Paper

Design and Development of a UDP-Based Connection-Oriented Multi-Stream One-to-Many Communication Protocol

Valentin Stanciu, Mugurel Ionuţ Andreica, Vlad Olaru, and Nicolae Ţăpuş

Politehnica University of Bucharest, Bucharest, Romania

Abstract—A communication protocol is a set of rules defined formally that describes the format of digital messages and the rules for exchanging those messages in or between computing systems. The Internet Protocol Suite used for communications throughout the Internet uses encapsulation to provide a way of abstracting protocols and services. This abstraction is grouped into layers of general functionality. For protocols on the transmission layer, many choices exist. But while popular protocols such as TCP, UDP and SCTP do provide connection oriented communication offering reliability, ordering and data integrity, solutions that offer such connections from one point to multiple endpoints are still limited. TCP only supports point-to-point communication and SCTP offers multi-homing functionality, but the transmission is still limited to two logical endpoints. In this paper we use the simple, stateless, transmission model of UDP in order to provide TCP-like services for one-to-many communication that is not limited to just multi-homing or other particular solutions. The protocol supports reliable communication from one endpoint to multiple endpoints in different transmission modes. In order to make it easier for developers to customize the protocol to their needs and possibly extend/modify it in order to create new variants from it, the protocol is developed in user space. Because of this design restriction performance wasn't the main objective of our work, but rather the ease of customization and experimentation with new protocol variants. The protocol was implemented in the C++ programming language using classes with virtual members. New variants of components, such as packet retransmission, can easily be implemented without changing the whole code base.

Keywords—communication protocol, connection oriented, multiple streams, one-to-many.

1. Introduction

The work presented in this paper consists of the design and development of a network protocol for reliable communication from one point to multiple points, on possible multiple machines.

The main motivation behind this work is to provide enhanced communication functionality from one endpoint to multiple endpoints. It should be noted that similar transport layer protocols such as TCP and SCTP don't provide the kind of functionality that this protocol provides. SCTP's multi homing support only deals with communication between two endpoints which are assigned multiple IP addresses on possibly multiple network interfaces; it does not deal with configurations that contain multiple endpoints (for example, clustered endpoints). Our work allows an application running on a machine to connect to a collection of machines as if they were a single one. It practically virtualizes a set of machines under the same endpoint, each machine being accessible under many streams. Using this approach, one can implement features such as load balancing, which is absent in SCTP.

Similar to SCTP, our protocol supports multiple streams inside each connection. This is an improvement to TCP's single-stream connections, as using multiple streams has the advantage of better parallelization that leads to better performance in the context of today's multi-core processors.

Using the one-to-many facilities and the support for multiple streams, new programing models for network connections and new design patterns can be created. This allows for easier application implementations and shorter development times for advanced functionality.

The common way of developing a networking protocol is to implement parts of it in kernel space. This not only splits the implementation into two parts (kernel space and user space), but also makes its configuration, tweaking and portability to multiple operating systems more difficult. To alleviate such problems, we decided to implement the protocol only in user space. Even if performance may be affected because of this decision, it is beyond the scope of this paper to provide fast absolute performance. The protocol uses the UDP transport protocol, on top of which it implements the desired connection-oriented functionality.

The rest of this paper is organized as follows. In Section 2 we discuss related work. In Section 3 we present the design of our one-to-many communication protocol and in Section 4 we provide implementation details. In Section 5 we present experimental results. Finally, in Section 6 we conclude and discuss future work.

2. Related Work

2.1. Transmission Control Protocol (TCP)

One of the core protocols of the Internet Protocol Suite is the transport layer protocol name Transmission Control Protocol (TCP) [1]. Complementing the Internet Protocol (IP), it is one of the two original components of the Internet Protocol Suite, and the reason the entire suite is commonly referred to as TCP/IP. TCP's design is for use as a highly reliable host-to-host protocol between hosts in computer communication networks.
TCP provides reliable inter-process communication between pairs of processes in host computers attached to distinct but interconnected computer communication networks. Very few assumptions are made as to the reliability of the communication protocols below the TCP layer.
Because of the wide spread adoption of TCP and because the facilities offered by it are well known and have been tested thoroughly over the years, a comparison with TCP is inevitable. Our protocol implements the features offered by TCP and extends them to one-to-many connections.

