https://www.rfc-editor.org/info/rfc8655.

[16] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Commun. ACM 21 (7) (1978) 558–565, http://dx.doi.org/10.1145/359545.359563.

[17] M. Lipiński, T. Włostowski, J. Serrano, P. Alvarez, White rabbit: A PTP application for robust sub-nanosecond synchronization, in: 2011 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication, 2011, pp. 25–30, http://dx.doi.org/10.1109/ISPCS.2011.6070148.

[18] Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems, IEEE Std 1588-2019 (Revision of IEEE Std 1588-2008), 2020, pp. 1–499, http://dx.doi.org/10.1109/IEEESTD.2020.9120376.

[19] Kernel Maintainers, The Linux Kernel, 2022, URL https://www.kernel.org/doc/html/latest/.

[20] GNU, The GNU C Library (glibc), 2022, URL https://www.gnu.org/software/libc/.

[21] T. Gleixner, Real-time Linux history, 2022, URL https://wiki.linuxfoundation.org/realtime/rtl/blog#preempt-rt_history.

[22] T. Gleixner, S. Rostedt, J. Kacur, RT-tests, 2022, URL https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/rt-tests.

[23] T. Gleixner, S. Rostedt, J. Kacur, Cyclictest latency debugging with ftrce, 2022, URL https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/tracing.

[24] S. Natarajan, Programming Language Primitives and Tools for Integrated Real-Time Systems Development (Ph.D. thesis), KTH Royal Institute of Technology, 2021, QC 20210517. URL http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-294315.

[25] S. Natarajan, M. Nasri, D. Broman, B.B. Brandenburg, G. Nelissen, From code to weakly hard constraints: A pragmatic end-to-end toolchain for timed c, in: Proceedings - Real-Time Systems Symposium, Institute of Electrical and Electronics Engineers Inc., 2019, pp. 167–180, http://dx.doi.org/10.1109/RTSS46320.2019.00025, QC 20200702 http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-274075.

[26] S. Natarajan, D. Broman, Temporal property-based testing of a timed c compiler using time-flow graph semantics, in: Proceedings 2020 Forum on Specification & Design Languages, FDL, Institute of Electrical and Electronics Engineers (IEEE), 2020, pp. 1–8, http://dx.doi.org/10.1109/FDL50818.2020.9232935, [ed] Alain Girault. URL http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-292191.

[27] P. Derler, T. Feng, E. Lee, S. Matic, H. Patel, Y. Zhao, J. Zou, PTIDES: A programming model for distributed real-time embedded systems, 2008.

[28] M. Lohstroh, C. Menard, S. Bateni, E. Lee, Toward a Lingua Franca for deterministic concurrent systems, ACM Trans. Embedded Comput. Syst. 20 (2021) 1–27, http://dx.doi.org/10.1145/3448128.

[29] M. Lohstroh, I. Romeo, A. Goens, P. Derler, J. Castrillón, E. Lee, A. Vincentelli, Reactors: A Deterministic Model for Composable Reactive Systems, 2020, pp. 59–85, http://dx.doi.org/10.1007/978-3-030-41131-2_4.

[30] D. Broman, P. Derler, J. Eidson, Temporal issues in cyber-physical systems, J. Indian Inst. Sci. 93 (3) (2013) 389–402.

[31] K.B. Stanton, Distributing deterministic, accurate time for tightly coordinated network and software applications: IEEE 802.1AS, the TSN profile of PTP, IEEE Commun. Stand. Mag. 2 (2) (2018) 34–40, http://dx.doi.org/10.1109/MCOMSTD.2018.1700086.

[32] C.S.V. Gutiérrez, L.U.S. Juan, I.Z. Ugarte, V.M. Vilches, Time-sensitive networking for robotics, 2018, CoRR abs/1804.07643, URL http://arxiv.org/abs/1804.07643, arXiv:1804.07643.

[33] Unified Architecture, OPC Foundation.

[34] OPCFoundation, OPC 10000-14 unified architecture part 14 pub sub, OPC UA Online Reference URL https://reference.opcfoundation.org/Core/Part14/.

