DOI: 10.15514/ISPRAS-2026-38(5)-2

Architecture for deterministic UDP/IP network stack in ARINC 653 real-time operating system

V.V. Aleinik, ORCID: 0009-0008-2125-7805 <valeinik@ispras.ru>
D.S. Fomin, ORCID: 0009-0002-0989-2861 <d.fomin@ispras.ru>
V.Yu. Cheptsov, ORCID: 0000-0003-3931-101X <cheptsov@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. In this paper we present UDP/IP network stack architecture for an ARINC 653 real-time operating system (RTOS). Potential application in airborne systems imposes a requirement for software certification according to DO-178C. Therefore, we develop network stack from the ground up and focus on structural elements for its construction: architecture of underlying network driver, dynamic memory allocation, packet buffer data structure, packet queue and timeout queue management. To test our network stack and Ethernet network drivers we devise a test system architecture and a set of approaches to system integration. Resulting software is capable to perform IP fragmentation and reassembly or to perform traffic shaping.

Keywords: real-time operating system; network stack; ARINC 653; UDP/IP.

For citation: Aleinik V.V., Fomin D.S., Cheptsov V.Y. Architecture for deterministic UDP/IP network stack in ARINC 653 real-time operating system. Trudy ISP RAN/Proc. ISP RAS, vol. 38, issue 5, 2026. pp. 21-36. DOI: 10.15514/ISPRAS-2026-38(5)-2.

Acknowledgments. We want to thank all members of the hard real-time operating systems development and verification group at ISP RAS for their invaluable assistance.

Архитектура детерминированного сетевого стека UDP/IP в операционной системе реального времени с поддержкой стандарта ARINC 653

В.В. Алейник, ORCID: 0009-0008-2125-7805 <valeinik@ispras.ru>
Д.С. Фомин, ORCID: 0009-0002-0989-2861 <d.fomin@ispras.ru>
В.Ю. Чепцов, ORCID: 0000-0003-3931-101X <cheptsov@ispras.ru>
Институт системного программирования РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. В данной статье представлена архитектура сетевого стека UDP/IP для операционной системы реального времени (ОСРВ) с поддержкой стандарта ARINC 653. Целевое применение транспортного протокола UDP/IP в системах интегрированной модульной авионики приводит к требованию на сертификацию программного обеспечения (ПО) по КТ178-С. Поэтому мы рассматриваем разработку сетевого стека с нуля и акцентируем внимание на его составных элементах: архитектуре нижележащего сетевого драйвера, динамическом распределении памяти, структуре данных для обработки сетевых пакетов, организации очередей пакетов и очередей таймаутов. Для тестирования нашего сетевого стека и сетевых драйверов мы предлагаем архитектуру тестовой системы, а также набор подходов по интеграции сети. Рассматриваемое ПО способно производить фрагментацию и сборку IP-пакетов, а также управлять формой трафика.

Ключевые слова: операционные системы реального времени; сетевые стеки; ARINC 653; транспортный протокол UDP/IP.

Для цитирования: Алейник В.В., Фомин Д.С., Чепцов В.Ю. Архитектура детерминированного сетевого стека UDP/IP в операционной системе реального времени с поддержкой стандарта ARINC 653. Труды ИСП РАН, том 38, вып. 5, 2026, стр. 21-36 (на английском языке). DOI: 10.15514/ISPRAS-2026-38(5)-2.

Благодарности: Благодарим всех сотрудников отдела технологий программирования ИСП РАН, занимающихся разработкой и верификацией операционных систем, за их неоценимую поддержку.

1. Introduction

Deterministic communication is required to construct robust onboard real-time computing systems such as integrated modular avionics and spacecraft control systems. These systems are often constructed from several SoCs, sensors and actuators connected into a real-time network. Timed data flows from sensors to control nodes and from control nodes to actuators form closed feedback loops that control the system. Real-time networks incorporate a variety of technologies: MIL-STD-1553 [1] and SpaceWire [2] networks, AFDX [3] protocol based on UDP/IP [4, 5] over Ethernet [6] network are commonly used in aerospace industry. Besides from its application in AFDX, UDP/IP network stack is a basis for ARINC 615A Data Loader [7] and is also used to prototype distributed algorithms for real-time networks during early stages of system development. For this purpose, UDP/IP either runs on emulated platforms over emulated Ethernet network or it runs on target hardware platform connected to testbed workstation that emulates sensor or actuator of interest. These potential applications make UDP/IP network stack a desired software.

A real-time operating system (RTOS) governs software execution: it enforces robust time and memory partitioning [8] of different applications, provides a standardized API to system services such as file system, error recovery, configuration management, inter-partition communication. ARINC 653 [9] standard specifies API to those services, however its API to network services only includes Name Service (equivalent of local DNS server) and Service Access Points (message queues in which each message has sender and receiver UDP/IP addresses). ARINC 653 provides only basic conceptual guidance to network stack implementers: worst case execution time (WCET) of all

functions must be bounded and small relative to typical feedback loop period, memory footprint must be known «on the ground», before the system is started.

Potential application of UDP/IP network stack in airborne systems imposes a requirement for software certification according to DO-178C [10]. To get the airworthiness certificate software source code must be covered with requirements and tested for compliance. DO-178C prohibits dead code and therefore the codebase functionality must be maintained close to required minimum. These factors filter out approach of software porting as available open-source networks stacks such as lwIP [11] have no requirements and have too much source code to write the requirements by hand. Use of commercial certified network stack is not possible as well.

In this paper we study problems of UDP/IP network stack design, development and verification. We start with a review of architectural patterns found in UDP/IP network stacks both in real-time and general-purpose operating systems. As software design builds upon the interface provided by hardware, we take into consideration the architecture of Ethernet driver. Network stack verification requires attention to details in terms of testbed organization and network emulation; therefore, we study existent approaches to network integration.

