tpIP: A Time-Predictable TCP/IP Stack for Cyber-Physical Systems

Schoeberl, Martin; Pedersen, Rasmus Ulslev

Published in:
Proceedings of 2018 IEEE 21st International Symposium on Real-Time Distributed Computing

Link to article, DOI:
10.1109/ISORC.2018.00018

Publication date:
2018

Document Version
Peer reviewed version

Citation (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.
tpIP: a Time-predictable TCP/IP Stack for Cyber-Physical Systems

Martin Schoeberl and Rasmus Ulslev Pedersen

Department of Applied Mathematics and Computer Science, Technical University of Denmark
Department of Digitalization, Copenhagen Business School
Email: masca@dtu.dk, rp.digi@cbs.dk

Abstract—Cyber-physical systems are networks of computers connected to the physical world. Often the interaction with the physical world is time critical. In that case computation and communication must be performed in real time. However, a standard implementation of a network stack is hardly time predictable.

This paper addresses the challenge of real-time communication for time-critical cyber-physical systems with a time-predictable network stack. We present tpIP, a real-time implementation of the TCP/IP stack. We achieve time predictability by two properties: (1) the application interface is based on polling functions, instead of blocking sockets, that fits for periodic real-time tasks; (2) the implementation is carefully crafted to enable static worst-case execution time analysis of all functions.

I. INTRODUCTION

Cyber-physical systems are controlling the physical world often under real-time constraints. The computing structure for the controlling is distributed to several computing nodes, which are connected by a network for communication and coordination. Not only needs the computing be time-predictable for such real-time systems, but also the latency of the communication needs to be upper bounded.

The trend is to use standard Internet technology instead of proprietary technologies for the fieldbus. This trend started by using Ethernet and part of the Internet protocols, such as Modbus TCP/IP [18], Modbus over UDP [39], or TTEthernet [33]. This trend is further reenforced by the movement to connect more devices via the Internet, the so called Internet-of-Things (IoT). The original IoT idea was not envisioning to use IoT for real-time and control systems. However, Industrial IoT is the research and development area where Internet technology is used for control applications. To enable Internet technology for real-time systems we need time-predictable communication technology, a time-predictable processor, and a time-predictable implementation of the network software.

At the link layer, a technology such as TTEthernet [33], provides time-predictable transport of packets. However, the software stack above usually follows the standard sockets approach with blocking read and write functions. This blocking is hardly analyzable for the worst-case execution time (WCET) [38]. Furthermore, standard implementations of the TCP/IP stack, which go back to code from the Berkeley TCP/IP implementation [16], are not programmed to enable WCET analysis.

This paper presents tpIP, a time-predictable TCP/IP stack for time-critical cyber-physical systems. We ensure time predictability by following design principles:

- The application interface to the TCP/IP stack is based on polling, instead of blocking accept, read, and write operations on sockets, to allow WCET analysis of tasks. This polling approach fits well with the organization of periodic real-time tasks.
- There is no dynamic memory allocation. All needed buffers are pre-allocated with a fixed configurable maximum size. After usage, the buffers are returned to a free pool of buffers, i.e., they are recycled. Therefore, the maximum memory usage is statically bounded.
- All loops are bounded to enable WCET analysis of TCP/IP stack functions.
- Packet buffers are handled by pointers form layer to layer. This results in a zero-copy implementation to avoid unnecessary memory copy.

Furthermore, as embedded systems might be very resource constrained systems, e.g., wireless sensor networks, we optimize the stack for a very low resource usage. Even a single packet buffer is good enough to support a simple HTTP server or client.

Our network stack is in the same philosophy as the embedded Java network stack ejIP [27]. Particularly, we avoid a blocking API and provide an implementation that it WCET analyzable. The overall aim of this project and the research direction is to provide a full solution (execution stack) of a time-predictable architecture [26].

The contributions of this paper are: (1) a new interface to a network stack that fits real-time systems and (2) an implementation of that network stack that is WCET analyzable. This combination allows to use standard Internet protocols for future time-critical cyber-physical systems. Furthermore, we present two independent prototype implementations of tpIP in C and in Scala in open source.