2.2. User Datagram Protocol (UDP)

User Datagram Protocol (UDP) [2] is one of the most commonly used protocols of the Internet Protocol Suite. UDP's simple transmission model without implicit connectivity provides a fast way to transmit data. One of the features of UDP is multicast addressing. Multicast is the delivery of a message or information to a group of destination computers simultaneously, in a single transmission from the source. The problem is that this form of communication is not reliable and messages may be lost or delivered out of order. Moreover, the multicast facility requires support from the underlying network devices, which is rarely available.
Our protocol is built over UDP because of its simple, performance-oriented design and offers the functionality of multicast with the reliability of TCP, as well as other forms of one-to-many addressing. It is worth mentioning, though, that in UDP's multicast model, the sender does need to know all the receivers (but only a group identifier), while our multicast model requires explicit knowledge of each receiver's network address.

2.3. Stream Control Transmission Protocol (SCTP)

The Stream Control Transmission Protocol (SCTP) [3] is a new protocol existing at an equivalent level with UDP and TCP on the protocol stack and provides transport layer functions to many Internet applications. SCTP has been approved by the IETF as a proposed standard in 2000 and updated over the years. SCTP is a reliable transport protocol operating on top of a connectionless packet network such as IP. It offers the following services to its users:

- acknowledged error-free non-duplicated transfer of user data,
- data fragmentation to conform to discovered path MTU size,
- sequenced delivery of user messages within multiple streams,
- an option for order-of-arrival delivery of individual user messages,
- optional bundling of multiple user messages into a single SCTP packet,
- network-level fault tolerance through supporting of multi-homing at either or both ends of an association.

Similar to TCP, SCTP offers a reliable transport service. SCTP makes sure that data is transported across the network without errors even if packet loss is possible and that the data arrives in the correct sequence. Similar still to TCP, SCTP creates a relationship between endpoints prior to data being transferred. This relationship denotes what is called a "session-oriented" mechanism for transmitting data. In SCTP terminology, such a relationship is called an association and is maintained until all data has been successfully transmitted.
SCTP improves upon the TCP design by adding support for message-based, multi-streaming, multi-homing delivery of chunks without head-of-line blocking, path selection and monitoring, validation and acknowledgment with protection against flooding and improved error detection.
Unlike TCP which is byte-oriented, SCTP is message oriented and supports framing of individual message boundaries. Data is sent as being part of a message and sent on a stream in the form of a packet. Error detection and correction is also resolved at the message level.

2.4. Other Communication Protocols and Techniques

Some of the underlying principles used in the design of our protocol were first mentioned (as concepts) in [4]. In [5] a (multi)point-to-(multi)point communication protocol was proposed which uses delay for congestion control, rather than a congestion window (like TCP, SCTP and our protocol). In [6] a one-to-many communication method based on constructing an application-aware overlay was proposed. This differs from our approach, in the sense that an overlay needs to be constructed and maintained. In [7] the authors maintain the idea of constructing an overlay for multicast communication, but this overlay is hidden "under" an "overlay socket". In the sense of introducing new types of sockets, this approach is similar to ours.
In regard to the user space implementation of our communication protocol, many previously proposed protocols were also implemented in user space [5] [6], [7]. Recently, an Internet draft proposal [8] was published for encapsulating SCTP packets in UDP packets (i.e., implementing SCTP in user space over UDP), in order to address SCTP's lack of kernel-level implementation availability and NAT traversal issues.

3. Protocol Design

There are many design paths to take when developing a user space networking protocol. The first option considered was implementing the protocol via overloading existing functions and using callback mechanisms. However, this makes the code hard to follow and even harder to extend or tweak particular functionalities.

