SoftwareX 23 (2023) 101436

Contents lists available at ScienceDirect

SoftwareX

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

Original software publication

net_chan: Deterministic network channels for distributed real-time systems

Henrik Austad [a,*], Geir Mathisen [b]

[a] SINTEF Digital, Mathematics and Cybernetics, Strindveien 4, N-7035 Trondheim, Norway
[b] NTNU, Engineering Cybernetics, O.S. Bragstads plass 2D, N-7034 Trondheim, Norway

ARTICLE INFO

Article history:
Received 22 February 2023
Received in revised form 22 May 2023
Accepted 7 June 2023

Keywords:
TSN
Real-time Linux
Reliable network-channels
Composable systems

ABSTRACT

Network Channels (net_chan) is an open-source library that provides a network construct for deterministic channels between systems in distributed systems. net_chan is built to harness the Quality of Service guarantees offered by Time Sensitive Networking and the clock accuracy provided by the Precision Time Protocol. The software provides a simple and intuitive API for building distributed systems over packet-switched networks. When run on a system with a deterministic Linux kernel, the system provides an accurate synchronization mechanism between applications running on different hosts.

Code metadata

Current code version	v0.1.2
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX-D-23-00124
Permanent link to Reproducible Capsule	
Legal Code License	MPLv2
Code versioning system used	git
Software code languages, tools, and services used	C, Python3, bash
Compilation requirements, operating environments & dependencies	C11, Linux PREEMPT_RT, AvNU mrpd, LinuxPTP
If available Link to developer documentation/manual	https://github.com/ElsevierSoftwareX/SOFTX-D-23-00124#readme
Support email for questions	henrik.austad@sintef.no

1. Motivation and significance

Since the 1970s, industrial systems have undergone a tremendous digital transformation. This third industrial revolution has since evolved into a fourth, affectionately known as "Industry 4.0". Characterized by pervasive sensor coverage and deeper system integration, previously standalone production cells are now being integrated into complete systems capable of flexible and advanced assembly. When faced with the "IT/OT convergence" that enables back-office systems to directly monitor industrial control loops, traditional Operational Technology (OT) networks must move towards an IT network architecture, and vice-versa. Smart manufacturing is the next logical step where this increased flexibility leads to Virtual Factories, where producers can specify a distributed production facility and let a partially assembled component travel between different factories on its way to completion. This "Factory-as-a-Service" is still somewhere over the horizon, both logistics and local automation flexibility require a level of systems integration previously unheard of before this production panacea can be realized.

One key difference between OT and IT systems is the real-time requirements as OT equipment manages physical processes and systems. A real-time system is a system that alongside the logical requirements also has *temporal* requirements, meaning that not only must a computation be correct, but it must also be so before a specified time. When such a real-time system is composed of elements located on different systems connected using some sort of networking medium, the resulting distributed real-time system will extend the same real-time requirements to the network [1].

Legacy industrial protocols and real-time ethernet variants are rarely compatible, causing equipment from various vendors to be largely incompatible [2]. With the onset of the fourth industrial revolution, the growth of both sensors and network-aware

* Corresponding author.
E-mail addresses: henrik.austad@sintef.no (Henrik Austad), geir.mathisen@ntnu.no (Geir Mathisen).

https://doi.org/10.1016/j.softx.2023.101436

controllers is growing rapidly, making this problem even more acute.

A crucial piece in this puzzle is thus open standards that allow for equipment interoperability and avoids "vendor lock-in". Such heterogeneous systems using Commercial Off-the-Shelf (COTS) IT networks have the potential to scale to much larger systems than what traditional OT systems have typically handled. Time Sensitive Networking (TSN) is a set of open standards that have been developed to provide a common set of traits that can provide robust, real-time capabilities to Ethernet networks. Using TSN terminology, critical traffic is declared as "streams" [3], and each stream is described such that sufficient bandwidth or transmission slots can be reserved along the path the traffic flows using a stream reservation protocol [4,5]. This reservation ensures that no frames belonging to critical streams are dropped due to exhausted buffers. To manage the flow of traffic, TSN specifies a set of *shapers* [6–9] that policies when (or if) packets can be transmitted [10], and within a "TSN Domain", time is synchronized to less than 1 μs error using PTP [11]. If further guarantees are needed, it is also possible to duplicate streams over separate, disjoint network paths [12]which will ensure that frames are protected in the case of a physical network outage. The standards are defined in such a way that one can select a subset of the available standards to reflect the needs of the target network (i.e. for a sensor network that only sees periodic traffic, it may be enough to support a credit-based shaper (CBS [6]), whereas if tight control loops need to co-exist, then the Time Aware Shaper (TAS [7]) should also be supported). Although this reduces vendor lock-in when planning and configuring distributed systems, the increased networking subsystem complexity adds new technical, architectural, and organizational obstacles.