We aim for UDP/IP network stack architecture in an ARINC 653 RTOS, however in this paper we focus on its structural elements. These elements include: dynamic memory allocation for real-time OS and packet buffer data structure, packet queue and timeout queue management. To test our network stack and underlying network drivers we devise a test system and a set of approaches to system integration. We hope that presented architectural approaches can be reused independently in design, development or verification of other real-time network stacks.

Presented approaches are implemented in CLOS [12, 13], a real-time operating system developed at ISP RAS, and tested on hardware platforms used in civil aviation and space industries.

2. State of the art

Prior work to this study involved designing network drivers for different network interface cards (NICs) in CLOS and resulted in basic networking capabilities. To distinguish existing and suggested approaches in this paper, we use «CLOS legacy network stack» for an existing CLOS Ethernet layer, and «CLOS extended network stack» for the network stack presented in this paper as it loosely builds on top of the existing driver layer.

2.1 Net driver architecture

Approaches to network stack organization are vastly different even among ARINC 653 operating systems. What remains constant is the concept of system partition: all device drivers and network services are placed in I/O partition, user partition communicates with it using OS services such as SAP or Queuing Channels. System partition allows to isolate fault in one user partition leaving the rest of the system functional and allows to interchange I/O partitions in case software is reused with another physical network. This approach is used in model-driven form in AIR RTOS [14], where all system partition code is configured from XML configuration.

Control flow of NIC device driver can be organized either in interrupt mode or in poll mode. In interrupt mode device gets an interrupt for each received packet and for each completed transmission (to release packet resources). In poll mode driver API functions check device status flags and set return code accordingly. VxWorks, QNX Neutrino, uOS use interrupt mode, CLOS uses poll mode. Interrupt mode is more effective in terms of CPU cycles as driver code does not run without network events, however uncontrolled interrupt arrivals increase WCET of other code in system partition and require synchronization primitives to prevent data races between interrupt handler and the rest of driver code.

Regardless of control flow organization, most Ethernet drivers have similar data flow (Fig. 1). Ethernet driver has packet buffers and descriptors organized into ring buffers. Ring buffer is shared between driver software and NIC hardware. For transmission driver forms packet in tx_data ring

buffer, fills descriptor data field that points to associated packet, forms a descriptor in tx_desc ring buffer and moves tx_tail pointer. Once informed about a new packet NIC hardware initiates a DMA transfer. Completed TX DMA transfer is marked by hardware with updated descriptor status field and updated tx_head pointer. Packet reception is similar to packet transmission: driver sets descriptor data pointer to associated data buffer in tx_data ring, fills descriptor in rx_desc ring, initializes DMA transfer by updating rx_tail. On packet reception NIC hardware updates descriptor status field and updates rx_head pointer. Typically, net drivers fill many descriptors simultaneously – this allows to handle transfers asynchronously and use network bandwidth effectively.

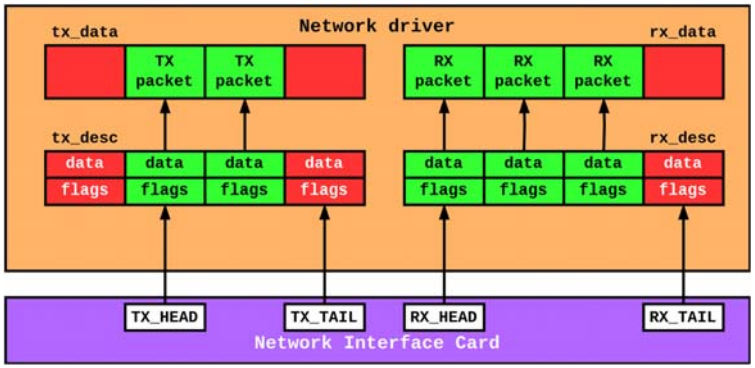

Fig. 1. DMA ring buffer shared between driver and NIC hardware.

In ARINC 653 systems due to static configuration descriptor ring buffers are typically configured and allocated statically and have a rigid association with driver. In contrast, data buffers have a dynamic association with descriptors that is presented by data field. Therefore, data buffers can either be allocated dynamically outside of the driver and passed to the driver in API calls or they can be statically allocated inside the driver in one-to-one correspondence with descriptors. First approach is used in advanced network stacks like lwIP, VxWorks stack and Linux stack [15], second approach is used in CLOS legacy network stack. We want to adopt best practices and therefore in CLOS extended network stack packet buffers are allocated dynamically outside of NIC driver.

2.2 Network stack buffer management

Buffer management is an important part of any networking software. A good buffer management scheme decreases the amount of copying in the data path and enables new usage scenarios.

Let's consider a buffer management scheme of legacy CLOS stack, in which all packet buffers belong to the network driver (Fig. 2). Let's assume that network stack consists of protocol layers and each layer is implemented in its own component and adds its own protocol header. Partition C code calls API functions of upper layer component – to send packets that were forwarded from user partition or originated in system partition.

Every transmitted packet eventually ends up in the driver DMA buffer, therefore upper layer component first acquires pointer to TX buffer from driver. In the process driver may release buffer after previous completed transmission. Afterwards upper layer component forms payload at a specified offset in the buffer and prepends header to it. Then partial packet is passed to lower layer component, where second header is formed. Afterwards lower layer component transmits packet into network driver, where DMA transfer is initiated.