[35] S. Cavalieri, F. Chiacchio, Analysis of OPC UA performances, Comput. Stand. Interfaces 36 (2013) 165–177, http://dx.doi.org/10.1016/j.csi.2013.06.004.

[36] D. Bruckner, M.-P. Stănică, R. Blair, S. Schriegel, S. Kehrer, M. Seewald, T. Sauter, An introduction to OPC UA TSN for industrial communication systems, Proc. IEEE 107 (6) (2019) 1121–1131, http://dx.doi.org/10.1109/JPROC.2018.2888703.

[37] IEC/IEEE 60802 TSN profile for industrial automation |. https://1.ieee802.org/tsn/iec-ieee-60802/.

[38] Open Source Automation Development Lab contributors, OPC UA PubSub over TSN: OSADL - Open Source Automation Development Lab eG. URL https://www.osadl.org/OPC-UA-PubSub-over-TSN.opcua-tsn.0.html.
[39] Object Management Group, Inc., Data distribution service, 2021, URL https://www.dds-foundation.org/.
[40] ROS: Home. https://www.ros.org/.
[41] T. Agarwal, P. Niknejad, M.R. Barzegaran, L. Vanfretti, Multi-level time-sensitive networking (TSN) using the data distribution services (DDS) for synchronized three-phase measurement data transfer, IEEE Access 7 (2019) 131407–131417, http://dx.doi.org/10.1109/ACCESS.2019.2939497.
[42] C.A.R. Hoare, Communicating sequential processes, Commun. ACM 21 (8) (1978) 666–677, http://dx.doi.org/10.1145/359576.359585.
[43] B. Buth, J. Peleska, H. Shi, Combining methods for the livelock analysis of a fault-tolerant system, in: International Conference on Algebraic Methodology and Software Technology, Springer, 1999, pp. 124–139.
[44] J. Whitney, C. Gifford, M. Pantoja, Distributed execution of communicating sequential process-style concurrency: Golang case study, J. Supercomput. 75 (2019) http://dx.doi.org/10.1007/s11227-018-2649-2.
[45] L. Lamport, R. Shostak, M. Pease, The Byzantine generals problem, ACM Trans. Program. Lang. Syst. 4 (3) (1982) 382–401, http://dx.doi.org/10.1145/357172.357176.
[46] Avnu, OpenAvnu git repository, 2022, URL https://github.com/Avnu/OpenAvnu.
[47] Standard for a Transport Protocol for Time-Sensitive Applications in Bridged Local Area Networks, IEEE Std 1722-2016 (Revision of IEEE Std 1722-2011), 2016, pp. 1–233, http://dx.doi.org/10.1109/IEEESTD.2016.7782716.
[48] H. Austad, Tool repository, 2022, URL https://github.com/henrikau/tools.
[49] R. Cochran, The Linux PTP project. URL https://linuxptp.sourceforge.net.
[50] H. Austad, Logfiles from testruns, NetChannels, 2022, URL https://lethe.austad.us/fs/netchan_22.tgz.
[51] D.B.d. Oliveira, D. Casini, R.S.d. Oliveira, T. Cucinotta, Demystifying the real-time Linux scheduling latency, in: M. Völp (Ed.), 32nd Euromicro Conference on Real-Time Systems, ECRTS 2020, in: Leibniz International Proceedings in Informatics (LIPIcs), vol. 165, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2020, pp. 9:1–9:23, http://dx.doi.org/10.4230/LIPIcs.ECRTS.2020.9, URL https://drops.dagstuhl.de/opus/volltexte/2020/12372.
[52] G.K. Adam, N. Petrellis, L.T. Doulos, Performance assessment of Linux Kernels with PREEMPT_Rt on ARM-based embedded devices, Electronics 10 (11) (2021) 1331, http://dx.doi.org/10.3390/electronics10111331, Number: 11 Publisher: Multidisciplinary Digital Publishing Institute. URL https://www.mdpi.com/2079-9292/10/11/1331.

Henrik Austad received the M.Sc. degree in engineering cybernetics in 2009 from the Norwegian University of Science and Technology (NTNU). He has since worked with authentication systems for distributed computing and the real-time engineering of telepresence codecs for Cisco Systems. Since 2019 he has been a research scientist at SINTEF Digital in Trondheim and is currently pursuing a Ph.D. at NTNU. His research interest include real-time Linux, cyber–physical systems, space applications and deterministic networking.