Popular communications frameworks in the industry have seen efforts to improve network reliability using TSN, such as DDS [13] and OPC-UA [14,15]. All of these have shown that TSN is capable of meeting stringent network real-time constraints, yet for constrained systems, or rapid prototyping, s small, simple-to-use library is often desirable. Another framework, EnGINE [16], is a framework for specifying the TSN network itself with network layout, test scenarios, fault injection and careful monitoring.

To help tackle the implementation complexity for effectively moving data from the application to the network layer, we have created `net_chan` [17], a simple-to-use library that is primarily aimed at the *end stations* (ES) and provides a logical channel into which any data value, or type, can be written. `net_chan` expects time to be synchronized using PTP (using LinuxPTP [18]), that the network is TSN Capable, and a Credit-Based Shaper for Linux' QDisc [19] to be available on the ES. Through `net_chan`, a clear C-API is provided, as well as a companion set of preprocessor macros is provided which greatly reduces the burden placed upon the developer. Behind the scenes, `net_chan` will handle the network complexities, ensuring sockets are properly configured, data correctly formatted and the required resources reserved throughout the network using the stream reservation protocol. The accuracy and efficiency have been thoroughly evaluated [20,21] and tested in real network scenarios with heavy interfering network traffic. By having access to an easy-to-understand, robust and efficient construct, extending existing and developing new distributed systems with low latency and high reliability are within easier reach for both research and industrial prototyping alike.

2. Software description

The guiding design principle of `net_chan` is to introduce a deterministic network channel that can be used with minimal setup and configuration. During the setup phase, a set of channel attributes must be provided such as expected data rate, destination address and a unique identifier (see Listing 1). Whereas TSN provides bounded latency and protection against dropped frames, further guarantees can be given to a stream (such as stream duplication), but this lies outside the scope of `net_chan` as it requires a centralized network control. Once such a channel is established and both sender (talker) and receiver (listener) are active, no data will be lost, nor excessively delayed. A reader will then block on a channel until data is received, whereas a write operation will be non-blocking. This is provided that the sender does not send more data than what is specified in the channel manifest (and thus reserved through the network). `net_chan` can also place a fixed delay on a channel that, when paired up with a corresponding reader, will ensure that *both* writer and reader continue *simultaneously*. As a general principle, any safety-critical system should be wary of *blocking* operations. What TSN, and ultimately `net_chan` provides, is a much lower timeout value that enables a much lower watchdog timeout value, thus letting the system react swiftly should the network suffer a catastrophic failure.

2.1. Architecture

For a distributed application, a `net_chan` channel appears as a logical channel which for all practical purposes is a direct connection to the receiving end as shown as the lower pipe in Fig. 1. Each system can have multiple incoming and outgoing channels and this is multiplexed/demultiplexed by `net_chan`. For C applications, a set of helper macros are also provided that makes the initial setup easy. It is important to note that this simplified usage comes with reduced type safety and no obvious way to handle returned error codes. The macros are a good place to start, but once the application is connected, the C-functions are the recommended approach. For C++ applications, it is also a small wrapper available that provides an OOP approach and is documented in the code repository [17]. The same repository also contains an `example/` directory with more elaborate examples.