Network stack typically resides in system partition, therefore during queuing channel transition packet is copied from user partition to kernel buffer and from kernel buffer to system partition. If we ignore the step of packet transfer between partitions (it can be done in many different ways) this

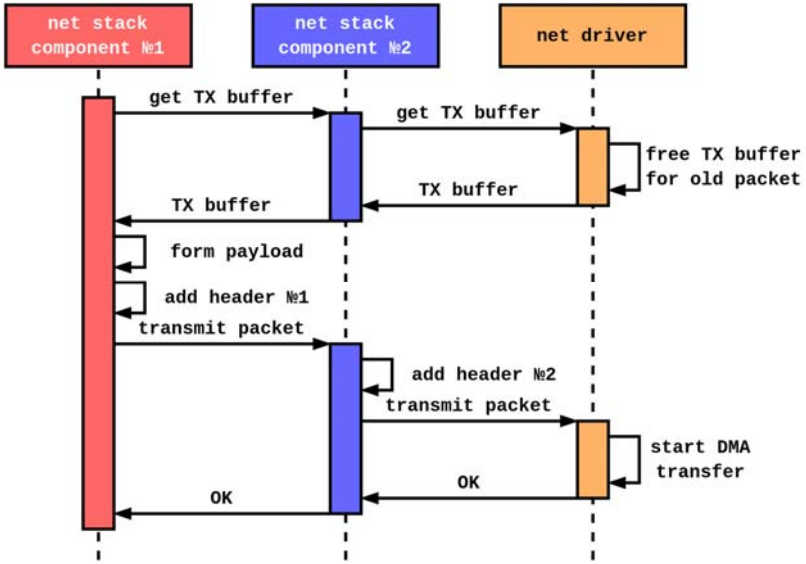

Fig. 2. Buffer is allocated and freed using network driver.

buffering scheme may be called zero-copy. However, this scheme is inflexible as system partition code is unable to first prepare a bunch of packets to transmit later. This functionality is required in ARP [16], where packets accumulate in network interface transmission queue during MAC address resolution, and in IP fragmentation [5], where fragments accumulate in case driver TX queue is full. Moreover, the buffering scheme presented does not support effective packet retransmission: to retransmit a packet network stack must store it in internal buffer and copy it to driver buffer for each retransmission event.

Let's consider a more sophisticated buffer management scheme that is employed in VxWorks, lwIP, uOS, Linux network stacks (Fig. 3). Packet buffer is allocated and freed by a separate buffer allocator. Its allocation mechanism varies among the operating systems: Linux uses advanced caching slab allocator [17], uOS uses classical first-fit allocator with hole coalescing, VxWorks and lwIP use buffer pool allocators with preconfigured fixed buffer sizes. The selection of underlying packet buffer allocator is important: among the mentioned allocators first-fit allocator does not provide bouned WCET while simplistic buffer pool allocator does it.

Second buffer management scheme decouples packet buffer allocator from network driver and separates packet metadata into a distinct data structure (Fig. 4). In Linux it is called *sk_buff* [18] (socket buffer), in lwIP – *pbuf* [11] (packet buffer). First, «pbuf» structure holds the position in the buffer occupied by packet data and supports encapsulation in protocol headers. Headroom and tailroom are memory ranges that are not used by packet data at the moment but can be used if network stack decides to prepend new packet header or append new packet trailer. Second, «pbuf» has a special field that allows to build packet queues out of it. Third, «pbuf» can hold reference counter that allows to associate several «pbuf» structures with one data buffer. These features allow to implement packet queuing requested by UDP/IP stack and packet retransmission functionality required to organize reliable data transfer of any kind.

In CLOS extended network stack, we use second buffering scheme and a packet buffer data structure that resembles «pbuf» used in lwIP.

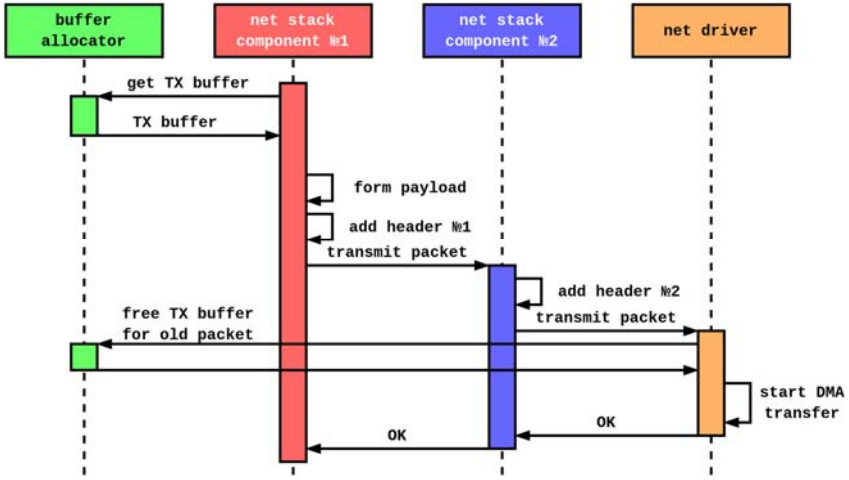

Fig. 3. Buffer is allocated and freed with special buffer pool allocator.

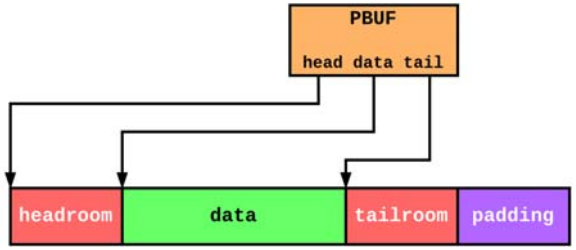

Fig. 4. Packet buffer data structure.

2.3 Approaches to network stack verification

Verification of networking software is non-trivial. Conventional means of software verification require testing. In this paper we take for granted that other means of secure development lifecycle, including, but not limited to static and dynamic software analysis, are already present and used. We also assume formal methods are employed for critical software verification, but leave them out of the scope of this paper.

To start with, testing of networking software requires an environment: during testing there is a system-under-test, a second node that communicates to the first one and a network in between. A second node and a network constitute the environment. Test system must set up the environment, choose the test scenario, run test and decide whether scenario is successful or not. To make matters worse, system-under-test may be represented either by software running in emulator with an emulated network or by software running on actual hardware with real network. First scenario is used to test protocol and distributed system implementations for functional correctness while second one allows to test timing requirements and device driver implementations. In the industry there are many approaches to network stack verification.

Linux network stack is tested with packetdrill [19]. Packetdrill generates traffic using OS system calls. In addition, the tool can construct UDP/IP packets, sniff packets from the kernel and perform

bitwise comparison between captured and generated packets. Packetdrill can run tests on a single machine, however a two-machine setup is also frequently employed.