This paper is organized as follows: Section II presents related work. Section III presents the architecture and design of tpIP. Section IV presents the implementation of the two prototypes of tpIP, discusses the proposed architecture, and provides WCET analysis results. Section V concludes the paper.
II. RELATED WORK

The Berkeley TCP/IP implementation, included in Berkeley’s Unix \[16\], is a well-known open-source TCP/IP stack. As this code is provided with the industry-friendly BSD license, it is used in many commercial implementations of TCP/IP. The stack also introduced the BSD socket API for networking, which is now the de-facto standard API for networking and evolved into the POSIX socket API \[14\]. However, for real-time systems and small embedded systems this network stack might be too large. Therefore, size optimized implementation of TCP/IP have been developed for embedded systems. Furthermore, to implement blocking sockets, support for multi-threading from the operating system is needed.

Two size optimized TCP/IP stack implementations for embedded devices are lwIP (lightweight IP) and uIP (micro IP) \[4\]. Both TCP/IP stacks avoid the need for multithreading by providing a different API than the Berkeley sockets. The application must be organized in a loop, which checks for arrived packets and timeouts. Both generate application events. A programming abstraction called protothreads \[7\] can simplify application programming for the event-driven uIP and lwIP stack. uIP has recently been extended to support IPv6 \[8\]. The main difference between lwIP and uIP is the handling of TCP retransmissions. While lwIP keeps the packet buffers allocated for retransmission, applications using uIP have to reproduce the data on a retransmission. The later approach further saves memory. Our tpIP network stack shares the idea of a polling interface to the network code. However, our focus is on a time-predictable network stack and analyzable memory footprint.

Microcontroller companies often provide a TCP/IP stack optimized for their product, e.g., Microchip’s TCP/IP stack includes its own cooperative multitasking system \[22\]. Cooperative multitasking enforces the user to split longer tasks into smaller ones to avoid long blocking of tasks that need a shorter period. We can envision to use tpIP in a similar single-threaded runtime as there are no blocking operations. However, on multicore, such as the T-CREST \[28\] platform, different layers of the TCP/IP stack can benefit from true concurrency of multiple processing cores.

The focus of RTIP-32 \[35\] is on deterministic and configurable memory usage. RTIP-32 implements the Berkeley socket API. It is intended for embedded systems and needs the On Time real-time operating system. Our tpIP stack is also concerned with bound memory consumption, but we also provide deterministic maximum execution time.

Devices for wireless sensor networks have extreme resource constraints \[12\]. For the first devices TCP/IP was considered too heavyweight for such devices. However, this has changed \[23\]. Using standard TCP/IP protocol for wireless sensor networks is an important move to enable the “Internet-of-things”. One of the first TCP/IP implementations used on wireless sensors was the uIP TCP/IP stack \[5\], \[6\].

The promise of using one protocol as the backbone for Internet-of-things has been outlined in \[37\]. In Internet-of-things, sensors and actuators embedded in physical objects are often linked using the same Internet protocol that connects the Internet.

We seek to develop a TCP/IP stack that is subject to some of the same requirements as listed in \[19\]: the development of the Common Industrial Protocol for Ethernet and standards with particular reference to time synchronization, real-time motion control, and safety.

One of the possible use cases for our embedded IP stack are real-time web-servers \[9\]. They investigate embedded web servers for real-time remote control and monitoring of an FPGA-based on-board computer system.

Some have conducted surveys on how a variety industrial protocols have been developed to address said weaknesses of TCP/IP, which solve the problems of standard Ethernet- and TCP/IP- or UDP/IP-based communication \[3\].

A TCP/IP stack in Java for embedded systems, ejIP \[27\], has been developed for real-time system. The idea for ejIP similar to tpIP, but restricted to Java systems where access to low-level hardware is possible.