2.2. Initialization

The first step is to describe each channel used in the system on the form shown in Listing 1 where the destination address, unique id and other relevant stream characteristics are listed. In the application code, a channel must be declared as either Tx (sender/talker) or Rx (receiver/listener). Architecturally, the way outgoing traffic is managed is fundamentally different than how incoming data is treated. Once declared, an outgoing channel cannot be used for incoming traffic and vice versa. In Listing 2, the macro `NETFIFO_TX()` is used to declare a new outgoing channel and during this step, it can also announce the stream capabilities to the network if SRP is enabled (`nf_use_srp()`).

The other end of the channel is described in Listing 3 where a new Rx-channel is declared using `pdu_create_standalone()`. This step will create a new data channel, connect it to the receiving socket, configure timeouts and also announce to the network that a new listener is ready for the particular channel. The standalone suffix indicates that the library should declare and initialize required support structures and keep an internal reference to these. For more advanced usage, the overall governing *nethandler* can also be instantiated and managed directly.

The sample manifest shown in Listing 1 contains fields for a single channel. The name `"sensor"` is used by both ends of the channel to identify the needed attributes, such as payload size, target address and message frequency. To ensure that a stream is properly identified, it also has a StreamID field, which must

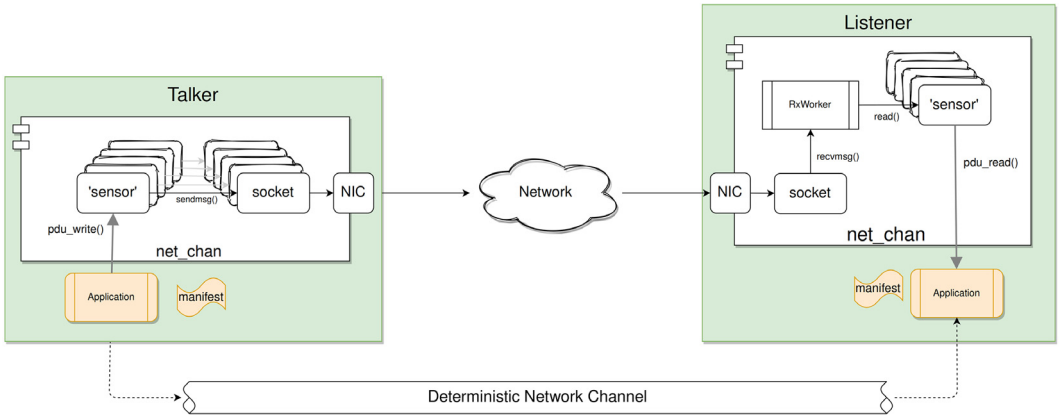

Fig. 1. Overall construction diagram for `net_chan` showing how data flows from one node in a distributed system through the layers, over the network and up the stack at the other end. Each tx-channel has a dedicated socket, which is multiplexed onto a single NIC. Conversely, on the receiving side, a single worker listens on a socket and de-multiplexes incoming data to the correct channels on which the listening application block.

```
1   #pragma once
2   struct net_fifo net_fifo_chans[] = {{
3     .dst       = {0x01, 0x00, 0x5E, 0x01, 0x11, 0x03},
4     .stream_id = 3,
5     .sc        = CLASS_A,
6     .size      = sizeof(uint64_t),
7     .freq      = 50,
8     .name      = "sensor",
9   }};
```

Listing 1: Shared channel manifest file. The file must be included and referenced by each part of a distributed application and except for class, size and freq, all other fields must have unique values (i.e. no overlapping stream_id). `.dst` is an exception as `net_chan` also supports unicast addresses, which must be identical for all streams destined for the same location. However, when using multicast groups, each stream should use its group address. In this example, the payload is a single 8-byte variable, but the payload can be specified to any size (up to max MTU, 1500 bytes).

be unique amongst all the published streams in the network. Specifying all channels in a single manifest-file, it allows tools to determine required bandwidth, assert StreamID uniqueness and so on.

2.3. Usage

Sending data through `net_chan` is done using either `WRITE()` or `pdu_send_now()`. It is also possible to create a synchronized writer-reader pair where we exploit the determinism in TSN to trigger both the talker and the listener to continue simultaneously. For a TSN class A stream, we are guaranteed that the total end-to-end latency will be within 2 ms (50 ms for a class B stream). To use this, either `WRITE_WAIT()` or `pdu_send_now_wait()`[1] which will pause the writer for 2 ms, expecting the reader to do the same. This is illustrated in Fig. 2.