To make the implementation of a communication protocol easier and to have better defined functions for certain tasks, the protocol needs an entity that would always run and send/receive messages. This can be either a separate process or a separate thread. A separate process would require intensive IPC (Inter-Process Communication) and synchronization, while a separate thread would mean every application that uses the protocol will each have its own thread that performs communication.

Because of the separate logic for sending and receiving data, it was decided that there would be 2 execution units for communication. Only one unit would mean that certain bottlenecks may occur when both receiving and transmitting large chunks of data. This case also increases in frequency since for every packet sent, an ACK packet may be required to be received.

Furthermore, because of the nature of networking protocols, some packets will be lost and a good protocol needs a way to handle such cases. In order to provide this functionality, a timeout mechanism is required. The mechanism needs to be as fine grained as the operating system permits and not block the client application that uses the protocol or one of the transmission execution units. Thus, another execution unit is needed with the sole purpose of providing timeout functionality.

In Linux, a socket descriptor is just a number that has meaning only to the process who owns it. The user space side has just a number and the kernel maps the pair *(number, process)* to a particular kernel socket. If a process transfers the socket number to another process, the socket descriptor becomes invalid as the kernel doesn't know the socket descriptor got copied to the new process. To copy a socket descriptor between processes, Unix sockets have to be used and the descriptor copied through them. However, this form of socket management quickly proves to be overly complicated to synchronize. When one process modifies the socket attributes, it has to announce the other processes that have a copy of the same socket in other to maintain consistency.

Because implementing the separate execution units as processes requires more advanced and tangled synchronization and communication, the execution units were implemented as threads. The total memory usage of the applications using the protocol will increase as each one will have a copy of the 3 execution units used for transmission, but networking speeds will not necessarily be impacted. The kernel resolves the problem of multiplexing different communications and *N* processes transmitting data may be even faster than just one process transmitting data.

The only downside of this decision is the inability to implement QoS (Quality of Service) – like functionality in the protocol. These services require knowledge of all (or as much as possible) data transmitted from a machine so even if every application using our protocol used a common engine for transmission, applications using other protocols like TCP would interfere with QoS.

3.1. Protocol Operation

The primary purpose of the protocol is to provide a reliable logical circuit or connection service between one to many endpoints. To provide this service on top of a less reliable communication system the following facilities are required:

Basic data transfer

The protocol packs some number of octets into packets for transmission through the networking system. This way, the protocol is able to transfer a stream of octets in each direction grouped into messages.

Reliability

In case data is damaged, lost, duplicated or delivered out of order, the protocol must recover and never enter an unknown state. This is achieved by assigning a sequence number to each packet transmitted and requiring an acknowledgment (ACK) from the receiving end. If the ACK is not received within a timeout interval, the data is retransmitted. The sequence numbers are also used by the receiver to correctly order packets that may be received out of order or to eliminate duplicates. Damage is handled by having a checksum the end of the packet header, calculated by the sender and checked by the receiver, who discards damaged packets.

Multiplexing

To allow for many execution units within a single host to use the communication facilities simultaneously, the protocol provides a set of streams within each process, further developing on the address and port identification elements.

The protocol must allow for multiple distinct communications to take place on the same machine or by the same process, but each one must use a different source endpoint.

Connections

To obtain the reliability and flow control mechanisms described above, the protocol initializes and maintains certain status information for each stream. Multiple streams with the same address and port source form a connection. Each connection is uniquely specified by endpoint addresses, ports and streams.

The protocol must first establish a connection (initialize the status information on each side) before two processes

can communicate and when that communication is complete, the connection is closed and the used resources freed. In order for the connections to be established over unreliable communication systems, a handshake mechanism with timeout-based sequence numbers is used to avoid erroneous initialization of connections.

Multiple endpoints

Similar protocols only implement reliable communication between two endpoints or unreliable communication from one to many.

The protocol provides a way to communicate with multiple hosts at the same time while offering the facilities of a point-to-point connection oriented communication.

In this form of communication, a retransmission model must be chosen. Either one or all of the peers must acknowledge packets so retransmission will not occur.