Erling Rennemo Jellum received his M.Sc. in Engineering Cybernetics from the Norwegian University of Science and Technology (NTNU) in 2020 and is currently pursuing a Ph.D. at the same department. His research interests include real-time systems, models of computation and reconfigurable logic. He co-founded SentiSystems in 2020.

Sverre Hendseth received his Siv.ing. degree (M.Sc. in technology) from the Norwegian Institute of Technology (NTH), in Trondheim, Norway in 1987 and his Dr. Techn. from NTU in 1994. He has since 2003 been an associate professor at the Norwegian University of Science and Technology where he teaches in real-time programming and real-time theory. His research interests include realtime systems, programming languages, and software engineering.

Dr. Geir Mathisen is professor within Dependable Distributed Embedded Systems at Department of Engineering Cybernetics, Norwegian University of Science and Technology Norway, (NTNU). He also has a position as senior scientist at SINTEF Digital. He is teaching real-time systems and design of embedded systems and supervise Ph.D. candidates. He has 30 years of experience in SINTEF, participated in several large industrial research projects, in EU funded research projects and acts as advisor for the industry. Areas of expertise are real time distributed system analysis, system architecture, design of real time control systems and embedded systems.

Torleiv H. Bryne received his M.Sc. and Ph.D. in Engineering Cybernetics in 2013 and 2017, respectively, both from the Norwegian University of Science and Technology (NTNU). He has previously been a Research Scientist at SINTEF, Trondheim, Norway, and is currently an Associate Professor at the Department of Engineering Cybernetics, NTNU. His research interests are in the field of estimation and timing applied to navigation and autonomous systems. Unmanned aerial vehicles and marine applications are the main focus areas of his research. He recently co-founded the spin-off company SentiSystems.

Kristoffer N. Gregertsen received the M.Sc. degree in engineering cybernetics in 2008 and the Ph.D. degree in engineering cybernetics in 2012, both from the Norwegian University of Science and Technology (NTNU) in Trondheim. He has worked at SINTEF Digital since 2012, and is currently senior scientist and research manager for the reliable automation group. His research interest include embedded real-time systems, communication middleware, cyber–physical systems, robotics, smart grid and space applications.

Sigurd M. Albrektsen received his M.Sc. degree in engineering cybernetics in 2011 and the Ph.D degree in engineering cybernetics in 2018, both from the Norwegian University of Science and Technology (NTNU) in Trondheim. He has worked as a research scientist at SINTEF Digital since 2011. Sigurd co-founded SentiSystems in 2020 where he currently serves as the head of embedded development. His research areas include embedded systems, real-time systems, robotics, localization technologies and sensor integration.

Bjarne E. Helvik received his Siv.ing. degree (M.Sc. in technology) from the Norwegian Institute of Technology (NTH), Trondheim, Norway in 1975. He was awarded the degree Dr. Techn. from NTH in 1982. He has since 1997 been Professor at the Norwegian University of Science and Technology (NTNU), the Department of Telematics and Department of information Security and Communication Technology, since 2022 as Emeritus. In the period 2009–2017, he has been Vice Dean with responsibility for research at the Faculty of Information Technology and Electrical Engineering at NTNU. He has previously held various positions at ELAB and SINTEF Telecom and Informatics. In the period 1988–1997 he was appointed as Adjunct Professor at the Department of Computer Engineering and Telematics at NTH. During 2003–2012 Principal investigator at the Norwegian Centre of Excellence Q2S — the Centre for Quantifiable Quality of Service and was in 2020–2021 Principal investigator at the Centre for Research based Innovation NORCICS — Norwegian Center for Cybersecurity in Critical Sectors. His field of interests includes QoS, dependability modeling, measurements, analysis and simulation, fault-tolerant computing systems and survivable networks, as well as related system architectural issues. His current research is on ensuring dependability in services provided by multi-domain, virtualised ICT systems, with activities focusing on 5G++ and SmartGrids.