Likewise, the API available to a listener consists of `READ()` /`pdu_read()`, which will block in a read-operation on the underlying pipe. The reader also has a `READ_WAIT()`/`pdu_read_wait()` function which will take the timestamp from when the frame was sent and wait to the same point in time as the talker. The accuracy of this is made possible by the time synchronization provided by TSN.

[1] The somewhat awkward name reflects the behavior, the network frame is sent immediately and *then* the thread will wait

2.4. Shutdown and cleanup

Omitted in both Listing 2 and 3 is the call to `CLEANUP()` which will tear down all channels, unannounce streams and free resources. If an application is terminated without cleaning up, the network will have stale stream reservations lingering, which may cause denied reservations at a later stage.

2.5. Other requirements

The period is an upper bound on the transmission rate and when coupled with payload size yields the needed bandwidth for each channel. Currently, the design allows for multiple incoming and outgoing channels but is limited to a single NIC, i.e. it cannot connect to multiple networks. Furthermore, `net_chan` is designed for Linux and its Qdisc infrastructure, and a few key steps must be performed before running an application extended with `net_chan`.

- `ptp4l` must be started and attached to the same NIC `net_chan` will use.
- AvNU's `mrpd` [22] must be started (running with MMRP, MVRP and MSRP enabled). `net_chan` will interface directly with `mrpd` using a client library extracted from the AvNU project to publish stream reservation messages and parse incoming messages. This allows the library to seamlessly reserve the required bandwidth and buffer capacity between a talker and listeners.

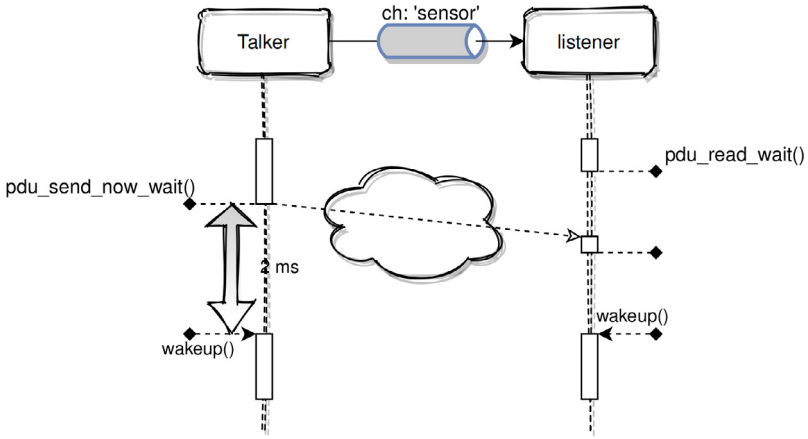

Fig. 2. Timing diagram for a composable channel as described in [20]. With the bounded delay guarantees provided by TSN, it is possible to continue simultaneously using a single writer/reader pair.

- A hierarchical Qdisc using `mqprio` is needed to associate different queues to the available HW queues on the I210 NIC.
- A Credit-Based Shaper (CBS) Qdisc with appropriate idle slope specified.

It is important to note that `net_chan` can function without these steps, but for optimal results, and to reproduce the results obtained in [20], this is needed. Configuration details for mqprio and CBS can be obtained by running the helper-script `nic-bw.py` in the `script/` subfolder and specifying the manifest file for the target application.

Note: `net_chan` currently expects to be the sole application on the ES with active TSN streams, i.e. by using the aggregated outgoing bandwidth specified in the manifest, the resulting idleSlope value used to rate limit outgoing data is not compatible with other applications.

3. Illustrative examples

```
1   #include <netchan.h>
2   #include "manifest.h"
3   void talker() {
4       NETFIFO_TX(sensor);
5       uint64_t data = 42;
6       while (1) {
7         WRITE(sensor, &data);
8         usleep(20000);
9       }
10  }
```

Listing 2: Minimal talker example using the C Macro helpers. A talker sending a sensor value every 20 ms (50 Hz), details of data acquisition omitted.