The AWS IoT Device Tester (Amazon) [20] provides automated testing of network stack implementations for IoT devices. Testing is performed using an echo server: a packet sent by the device is looped back to the same device. This approach is sufficient for simple network stacks with minimal required functionality.

An advanced approach to network interaction is presented in the work from the Institute of Military Science and Technology [21]. The authors describe an automated testbed that can launch an arbitrary number of projects on both real and emulated hardware, and can also incorporate sensor devices emulators. Network deployment is performed automatically. This approach is suitable not only for functional testing of the network stack, but also for latency measurements, protocol prototyping, and full-system testing.

3. UDP/IP stack architecture

3.1 Buffer pool allocator for ARINC 653

In hard real-time operating systems dynamic memory allocation is considered an anti-pattern for two reasons. First, all hard real-time code must have a bounded and small WCET while most allocators that implement interface of malloc (void *malloc(size_t size)) from C standard library are optimized for average performance instead of worst-case performance. Second, most allocators suffer from memory fragmentation: list of all prior allocations and deallocations is not sufficient to predict whether next allocation will succeed or not, a knowledge of allocator internal structure is required.

To make a simple allocator with bounded WCET we abandon arbitrary sized memory regions in favor of buffer pool with preconfigured fixed buffer sizes. Buffer pool is associated with ARINC 653 partition and is allocated statically by CLOS memory subsystem [8]. Buffer pool configuration (Fig. 5) specifies number of buffers for each size contained in buffer pool.

Buffer allocation algorithms is as follows:

- User calls function (void* OsAllocBuffer(POOL* allocator, uint16_t size)) to request buffer with at least size bytes.
- Allocator performs binary search over buffer sizes to find pool with at least one buffer of conforming size.
- Allocator retrieves a buffer from free list.

```
<Memory>
 <OsBufferPool Name="memory" EnableCache="TRUE">
  <BufferType Count="32" Size="64"/>
  <BufferType Count="32" Size="128"/>
  <BufferType Count="16" Size="256"/>
  <BufferType Count="16" Size="512"/>
 </OsBufferPool>
 <OsBufferPool Name="desc" EnableCache="TRUE">
  <BufferType Count="256" Size="16"/>
 </OsBufferPool>
</Memory>
```

Fig. 5. Request for two buffer pools in partition XML configuration file.

Buffer deallocation algorithm is as follows:

- User calls function (void OsFreeBuffer(void* buffer)) to free previously requested buffer.

- Library performs binary search over memory ranges of all allocators in a partition to determine buffer pool allocator that produced the buffer. To optimize this stage user can pass allocator explicitly.
- Allocator performs a second binary search over all buffer sizes in a buffer pool.
- Allocator inserts buffer in free list of buffers of appropriate size.

This simplistic algorithm has low and bounded WCET for memory allocation and deallocation and is suitable for use in hard real-time environments.

3.2 Packet queue management

To organize packet queues, we use intrusive circular doubly linked list (Fig. 6). If we want to organize a queue of NET_PACKET structures, we insert QUEUE_ELEM into it and use CONTAINER_OF to retrieve encapsulating object for queueElem. In contrary to classic linked list organization this approach allows to store one object in several unrelated queues. We can also organize polymorphic queues that contain elements of different types: each QUEUE_ELEM element stores dynamicType that represent type of encapulation (EVENT_TYPE in the example) and is used to dispatch to corresponding handler procedure.

Code example in Fig. 6 illustrates a simplistic event handling system: functions AddEventGpioImpulse and AddEventNetPacket can be used to add events to the system while function HandleOneEvent is used to handle events. Note that organization of event flow decouples time management and memory management from functional logic. Number of objects in the queue is limited by corresponding buffer pools size, number of events handled in one call is determined by WCET of longest event handler and availible processing time in the periodic schedule. Both values are configured by system integrator.

3.3 Management of timeouts

To manage timeouts network stack needs to set a new timeout, to check whether closest active timeout has passed and to disarm an active timeout that is no longer needed. An advanced network stack has many timeouts of different sources and requires a specialized data structure for scalable handling of timeouts. As timeout queue we use red-black tree with intrusive tree nodes, circular linked list for elements with matching key and cached minimum element. As index key we use timeout deadline time, a 64-bit value.

It has been proven [22] that worst-case running time of insertion and deletion is O(lg(n)), where n is the number of timeouts in the system. Red-black tree is expected to behave better then AVL tree in terms of average running time as AVL tree rebalancing can perform up to O(lg(n)) rotations [23] while red-black tree is isomorphic to 2-3-4 B-tree and therefore performs at most three rotations per operation. Both CLOS and Linux OS schedulers use red-black tree to manage timeouts, making it a reliable choice.

3.4 Address Resolution Protocol

UDP/IP network stack implies an underlying ARP implementation. During design we aimed for address resolution implementation that would allow to use our network stack both in conventional non-real-time environment and in deterministic network. Deterministic network requires packet transmission procedure to have a bounded WCET. This requirement contradicts to ordinary ARP operation scheme: to resolve an address ARP broadcasts packets that interfere with all data flows in the network and unknown duration of address resolution procedure is included in WCET for packet transmission. To make ARP hard-real-time-friendly one should split data exchange into two stages: first setup stage has no real-time communication and consists only of ARP requests and replies, after setup phase all IP-MAC pairs are marked as permanent and during second operational stage real-time communication takes place. This scheme can be implemented more easily if network topology

and addresses are known in advance and preconfigured as OS boot parameters or rest in persistent storage.

```
typedef enum {
  EVENT_IMPULSE,
  EVENT_PACKET
} EVENT_TYPE;

typedef struct {
  QUEUE_ELEM queueElem;
  UINT8 impulseType;
} IMPULSE;

typedef struct {
  QUEUE_ELEM queueElem;
  UINT8* data;
} NET_PACKET;

VOID AddEventGpioImpulse(NETQUEUE* queue, IMPULSE* impulse) {
  OsNetQueueElemInit(&packet->queueElem, EVENT_IMPULSE);
  OsNetQueueAddTail(queue, &impulse->queueElem);
}

VOID AddEventNetPacket(NETQUEUE* queue, NET_PACKET* packet) {
  OsNetQueueElemInit(&packet->queueElem, EVENT_PACKET);
  OsNetQueueAddTail(queue, &packet->queueElem);
}

VOID HandleOneEvent(NETQUEUE* queue)
{
  QUEUE_ELEM* elem = OsNetQueueRemoveHead(queue);
  if (elem == NULL) {
    return;
  }

  switch (OsNetQueueElemDynamicType(elem))
  {
  case EVENT_IMPULSE:
    IMPULSE* impulse = CONTAINER_OF(elem, IMPULSE, queueElem);
    // Transmit GPIO impulse.
    break;
  case EVENT_PACKET:
    NET_PACKET* packet = CONTAINER_OF(elem, NET_PACKET, queueElem);
    // Transmit network packet.
    break;
  default:
    break;
  }
}
```