III. TIME PREDICTABLE NETWORKING

As we are interested to support real-time systems, the network stack shall be timing analyzable, which means that we can derive statically the WCET bound \[38\] of all functions. Besides coding the base functionality in a time-predictable way, the API needs to be structured to support WCET analysis.

The standard approach to use blocking reads and writes on sockets is not feasible. Instead we propose to use a polling API and non-blocking read and write operations. Furthermore, buffer allocation is handled conservative by setting aside a packet pool before application start.

A. Overview

Figure \[1\] provides an overview of the tpIP stack and its usage. The tpIP stack is organized according to the different protocol layers: datalink, network (IP), transport (TCP/UDP), and transfer (e.g., HTTP, TFTP).

The application is built on top of the transport layer or can directly use the transport layer. The transport layer (TCP or UDP) provides a connection between applications residing on different hosts. Internetwork communication is supported by the network layer, which support routing as well as dispatching on the host based on the transport protocol.

In principle, the whole stack can be organized within a single thread of control. However, to decouple different layers, we can introduce queues of packets between different parts. In the example in Figure \[1\] we decouple the datalink layer from the network layer.

We provide prototype implementations of tpIP first in Scala and then in C to demonstrate that it is time-predictable. With a system, such as Patmos \[30\], it is possible to perform WCET analysis using platin \[11\].

One can argue that time-predictability at the network stack is useless when the underlying network is not time-predictable. Therefore, we envision to use a time-predictable physical layer,
such as TTEthernet [33] or Time-Sensitive Networking, which is part of the IEEE 802.1 working group. Therefore, we can provide end-to-end guarantees for distributed real-time systems. Those systems are also called Industrial Internet-of-Things [13].

### B. Time Predictability

To achieve time predictability and enable WCET analysis programs need to fulfill several properties:

1. Bounded loops
2. Bounded recursion
3. Non-blocking function calls
4. No dynamic memory management on the heap
5. Avoid function pointers

Without having upper bounds on loop iterations and recursion depth no WCET analysis is possible. If the loop bounds are not trivial, they might need to be annotated in a tool specific way. For the implementation of protocols there is usually no need for recursion, therefore this is not an issue for the implementation of tpIP

Functions with a blocking semantic, such as reading from a file, are also an issue for WCET analysis as the blocking time is usually unknown. Therefore, we will define an application programming interface (API) for tpIP that is free of blocking calls. To enable schedulability analysis, real-time tasks are usually organized as periodic tasks with a period and rate monotonic assigned priorities. Therefore, our tpIP stack is also organized around periodic functions. The tpIP main function must be called by the application from within such a periodic loop.

Heap allocation with malloc or new is hardly time-predictable. Therefore, we preallocate all needed buffers at initialization time and avoid all heap allocation during runtime. Allocation on the stack for temporary data is not an issue for WCET analysis.

Function pointers are not in principle an issue for WCET analysis. However, current WCET analysis tools may have a hard time to extract possible target addresses of function pointers from the binary code. Function pointers are useful in a TCP/IP stack to distribute packets depending on registered port numbers to different handlers. Do avoid function pointers
class PeriodicApp(period: Int) extends RtThread(period) {
    val tpip = new Tpip()
    var i = 0

    def run(): Unit = {
        while (true) {
            tpip.run()
            i += 1
            if (i == 3) {
                println("Send a ping")
                val p = tpip.ll.txChannel.freePool.deq
                doPing(p)
                tpip.ll.txChannel.queue.enq(p)
            }
            waitForNextPeriod()
        }
    }
}

Fig. 2: Invoking the network code from a periodic thread and sending a ping after three seconds.

we envision to hard code the dispatch at the application level.

C. A Non-blocking API

The classic interface to the TCP/IP network stack is via sockets and then using blocking read and write calls. This abstraction is elegant as it reuses the API for file IO. However, to enable WCET analysis we need to avoid blocking functions. Therefore, our API in tpIP uses polling functions for read or write. E.g., the data link layer provides new received packets via a FIFO queue. The dequeueing function is non-blocking. When a packet is available it returns the packet. Otherwise, a null pointer is returned. The upper layer, in this case IP, is responsible to periodically check for new packets.