Congestion control

Congestion control is implemented the same way as for TCP, by using a congestion window representing the maximum number of packets (or of bytes) sent but not yet acknowledged. The difference from a standard point-to-point protocol is the definition of a packet being acknowledged (e.g., depending on the transmission mode, a packet is acknowledged if one, some or all of the destinations acknowledge the packet).

3.2. Sequence Numbers

One of the main concepts in the protocol design is that a sequence number is assigned to every packet sent over a connection. This is similar to every other protocol that provides reliable communication. Because every packet has a sequence number, each and every one of them can be acknowledged and this allows for detection of duplicates or lost packets. Every stream of the communication has a sequence number for each direction.
It is essential to remember that the actual sequence number space is finite, from 0 to $2^{32}-1$. Since the space is finite, all arithmetic dealing with sequence numbers must be performed modulo 2^{32}. If this protocol is extended, a sliding window can never be larger than 2^{31}.
Using sequence numbers, the protocol must perform the following operations:

- determine that an incoming packet contains a sequence number that is expected,
- determine that an acknowledgement number refers to a sent packet with a sequence number not yet acknowledged.

When an acknowledgement has been received for all sent sequence numbers, the protocol can conclude that all packets have been sent successfully.

3.3. Automatic Repeat Request

Automatic repeat request (ARQ), also known as *Automatic Repeat Query*, is an error-control method for data transmission that uses acknowledgements and timeouts to achieve reliable data transmission over an unreliable service. If the sender does not receive an acknowledgment before the timeout, it usually retransmits the packet until the sender receives an acknowledgment or exceeds a predefined number of retransmissions.
Our protocol uses a variant of the *Go-Back-N* ARQ for transmission. In *Go-Back-N*, the sending process continues to send a number of packets specified by a window size even without receiving an acknowledgement packet from the receiver. It can be viewed as a particular case of a sliding window protocol with the transmit window size of N and the receive window size of 1.
If the sender sends all the packets in its window and receives no ACK or only future or past ones, but not the expected one, it will go back to the sequence number of the last valid ACK it received from the receiver, fill its window starting with that frame and continue the process over again. Only the expected ACK will advance the expected sequence number. Past "duplicate" ACKs or future "out-of-order" ACKs will be ignored.
Go-Back-N ARQ is more efficient than Stop-and-wait ARQ since unlike waiting for an acknowledgement for each packet, the connection is still being utilized as packets are being sent. In other words, during the time that would otherwise be spent waiting, more packets are being sent. This method also results in one frame possibly being sent more than once and the receiver has to be aware of duplicates and discard them.
If the highest possible throughput is desired, it is important to force the transmitter not to stop sending data earlier than one round-trip delay time (RTT) because of the limits of the sliding window protocol. In order for the protocol not to limit the effective bandwidth of the link, the limit on the amount of data that it can send before stopping to wait for an acknowledgement should be larger than the bandwidth-delay product of the communication link.
The protocol can be configured in the initialization stages on how to handle retransmission in a multiple endpoint communication. In the *broadcast mode*, all the peers must acknowledge the sent packets before transmission can move forward. In the *any-cast mode*, at least one peer must acknowledge the packet. In both of these modes, each packet is actually sent to all the endpoints. In the more general case, at least P destinations must acknowledge the packet in order to consider it fully acknowldeged at the source.
We also implemented a mechanism in which a packet must be sent to at least a number K of the destinations ($K \leq$ the number of destinations). In this case, a packet may be considered acknowledged either if all, at least one or, more generally, at least $P \leq K$, of the destinations to which it was sent acknowledge it.
When we do not need to send each packet to each destination, we also implemented a load balancing mechanism, in

which the (K) destinations for the packet are selected based on a load metric computed at the source for each destination (e.g., a combination of average response time for the packets sent to it during the last few seconds/minutes and the number of lost packets sent to it in the same time interval; the response time is defined as the time difference between the moment when the packet was sent and the moment when its corresponding ACK was received).
Note that when a packet is sent to multiple destinations, each copy of the packet is handled by the sender as a separate, different packet for timeout and retransmission purposes.