Fig. 6. Example of intrusive polymorphic linked list usage.

We design our ARP implementation as set of state machines, one state machine for each IP address in ARP table (Fig. 7). Each state represents current stage of address resolution process. ARP entry states include:

- NOT_PRESENT – IP address is not known or long forgotten.
- PERMANENT – IP address is known and has a fixed MAC address.
- PENDING – IP address is not known, numRetries ARP request retries happened.
- FAILED – IP address resolution failed; ARP will mark host as unreachable for some time.
- REACHABLE – IP address is resolved; ARP will mark host as reachable for some time.

- OUTDATED – IP address has an associated MAC address, however ARP performed numRetries request retries to validate IP-MAC correspondence.

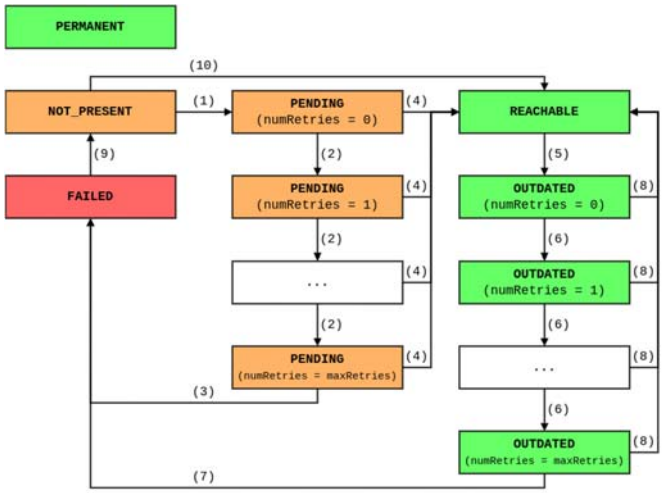

Fig. 7. ARP IP address state machine.

Transitions are induced by packet arrivals (№ 4, № 8, № 10), user request for address resolution (№ 1), user request to add a permanent entry and timeouts (ARP entry expiration timeout (№ 5, № 9), packet retransmit timeout (№ 2, № 3, № 6, № 7)). Every of these four events is implemented with a separate C function: RecvArpMessage, ResolveIp, AddPermanentIp, HandleTimeouts. To control WCET of HandleTimeouts system integrator has configuration parameter max_handled_timeouts that control maximum number of timeouts handled during one function call.

3.5 Top-level UDP/IP stack architecture

To study the UDP/IP network stack architecture let's examine packet transmit path (Fig. 8). To transmit a packet user first requests a buffer from network interface (an interface provided by UDP/IP stack). As was discussed earlier, this buffer is no longer allocated in the device driver and is taken from corresponding buffer pool. Then network user forms packet payload and passes packet buffer through the network interface. At this stage UDP/IP packet headers are also formed. Network stack tries to resolve MAC address as early as possible, because sometimes request to ResolveIp function may result in ARP request traversing the network.

In order to provide delivery guarantees to the end user we perform bandwidth accounting as it is done in AFDX [3]. On the logical level we see the network as a set of virtual links. Each virtual link is a route from one source UDP/IP address to one destination UDP/IP address. Each virtual link has a statically assigned worst-case bandwidth (bytes/s) governed by credit-based traffic shaper [24]. This allows to estimate and control data flow through each path in the network. Therefore, each packet is assigned it to a virtual link and added to VL packet queue. If there is enough budget to transmit a packet it is forwarded to fragmentation module. Transmission of packets not associated with any virtual links is possible but discouraged in hard real-time scenarios.

Fragmentation module implements IPv4 fragmentation and packet reassembly. Its main design challenge is that driver queue can fill during the process of fragmentation. Therefore, fragmentation process must be interruptible and therefore must be implemented as a state machine and not as a sequential algorithm.

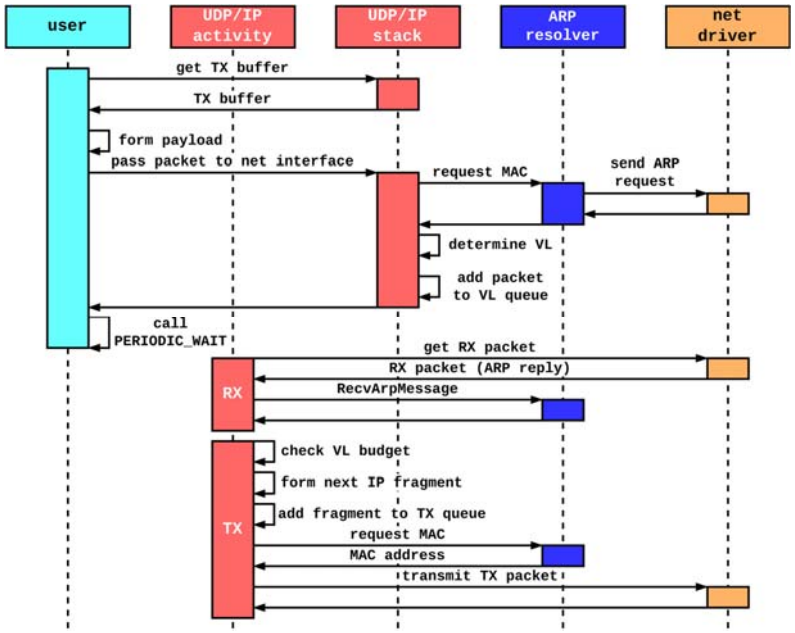

Fig. 8. UDP/IP network stack transmit path.