Therefore, the application needs to call the network code in a periodic thread. Figure 2 shows a periodic thread written in Scala, but inspired by the real-time specification for Java [2]. The real-time thread is created with a period as argument for the constructor of RtThread. The thread executes in an endless loop and each call to waitForNextPeriod() waits for the next periodic release of the thread. This pattern is a standard approach for periodic real-time threads. Within the loop the tpIP network stack is invoked by calling tpip.run(). This function contains itself calls all functions from the different layers of the network stack that is shown in Figure 1.

D. Non-blocking Channels

For the communication between layers we provide an abstraction of a channel. A channel contains two queues: (1) a free pool queue and (2) a data queue. A channel is also associated with preallocated packet buffers. Initially all preallocated buffers are put into the free pool queue. When a new buffer is needed, e.g., on receive of an Ethernet frame, a buffer is dequeued from the free pool, filled with the Ethernet content, and put into the receive queue, e.g., from the link layer to the IP layer. The upper layer, in our example IP, dequeues the packet and uses it. When the packet is no longer needed, e.g., because the data has been copied into an application buffer, the packet must be put back into the free pool of the channel. In this way, we recycle the packet buffers.

Furthermore, each of the two queues has just a single reader and a single writer. Therefore, we can use a non-blocking implementation of the queues [15].

We can use these channels between any layers which we would like to run independently. However, as any buffering between components, this adds to the end-to-end latency. Having this decoupling of the layers enables us to use several threads for the tpIP stack, which is especially useful on a multicore architecture. However, with a multicore architecture, such as T-CREST [28], which support on-chip message passing with a network-on-chip, it will be useful to extend this channel to use the network-on-chip.

In our first prototype implementation of tpIP in Scala we just use as receive and a transmit channel between the data link and the IP layer to provide some elasticity and to show the principles. In the C prototype we currently just use a single packet buffer for each direction.

E. WCET Aspects

The tpIP architecture is founded on a minimal number of primitives. One primitive to tpIP is the a queue [17]. IP encompass the following layers: application, transport, Internet, and data link. A queue may be used to connect any of the different layers of the IP protocol [20].

In our example IP stack shown in Figure 1 we insert transmit and receive queues between the network layer and the data link layer. Each of the queues is an object with a maximum capacity (number of empty buffers) defined at initialization time. The enqueueing and dequeueing operations are performed within a periodic thread. This periodic thread is configured at initialization time to deliver at most one packet each period, τx, to the data link layer for the Tx queue. The receive queue, Rx, is likewise configured to be periodically polled each τx, depending on the required timing resolution of the real-time application.

The functional aspects just described are closely related to the WCET aspects of Figure 1. However, starting from the lowest layers, the physical and MAC layers need to provide essential timing guarantees on host-to-host (or process to process) latency and jitter in baseline scenarios such as two hosts in a client/server setup using one shared Ethernet cable. This scenario is important as it provides upper bounds on both the transmission time and the jitter properties of same. For instance, IEEE 1588 PTP [1] can synchronize clocks to less than 10 ns [34] with approx. 99.7% probability.

The tpIP stack software covers especially the network and the transport layers. In this domain of the WCET analysis the main tool is maximum flow analysis of all possible execution flows through the compiled software instructions [21]. The compiled program is partitioned into basic blocks that provide linear execution flows and thus the WCET for a basic block, BBWCET, is just a sum of the execution time for the number of processor cycles for the instructions making up this basic
block. The basic blocks are linked with loops and therefore the WCET analysis includes a max. flow analysis of the “worst” time it can take to execute a given basic block several times.

As an example we consider the checksum calculation and checksum verification which is a common operation both for the transmit and receive part of the tpIP stack. If the payload were defined to be a maximum of 512 bytes, then the 16-bit-oriented checksum would loop up to 256 times. This example points to the influence of the maximum transfer unit (MTU) on the WCET and especially in the cases where the transport protocols can carry a varying size payload.