3.4. Header Format

As with any protocol implemented over the network stack, each layer encapsulates the ones above. In computer networking, encapsulation is a method of designing modular communication protocols in which logically separate functions in the network are abstracted from their underlying structures by inclusion or information hiding within higher level objects. The UDP protocol encapsulates our protocol header and adds destination and source addresses and destination and source port numbers. A description of the contents of the header follows:

- Version (8 bits): The version field indicates the format of the protocol header.
- Source stream (8 bits): The source stream number.
- Destination stream (8 bits): The destination port number.
- Flags (8 bits): Flags that indicate if the packet starts or ends a message or if it is the first packet sent on this connection.
- Sequence number (32 bits): The sequence number of the first data byte in this segment. If this is the first message sent, this number is the initial sequence number and the first data byte is this number plus one.
- Acknowledgment number (32 bits): The value of the next sequence number the sender of the segment is expecting to receive.
- Data payload size (32 bits): Size of the data contained in the packet.
- Cyclic redundancy check (32 bits): The checksum field is the 32 bit one's complement of the one's complement sum of all 32 bit words in the header.
- Payload data (variable length): The actual data the application wants to transmit.

3.5. Connection Open

When opening a connection, for each stream of this connection a "three-way handshake" procedure similar to TCP's

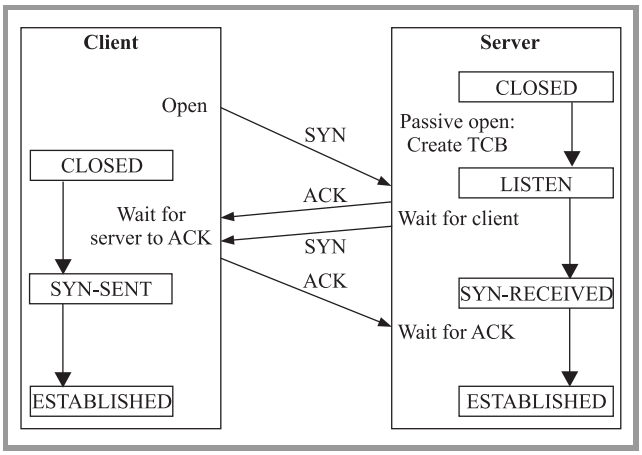


Fig. 1. Connection open.

Connection Opener is used. This procedure is normally initiated by the starting point and responded by the endpoint. In a multiple endpoints environment, each stream has its own independent opening. The three-way handshake reduces the possibility of false connections. It is a trade-off between memory and messages to provide information for this checking. The three-way handshaking works as follows:

1. A SYN (Synchronize) segment (as indicated by the bit flag) containing a 32-bit sequence number A called the Initial Send Sequence (ISS) is chosen by, and sent from, the starting point (Host 1). This 32-bit sequence number A is the starting sequence number of the data in the packet and is incremented by 1 for every byte of data sent within the segment. The SYN segment also places the value A+1 in the first byte of the data.

2. Host 2 (the destination) receives the SYN with the sequence number A and sends a SYN segment with its own totally independent ISS number B in the sequence number field. In addition, it sends an increment on the sequence number of the last received segment in its acknowledgment field. This stage is often called the SYN-ACK. It is here that the MSS is agreed.

3. Host 1 receives this SYN-ACK segment and sends an ACK segment containing the next sequence number. This is called Forward Acknowledgement and is received by Host 2. The ACK segment is identified by the fact that the ACK field is set.

Protocol peers must not only keep track of their own initiated sequence numbers but also those acknowledgement numbers of their peers. When connecting to multiple hosts, a similar procedure is followed for each endpoint and the starting host has a separate sequence number and acknowledgment number for each peer. Distinction between streams is provided by the stream destination field.