Eventually, network stack forms IP fragments and puts them into the device transmit queue. At that stage address resolution should finish. Otherwise, fragments will block waiting for address resolution and eventually will be dropped by a dedicated timeout (a sort of deadlock prevention). At the end packets are retrieved from device transmit queue and passed to the device driver.

4. Test system architecture

The test system manages overall testing process. It automatically deploys test environment, controls test execution, returns a verdict – either «PASS» or «FAIL». In the simplest setup, the verdict is determined by reading the board's console port output. However, network-oriented tests introduce the network interface as an additional I/O channel. This means the test system must either support direct connection of network devices or deploy a network that allows multiple hardware instances to communicate.

4.1 Test environment

Test node consists of a software component under test and hardware platform that runs it. In our case software component is a RTOS with a network stack. Test node can either be a hardware node or a node emulated by QEMU.

Most CLOS applications are abstracted from hardware platform and can be run both on hardware and in emulator. However, from the test system's perspective, a hardware test node and an emulated one are treated differently. Real hardware requires physical means of packet transmission like an Ethernet cable, whereas emulated hardware does not. Tests that work only with emulated test nodes run locally on the developer's computer. Real hardware, in contrast, is pooled together in a centralized hardware-software testbed that is accessed by developers over the network. Developers compete for exclusive access to individual hardware instances.

Test environment can include a mock model. Mock model simulates test node behavior and participates in the network with the same rights as a test node. Unlike test node, mock model is never used as test subject. It provides full support for various network protocols – Ethernet, UDP, IP, ARP, etc. – and ensures that protocol-level behavior is correct. Test developer implements a mock model much faster than a test node. We designed three mock model types suitable for writing tests of varying complexity.

The simplest form of a mock model is an echo server: it receives a packet from a test node and returns it unmodified, effectively acting as a loopback. It is used for basic usability check as an assistance during network driver development.

A more sophisticated mock model is implemented using netcat utility. Packet exchange scenario for each test is configured as YAML file and has commands: «transmit packet X», «receive packet Y», «wait for M milliseconds». Test system executes this scenario and issues a verdict. This type of mock model is used for thorough network driver or NIC hardware testing.

The ultimate mock model is a Python script. It uses a library of predefined functions to transmit and receive packets. This approach to mock model construction uses a full range of Python language facilities and is superior to other approaches in terms of expressiveness. It allows to implement stateful protocol logic and perform controlled fault injection.

4.2 Test suite configurations

Our test system supports arbitrary combinations of participants. Below are some common configurations and their typical use cases:

- **One test node and one mock model**. This is the most basic setup, used to test a single node, where the mock model serves as a component of the test environment.
- **Two test nodes**. This configuration is necessary when capabilities provided by a mock model are insufficient (for instance, testing of Ethernet checksumming or MAC-based filtering).
- **Multiple test nodes and mock-models**. This configuration is required for system-level testing and verification of complicated distributed algorithms.

4.3 Network organization

Test nodes and mock models communicate over an emulated or physical network. However, network setup is non-trivial and depends on test configuration. Let's review the approaches we use for network organization:

- **Physical Ethernet network**. We have a physical LAN that connects all hardware platforms. It is used when only hardware-based test nodes are employed.
- **Hardware-to-Linux-server communication.** When a hardware-based test node needs to send packets to a mock model (or to an emulated test node), it can forward them to a Linux server residing on the same network. The Linux server forwards packets to the mock model. This approach does not support multiple mock models and emulated test nodes, because Linux server is connected in LAN with only one interface. It means that packets sent to a single MAC address need to be delivered to different network nodes.
- **QEMU sockets in multicast mode.** QEMU documentation [25] recommends this mode for setting up a network of multiple QEMU instances. Several QEMU instances forward the I/O emulated onboard devices to UDP sockets operating in multicast mode. As a result, any packet sent by one test node is delivered to all the others. The main drawback of this method is that multicast traffic propagates beyond the developer's machine across the wider network. This can lead to an incident in which two machines in one LAN inadvertently intercept each other's multicast traffic.

- **Python router.** A Python utility emulates the behavior of an Ethernet switch. This utility interconnects all emulated test nodes and all mock models.
- **Python sniffer.** A Python utility bridges the hardware and the emulators. It runs on a Linux server located within the network of hardware test nodes. It captures packets from the physical network and forwards them to the python-router utility. Conversely, it receives packets from python-router and injects them into the physical network. A notable drawback of this utility is the complexity of its configuration. For example, it must capture broadcast packets that belong to the current test session, while filtering out other broadcast traffic.

This set of network emulation methods supports all possible combinations of test nodes and mock models (Fig. 9). In theory, it enables full-system testing. However, this network architecture does not provide deterministic network delays as the execution time of the Python utilities cannot be reliably estimated.

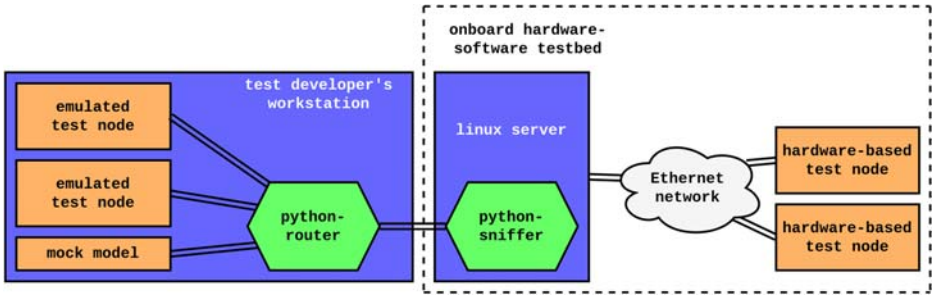

Fig. 9. Preferred network organization scheme.