Finally, the “message” (i.e. the packet payload) arrives to the host application process for which is was intended. The WCET time for the whole stack, as outlined in Figure 1 is the sum of the physical/MAC layer latency and jitter added to the WCET for the actual tpIP stack itself.

IV. PROTOTYPE IMPLEMENTATION

For a start we have chosen two high-level languages (Scala and C) for prototyping, as they allow for quick evaluation of ideas and concepts. However, for the real version of tpIP we need to recode the stack in plain C, as this is still the most common language for embedded real-time systems.

The programming language Scala is running on top of the Java runtime system and hence covers a large part of all the languages in use if one consider them representative for the JVM [36]. However, we currently do not have a time-predictable platform for Scala that supports WCET analysis. The Java processor JOP [25] with its WCET analysis tool WCA [31] would be a time-predictable implementation of the JVM. However, porting the needed Scala runtime libraries to JOP is out of the scope of this work.

Therefore, we restarted the tpIP implementation in C. In that case we can target the time-predictable processor Patmos [30], [32]. For Patmos, we have two WCET analysis tools available: aiT [10] tool from AbsInt and the open-source tool platin [11], which allow static computation of WCET bounds. In the future we will explore to use multiple Patmos cores in the T-CREST multicore platform [28], [29] to speedup the network stack and the application using the network stack.

In Scala we implemented a (special) link layer, IP, ICMP or better known as ping, and base support for UDP. In C we implemented SLIP as simple (and time-predictable) link layer protocol. If the payload (i.e. the packet payload) arrives to the host application process for which it was intended. The WCET time for the whole stack, as outlined in Figure 1 is the sum of the physical/MAC layer latency and jitter added to the WCET for the actual tpIP stack itself.

However, we are aiming to develop the stack from remote places and need a way to tunnel IP packets over the normal Internet. Therefore, we defined a simple link layer protocol where we transmit IP packets over HTTP. We encode the binary packet in “hexadecimal” ASCII characters such that, for instance, 0xAB becomes the characters ‘A’ and ‘B’ (or ‘a’ and ‘b’). That string is used as URL for a HTTP GET request. This URL is the encoded IP packet for the destination, using a web server for receiving those IP packets. After this HTTP GET the HTTP server may deliver some debugging information, which shall be ignored. After the reply to the HTTP GET, the server shall close the connection. A convenient feature of this protocol is that we can simply use a web browser for testing. We call this virtual link layer Blaa Hund.[5]

A valid Blaa Hund GET request (not containing a valid IP packet) is:

GET http://iprt.ngrok.io/adcd0123

We use NGROK[2] as a tunnel to localhost to expose our Blaa Hund server behind a NAT-enabled gateway.

For the C based development we started with implementation on a standard PC and used two serial cables connected with each other. On one serial port we started the Linux version of SLIP (slattach) and on the other serial port we run the tpIP implementation of SLIP.

B. On an Embedded Platform

We target with tpIP embedded real-time systems. Therefore, we need a platform that allows WCET analysis. We chose the time-predictable processor Patmos [30] as such an execution platform. For Patmos, we use the open-source WCET analysis tool platin [11], which allow static computation of WCET bounds.

Patmos is implemented in an FPGA. We use the default configuration of Patmos, which runs on the Altrea/Intel DE2-115 FPGA board for the experiments. We connect a second serial cable to an expansion header for the SLIP interface. The details to reproduce the results are available in a README[3].

C. Code Examples and WCET Analysis

In this subsection we present small code fragments as examples and provide their WCET analysis.

Listing 1: Illustration of IP header checksum

```c
int calculateipchecksum(ipstruct_t* ip_p) {
  unsigned char* h = ip_p ->header;

  int checksum = ((h[0]<<8)+h[1]) + ((h[2]<<8)+h[3]) +
                  ((h[8]<<8)+h[9]) +
                  ((h[12]<<8)+h[13]) + ((h[14]<<8)+h[15]) +
                  ((h[16]<<8)+h[17]) + ((h[18]<<8)+h[19]);
```}

1Correctly written Bl˚a Hund in Danish, which means blue dog, which also happens to be a cafe in Frederiksberg, Denmark, where we invented that protocol.