Normally, our protocol has one separate stream for each destination. However, there is no problem adding more streams for some of the destinations by using the API.
Because of the security vulnerability of this procedure, the three-way handshake is planned to be changed with a four-way cookie-based handshake similar to SCTP in a future version of the protocol.
In the rest of this paper, we will define the roles of "client" and "server" as follows. The client is the node initiating the connection and the server is the one accepting a connection. Thus, in the case of our protocol, the client is the source node and each destination node acts as a server. These concepts are also used in Fig. 1 which depicts the steps of the connection opening procedure.

3.6. Connection Close

In the case of a normal connection close, each side terminates its end of the connection by sending a special packet with the FIN (finish) flag set. The packet is called a FIN message and serves as a connection termination request to the other device. The device receiving the FIN responds with an acknowledgment (ACK) and a FIN to indicate that it was received and it is ready to close.

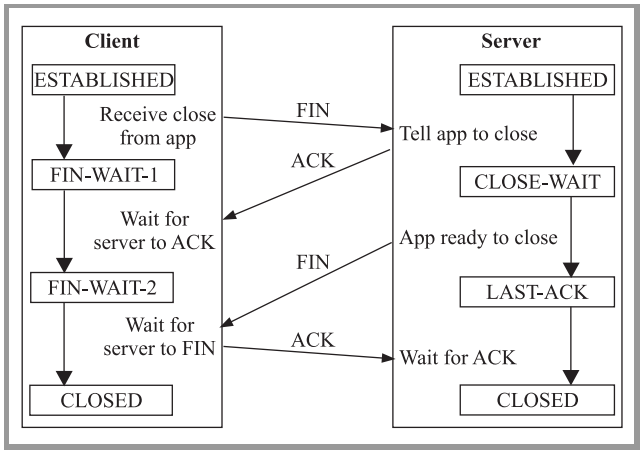


Fig. 2. Connection close.

The connection as a whole is not considered terminated until both sides have finished the shutdown procedure by sending a FIN and receiving an ACK for each stream belonging to that connection. This procedure is similar to the normal termination of a TCP connection, but repeated for every stream (see Fig. 2).

3.7. Communication System Architecture

The core of the communication system consists of three execution units: a sender, a receiver and a timer. These execution units send and receive data from the Internet and deposit them to buffers in each of the managed streams; connections can have more than one stream for transmission. Each execution unit is actually implemented as a thread pool. There are no other requirements regarding them (e.g., the thread pool may contain a fixed number of threads or it may adjust its number of threads dynamically, according to needs).
The sender unit is tasked with checking client send buffers for any data to be sent, pack it in the correct format and send it over the network. The receiver thread waits for data from the network, reads it, unpacks it and puts it into the receive buffer of the appropriate client.
When a packet is sent to the network, the sender also tells the timer to announce it after a certain period has passed. Once an ACK for the packet has been received, the timer is told to cancel that announcement. If no packet is received, the timer does notify the sending unit and that unit implements a retransmission algorithm.
From the point of view of the receiver and sender, each stream can be considered as an individual client. Each stream has its own buffers and is identified by the stream id in the header of each packet.

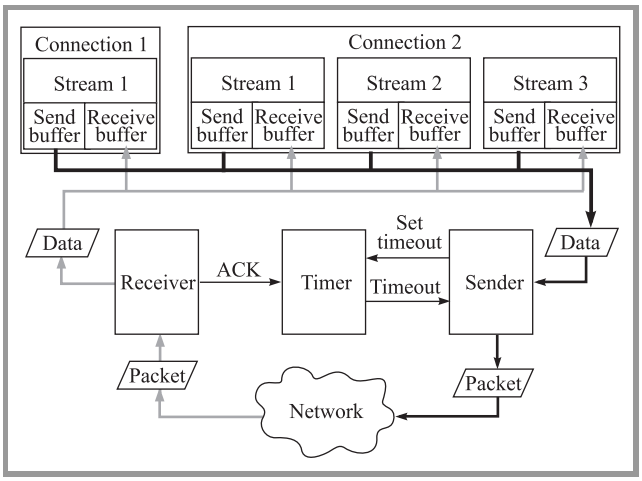


Fig. 3. Communication system architecture.