The talker/listener pair shown in Listing 2 and 3 implement a minimum version of the system shown in Fig. 1. As can be seen, the extra code needed to declare a logical channel "`sensor`" between the two tasks is minimal, allowing the developer to concentrate on the structure of the program rather than setting up and managing the network itself.

In Listing 3, the C-API is referenced directly and as we can see, the code is slightly more verbose. Even so, the benefit should be obvious as we now have access to return codes and can pass along references to the channel to other contexts.

4. Impact

`net_chan` specifically targets small to medium systems where it is relevant to investigate the benefits of using a distributed architecture or the performance difference between distributed and monolithic systems. The primary use case is academic, the software is not intended for high criticality or production systems even though TSN can be used in such scenarios. In `net_chan`, we have taken steps to improve real-time performance such as major page faults (no dynamic memory allocation after channel creation), minor page faults (by using `mlockall()`) and preventing c-state transitions.

With TSN it is possible to create robust, deterministic distributed real-time systems using COTS network hardware which opens up new possibilities for both research projects as well as commercial ideas. Using the provided network channel primitive, we can easily construct more complex mechanisms such as network rendezvous or multicast signaling to trigger multiple hosts at the same time.

This software has been used extensively in our paper [20] where we demonstrate and quantify its ease of use, robustness and accuracy. Using the time-delayed pair (`WRITE_WAIT()`/`READ_WAIT()`), `net_chan` provides a composable network primitive. We plan to continue to use `net_chan` for both sensor integration and control systems in the future and expect `net_chan` to grow alongside the new projects.

5. Conclusions and future work

In this paper, we have presented `net_chan` that provides a network primitive called "network channels". This is made available as a library that can be included in any C or C++ program that instantly provides deterministic network channels provided a GNU/Linux system and a supported Network Integrated Card.

Current testing of `net_chan` has been limited to only a few simultaneous channels of fairly low data volume and -rate. The initial tests have been targeted toward delivery accuracy and reliability with unrelated high interference to determine how well TSN can protect critical traffic. The next set of tests will focus on how non-related critical streams in a TSN network can, or will, affect the timing and determinism of traffic.

```c
#include <netchan.h>
#include "manifest.h"
void listener() {
    struct netchan_avtp *nc = pdu_create_standalone("sensor", false,
                                                    net_fifo_chans,
                                                    ARRAY_SIZE(net_fifo_chans));
    uint64_t data;
    while (1) {
        if (pdu_read(nc, &data) == -1) {
            // handle error
            break;
        }
  /* use data */
    }
}
```

Listing 3: Minimal listener example using the C-API. Errors are caught (but not handled for brevity).

For the future, we aim to further broaden the size of systems being tested from only a few simultaneous channels, to many more and increase the complexity of the systems using `net_chan` substantially. The current implementation uses the cbs Qdisc, the near future will see an update where we instead will use the etf Qdisc which uses SO_TXTIME to send a packet at a specific time. We also plan to extend `net_chan` to smaller systems running FreeRTOS and Zephyr to open the possibility for wider sensor integration. Finally, we hope to extend other real-time coordination frameworks such as Lingua Franca with `net_chan`.

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

No data was used for the research described in the article

Acknowledgments

This work was funded through the Norwegian Research Council under grant 323340.

References

[1] Kopetz H. Real-time systems - design principles for distributed embedded applications. Real-time systems series, Springer; 2011, http://dx.doi.org/10.1007/978-1-4419-8237-7.

[2] Lázaro J, Cabrejas J, Zuloaga A, Muguira L, Jiménez J. Time sensitive networking protocol implementation for Linux end equipment. Technologies 2022-06;10(3):55. http://dx.doi.org/10.3390/technologies10030055, URL https://www.mdpi.com/2227-7080/10/3/55.

[3] IEEE Audio Video Bridging (AVB) systems. 2021, p. 1–45. http://dx.doi.org/10.1109/IEEESTD.2021.9653970, IEEE Std 802.1BA-2021.

[4] IEEE Stream Reservation Protocol (SRP). 2010, p. 1–119. http://dx.doi.org/10.1109/IEEESTD.2010.5594972, IEEE Std 802.1Qat-2010.