2online on GitHub at: https://github.com/t-crest/iot-rt/tree/master/tpip
if ((checksum & 0xFFFF0000) > 0)
    checksum = (checksum >> 16) + (checksum & 0x0000FFFF);
if ((checksum & 0xFFFF0000) > 0)
    checksum = (checksum >> 16) + (checksum & 0x0000FFFF);
if ((checksum & 0xFFFF0000) > 0)
    checksum = (checksum >> 16) + (checksum & 0x0000FFFF);
checksum = (checksum & 0xFFFF0000) > 0)

Listing 2: SLIP run function with character receiving

D. An Application Example

To explore tpIP and the time-predictability of using IP-based communication we started to implement a subset of a protocol from a real application. The protocol is used in a railway application for the Austrian Railways’ (ÖBB) as support system for single track lines [24], which is in operation since about 2004.

The OEBB application consists of a master station and devices in the locomotives. The system helps the superintendent at the railway station to keep track of all trains on the track. He can submit commands to the engine drivers. The devices in the trains contain a GPS receiver and check their current position and generate an alarm when the train enters a track segment without a clearance.

At the central station all track segments are administered and controlled. When a train enters a non-allowed segment all trains nearby are warned automatically. This warning generates an alarm at the locomotive and the engine driver has to perform an emergency stop.

The exchange of positions, commands, and alarm messages is performed via a public mobile phone network (via GPRS). The connection is secured via a virtual private network. The application protocol is time-triggered and uses UDP/IP as transport layer. Both systems (the central server and the terminal) regularly send their complete status. Events are transmitted as flags in the message and the reception is acknowledged by setting the according flag. After that the flag is set back and the acknowledge flag in turn as well. The event notification uses therefore a 4-way handshake.

The deadline for the communication of important messages is in the range of several seconds. A network error can be detected with a timeout on not receiving packets from the communication partner. In that case, the operator is informed about the communication error. He is than in charge to perform the necessary actions.

This system is not qualified as a safety-critical system. The communication over a public mobile phone network is not reliable and the system has not certified for safety. The intention is just to support the superintendent and the engine drivers.

We have implemented only a small feature, the event notification and acknowledgment, of the application protocol. We plan to implement the core functionality of the protocol to be able simulate a train. Then we can connect our system to the original traffic display and command program, which is shown in Figure 3.

E. Future Work

Fog computing brings the Cloud “closer to the ground” (to the edge of the network) to enable real-time control. The European training network “Fog Computing for Robotics and Industrial Automation” (FORA) research program focuses on: a reference architecture for Fog computing; resource management and middleware for mixed-critical Fog applications; safety and security assurance; and real-time machine learning. Within FORA we will develop the open-source version of the Fog computing node. The presented tpIP network stack is the starting point for the time-predictable interconnect between FORA Fog nodes. In FORA we will add support for TTEthernet to tpIP.
F. Source Access

The prototype source code for tpIP is available at GitHub in Scala and C [https://github.com/t-crest/iot-rt]. The C based experiments are described in the README at [https://github.com/t-crest/iot-rt/tree/master/tpip]. The compilation and start of the Scala implementation is described in the README at [https://github.com/t-crest/iot-rt]

V. Conclusion

We have created an architecture for an Internet protocol stack, called tpIP, for time-critical cyber-physical systems. The main feature of tpIP is to be time-predictable and that is shall be possible to derive worst-case execution time bounds for the stack when executing on a time-predictable platform. This is possible by changing the classic TCP/IP API from blocking read/write operations to a polling API and to code the stack so that all loops are bounded and that all resources (buffers) are statically allocated. We provide two open source prototype implementations of tpIP.

Acknowledgement

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No. 764785.

REFERENCES