4.4 Configuration management

A test node is associated with a set of configuration parameters, such as its own MAC address and MAC addresses of other test nodes. The simplest way to manage these parameters is to hard-code them in the program source code. In our case, these parameters are generated by the test system and passed to the test node as boot parameters. This way test developer no longer needs to manually select MAC addresses. Therefore, multiple concurrently running test scenarios no longer fail due to misconfigured parameters.

5. Results

To demonstrate functionality of our network stack we constructed a simple test consisting of two test nodes. For simplicity, both test nodes have only one ARINC 653 partition that contains both user code and UDP/IP network stack. First test node transmits 100 packets with size of 4096 bytes and with content of «Hello X», second node receives them from the network stack. Route from first node to the second is a virtual linked and we assign maximum allowed bandwidth to it (either 4000 bytes/s or 8000 bytes/s).

Our UDP/IP network stack performs address resolution using ARP and transmits packets. As packet size of 4096 exceeds MTU, IP fragmentation and reassembly is involved. Inter-packet gap is 1.11s for 4000 bytes/s (Fig. 10) and 0.86s for 8000 bytes/s (Fig. 11) and therefore traffic shaping works.

6. Conclusion

In this paper we presented UDP/IP network stack architecture for an ARINC 653 RTOS. We also suggested test system architecture to verify network stack implementation as well as underlying network driver stack.

	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	RealtekU_00:00:01	Broadcast	ARP	42	ARP Announcement for 192.168.0.101
2	0.084028418	RealtekU_00:00:01	Broadcast	ARP	42	Who has 192.168.0.102? (ARP Probe)
3	0.135492764	RealtekU_00:00:02	RealtekU_00:00:01	ARP	42	192.168.0.102 is at 52:54:00:00:00:02
4	1.159362996	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
5	1.159776703	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
6	1.159801111	192.168.0.101	192.168.0.102	UDP	1132	10001 → 10002 Len=4050
7	2.269382814	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
8	2.269422204	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
9	2.269433526	192.168.0.101	192.168.0.102	UDP	1132	10001 → 10002 Len=4050
10	3.380272536	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
11	3.380311658	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
12	3.380555694	192.168.0.101	192.168.0.102	UDP	1132	10001 → 10002 Len=4050
13	4.482811906	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
14	4.482854656	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
15	4.482864843	192.168.0.101	192.168.0.102	UDP	1132	10001 → 10002 Len=4050
16	5.589242949	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
17	5.589280559	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
18	5.589641666	192.168.0.101	192.168.0.102	UDP	1132	10001 → 10002 Len=4050
19	6.698568241	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
20	6.698610297	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
21	6.698618348	192.168.0.101	192.168.0.102	UDP	1132	10001 → 10002 Len=4050
22	7.805210611	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
23	7.805247989	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
24	7.805255440	192.168.0.101	192.168.0.102	UDP	1132	10001 → 10002 Len=4050
25	8.922598967	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
26	8.922638418	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
27	8.922936556	192.168.0.101	192.168.0.102	UDP	1132	10001 → 10002 Len=4050

Fig. 10. Traffic formed by UDP/IP network stack for 4000 bytes/s.

	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	RealtekU_00:00:01	Broadcast	ARP	42	ARP Announcement for 192.168.0.101
2	0.083716125	RealtekU_00:00:01	Broadcast	ARP	42	Who has 192.168.0.102? (ARP Probe)
3	0.100938331	RealtekU_00:00:02	RealtekU_00:00:01	ARP	42	192.168.0.102 is at 52:54:00:00:00:02
4	0.576577179	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
5	0.576958503	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
6	0.576979883	192.168.0.101	192.168.0.102	UDP	1132	10001 → 10002 Len=4050
7	1.160640805	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
8	1.160678294	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
9	1.161000363	192.168.0.101	192.168.0.102	UDP	1132	10001 → 10002 Len=4050
10	1.741371208	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
11	1.741413913	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
12	1.741423727	192.168.0.101	192.168.0.102	UDP	1132	10001 → 10002 Len=4050
13	2.323380367	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
14	2.323415808	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
15	2.323423027	192.168.0.101	192.168.0.102	UDP	1132	10001 → 10002 Len=4050
16	2.904794698	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
17	2.904830166	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
18	2.904838471	192.168.0.101	192.168.0.102	UDP	1132	10001 → 10002 Len=4050
19	3.487370497	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
20	3.487406255	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
21	3.487414046	192.168.0.101	192.168.0.102	UDP	1132	10001 → 10002 Len=4050
22	4.067908945	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
23	4.067933169	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
24	4.067937443	192.168.0.101	192.168.0.102	UDP	1132	10001 → 10002 Len=4050
25	4.648663127	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
26	4.648698191	192.168.0.101	192.168.0.102	IPv4	1514	Fragmented IP protocol (proto=UDP 17,
27	4.649009240	192.168.0.101	192.168.0.102	UDP	1132	10001 → 10002 Len=4050

Fig. 11. Traffic formed by UDP/IP network stack for 8000 bytes/s.

UDP/IP networking is more than 50 years old, yet to construct a UDP/IP stack suitable for hard real-time applications we had to reinvent approaches for packet buffer allocation, queuing, timeout management, and address resolution. In order to be able to test this with airborne equipment we also had to construct our own sophisticated test system.

We hope our experience can be applied during design, implementation or testing of other real-time network stacks.

The developed architecture is implemented in CLOS real-time operating system, developed at ISP RAS, and tested on available hardware used in aerospace industry.

References

[1]. Review and Rationale of MIL-STD-1553A and MIL-STD-1553B, 2012. Available at: https://www.milstd1553.com/wp-content/uploads/2012/12/MIL-STD-1553B.pdf, accessed: 14.08.2026.
[2]. SpaceWire – Links, nodes, routers, and networks, Std. ECSS-E-ST-50-12C Rev.1, May 2019.
[3]. Aircraft Data Network Part 7 – Avionics Full Duplex Switched Ethernet (AFDX) Network, Aeronautical Radio, Inc. ARINC Specification 664P7, Jun. 2005.
[4]. Postel, J., 1980. User datagram protocol, RFC 768.
[5]. Postel, J., 1981. Internet Protocol – DARPA Internet Program Protocol Specification, RFC 791.
[6]. Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications, IEEE Std. 802.3, 2002.