[5] IEEE Stream Reservation Protocol (SRP) enhancements and performance improvements. 2018, p. 1–208. http://dx.doi.org/10.1109/IEEESTD.2018.8514112, IEEE Std 802.1Qcc-2018.

[6] IEEE forwarding and queuing enhancements for time-sensitive streams. 2010, p. 1–72. http://dx.doi.org/10.1109/IEEESTD.2010.8684664, IEEE Std 802.1Qav-2009.

[7] IEEE enhancements for scheduled traffic. 2016, p. 1–57. http://dx.doi.org/10.1109/IEEESTD.2016.8613095, IEEE Std 802.1Qbv-2015.

[8] IEEE asynchronous traffic shaping. 2020, p. 1–151. http://dx.doi.org/10.1109/IEEESTD.2020.9253013, IEEE Std 802.1Qcr-2020.

[9] IEEE cyclic queuing and forwarding. 2017, p. 1–30. http://dx.doi.org/10.1109/IEEESTD.2017.7961303, IEEE 802.1Qch-2017.

[10] IEEE Per-stream filtering and policing. 2017, p. 1–65. http://dx.doi.org/10.1109/IEEESTD.2017.8064221, IEEE Std 802.1Qci-2017.

[11] IEEE timing and synchronization for time-sensitive applications. 2020, p. 1–421. http://dx.doi.org/10.1109/IEEESTD.2020.9121845, IEEE Std 802.1AS-2020.

[12] IEEE standard for local and metropolitan area networks–frame replication and elimination for reliability. 2017, p. 1–102. http://dx.doi.org/10.1109/IEEESTD.2017.8091139, IEEE Std 802.1CB-2017.

[13] Agarwal T, Niknejad P, Barzegaran MR, Vanfretti L. Multi-level Time-Sensitive Networking (TSN) using the Data Distribution Services (DDS) for synchronized three-phase measurement data transfer. IEEE Access 2019;7:131407–17. http://dx.doi.org/10.1109/ACCESS.2019.2939497.

[14] Bruckner D, Stănică M-P, Blair R, Schriegel S, Kehrer S, et al. An introduction to OPC UA TSN for industrial communication systems. Proc IEEE 2019;107(6):1121–31. http://dx.doi.org/10.1109/JPROC.2018.2888703.

[15] Li Y, Jiang J, Lee C, Hong SH. Practical implementation of an OPC UA TSN communication architecture for a manufacturing system. IEEE Access 2020;8. http://dx.doi.org/10.1109/ACCESS.2020.3035548.

[16] Rezabek F, Bosk M, Paul T, Holzinger K, Gallenmüller S, Gonzalez A, et al. EnGINE: Developing a flexible research infrastructure for reliable and scalable intra-vehicular TSN networks. In: 2021 17th international conference on network and service management. 2021-10, p. 530–6. http://dx.doi.org/10.23919/CNSM52442.2021.9615529.

[17] Austad H. Netchan v0.1.2. 2023, http://dx.doi.org/10.5281/zenodo.7635611, Zenodo.

[18] Cochran R. The Linux PTP Project. 2021, URL https://linuxptp.sourceforge.net.

[19] Costa Gomes V. TSN: Add QDISC based config interface for CBS. 2017, LWN, URL https://lwn.net/Articles/736335/.

[20] Austad H, Rennemo Jellum E, Hendseth S, Mathisen G, Håland bryne T, Nyborg Gregertsen K, et al. Composable distributed real-time systems with reliable network channels. J Syst Archit 2022. http://dx.doi.org/10.2139/ssrn.4229860.

[21] Austad H, Mathisen G. Bounding the end-to-end execution time in distributed real-time systems: Arguing the case for deterministic networks in Lingua Franca. In: Proceedings of cyber-physical systems and internet of things week 2023. CPS-IoT Week '23, New York, NY, USA: Association for Computing Machinery; 2023, p. 343–8. http://dx.doi.org/10.1145/3576914.3587499.

[22] AvNU. OpenAvnu git repository. 2022, URL https://github.com/Avnu/OpenAvnu.