[7]. Avionics Application Software Standard Interface Part 1 – Required Services, Aeronautical Radio, Inc. ARINC Specification 653P1-5, Dec. 2024.
[8]. Cheptsov, V. and Khoroshilov, A., 2023, December. Robust resource partitioning approach for arinc 653 rtos. In 2023 Ivannikov ISPRASOPEN Conference (ISPRAS), IEEE, pp. 33-39.
[9]. Airlines Electronic Engineering Committee, Software Data Loader using Ethernet Interface, ARINC Report 615A-3, Aeronautical Radio, Inc. 2551 Riva Road, Annapolis, Maryland 21401-7435, 2007.
[10]. RTCA DO-178C, Software Considerations in Airborne Systems and Equipment Certification, December 2011.
[11]. Dunkels, A. Design and Implementation of the lwIP TCP/IP Stack. Swedish Institute of Computer Science, 2001, 2(77).
[12]. Бурдонов И.Б., Косачев А.С., Петренко А.К., Хорошилов А.В., Чепцов В.Ю. Семейство операционных систем КЛОС. Труды ИСП РАН, том 37, вып. 6, часть 4, 2025 г., стр. 11-26. DOI: 10.15514/ISPRAS–2025–37(6)-47. / Burdonov I.B., Kossatchev A.S., Petrenko A.K., Khoroshilov A.V., Cheptsov V.Yu. CLOS Operating System Family. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 4, 2025., pp. 11-26 (in Russian). DOI: 10.15514/ISPRAS–2025–37(6)-47.
[13]. Mallachiev K.M., Pakulin N.V., Khoroshilov A.V. Design and architecture of real-time operating system. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 2, 2016, pp. 181-192. DOI: 10.15514/ISPRAS-2016-28(2)-12.
[14]. Gomes, B., Silveira, D., Gouveia, L. and Mendes, L., 2019, February. Air hypervisor using RTEMS SMP. In European Workshop on On-Board Data Processing (OBDP2019), ESTEC (ESA).
[15]. Stephan, A. and Wüstrich, L. The path of a packet through the linux kernel. In Seminar Innovative Internet Technologies and Mobile Communications (IITM), Winter Semester, 2023.
[16]. Plummer, D. An ethernet address resolution protocol: or converting network protocol addresses to 48. bit ethernet address for transmission on ethernet hardware, 1982, no. rfc826.
[17]. Bonwick, J. The slab allocator: An object-caching kernel memory allocator. In USENIX summer, 1994, vol. 16.
[18]. Alan Cox, Kernel Korner: Network Buffers and Memory Management, The Linux Journal, 1996, issue 30.
[19]. Cardwell, N., Cheng, Y., Brakmo, L., Mathis, M., Raghavan, B., Dukkipati, N., Chu, H.K.J., Terzis, A. and Herbert, T. packetdrill: Scriptable network stack testing, from sockets to packets. In 2013 USENIX Annual Technical Conference (USENIX ATC 13), 2013, pp. 213-218.
[20]. AWS IoT Device Tester for FreeRTOS. Available at: https://docs.aws.amazon.com/freertos/latest/userguide/device-tester-for-freertos-ug.html, accessed: 14.08.2026.
[21]. Cho K. A Reconfigurable Integration Test and Simulation Bed for Engagement Control Using Virtualization. Journal of Korean Institute of Military Science and Technology, 2023, vol. 26, no. 1, p. 91-101.
[22]. Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 1990. Introduction to algorithms *(3rd ed.)*. Cambridge: MIT press.
[23]. Адельсон-Вельский Г.М., Ландис Е.М. Один алгоритм организации информации. Доклады Академии Наук, Российская академия наук, 1962, vol. 146, no. 2, pp. 263-266.
[24]. Institute of Electrical and Electronics Engineers, IEEE 802.1Qav IEEE Standard for Local and metropolitan area networks. Virtual Bridged Local Area Networks. Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams, IEEE, Standard IEEE 802.1Qav-2009, Dec. 2009.
[25]. QEMU online documentation. Available at: https://www.qemu.org/docs/master/system/invocation.html, accessed 14.08.2026.

Информация об авторах / Information about authors

Владислав Владимирович АЛЕЙНИК – сотрудник отдела технологий программирования ИСП РАН; аспирант МФТИ. Сфера научных интересов: операционные системы реального времени, архитектура сетевого стека, высокоскоростные бортовые компьютерные сети.

Vladislav Vladimirovich ALEINIK – employee of the department of Programming Technologies of the Ivannikov Institute for System Programming of the Russian Academy of Sciences; postgraduate student at Moscow Institute of Physics and Technology. Research interests: real-time operating systems, network stack architecture, real-time onboard computer networks.

Дмитрий Сергеевич ФОМИН – сотрудник отдела технологий программирования ИСП РАН; бакалавр ВМК МГУ. Сфера научных интересов: операционные системы реального времени, распределённые системы, тестирование сетевых стеков.

Dmitriy Sergeevich FOMIN – employee of the department of Programming Technologies of the Ivannikov Institute for System Programming of the Russian Academy of Sciences; graduate bachelor student at Moscow State University. Research interests: real-time operating systems, distributed systems, network stack verification.

Виталий Юрьевич ЧЕПЦОВ – архитектор операционных систем в ИСП РАН. Основные научные интересы: встраиваемыми системы ответственного назначения, системы с жёстким реальным временем для бортовой аппаратуры, безопасность Unix-совместимых ОС общего назначения, загрузочное программное обеспечение UEFI.

Vitaliy Yurievich CHEPTSOV – OS architect at ISP RAS. Main research interests: safety-critical embedded systems, hard real-time systems for airborne equipment, security of general-purpose Unix-like operating systems, UEFI boot software.