A general view of the architecture is presented in Fig. 3. Thick lines represent data being moved around. Thick grey lines show data that is received by clients while data in thick dark lines illustrate data sent by the clients. Even though in the figure lines from streams interconnect, streams have no way of seeing data from other streams.
When the application uses the API to send or receive data, it only accesses the local stream buffers. For sending, data is inserted into the send buffer (and later picked up by the sender unit). For receiving, data is fetched from the receive buffer (which was placed there by the receiver unit).
When the application tries to receive data from the network, but the receiving buffer is empty, a wait is performed on a conditional synchronized variable until the operation can be completed. The same thing happens when the application tries to send data, but the send buffer is full.
From the point of view of the receiver, if data is received from the network, but the receive buffer of the stream is full, the data is discarded and no ACK is sent. This will force the sender of the data to retransmit at a later time, when the application might have read the data and emptied the receive buffer.

4. Communication System and Protocol Implementation

The code is divided into 3 components: The API (Application Programming Interface) class, the execution units implementations and the global components. The API is implemented as a single C++ class to provide all the functionality in one place. Because the actual work of the protocol is done in an asynchronous fashion, the execution units tasked with transmission are separate from the API. These thread pools can be considered the core of the communication system and they are not part of a class or container as each thread is individual and only performs work related to itself. The only time when the core needs to interact with other entities is when exchanging data. These exchanges are governed by global synchronization mechanisms.
Besides the API and the core threads, there are also common components used by both parts. These contain both miscellaneous functionality such as CRC calculation and protocol particular structures like packet queue definition. They are called global (with respect to the protocol) because they are used by different parts of the code base and different components need access to them.
To implement this protocol extensive use of threads was required and a lot of inter-thread synchronization. To achieve this, the POSIX thread library *pthread* was used.

4.1. Linux POSIX Threads

The POSIX thread library contains a standards based thread API for C/C++. It allows one process to spawn a new concurrent execution flow. It is most effective on multiprocessor or multi-core systems where the process flow can be scheduled to run on another processor thus gaining speed through parallel or distributed processing. All threads within a process share the same address space. A thread is spawned by defining a function and its arguments which will be processed in the thread.
The threads library provides three synchronization mechanisms:

- Mutex (mutual exclusion lock) – enforces exclusive access by a thread to a variable or set of variables.
- Join – makes a thread wait until another thread is complete (terminated).
- Condition variable (data type *pthread_cond_t*) – blocks a thread's execution until a condition is met.

Mutexes are used to prevent data inconsistencies due to operations by multiple threads upon the same memory area performed at the same time or to prevent race conditions. A contention or race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depend on the order in which these operations are performed. Mutexes are used for serializing shared resources such as memory. Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it. All global variables accessed by the protocol are accessed only after first obtaining a mutex that governs them.
A join is performed when one thread wants to wait for another thread to finish. The project only uses joins when the last connection of the process is closed and the protocol threads used for transmission need to be closed. A join is used to make sure the threads finish sending/receiving all the data and then exit gracefully.
A condition variable is a variable of type *pthread_cond_t* and is used with the appropriate functions for waiting and, later, continue processing. The condition variable mechanism allows threads to suspend execution and relinquish the processor until some condition is true. A condition variable must always be associated with a mutex to avoid a race condition created by one thread preparing to wait and another thread which may signal the condition before the first thread actually waits on it, thus resulting in a deadlock. The thread will be perpetually waiting for a signal that is never sent. Any mutex can be used; there is no explicit link between the mutex and the condition variable. Condition variables are used a lot throughout the code base. Waiting for a packet to be received is done via waiting for a condition to be met. Another example is the sending thread that waits for data to be sent from any of the clients.

4.2. Application Programming Interface

To make use of the protocol, an API is provided. The API is implemented in C++ using classes with virtual members. If another developer wants to extend or tweak the functionality of the protocol (s)he can do so by extending a class and changing its virtual members. It is also designed to be similar to the POSIX standard for sending and receiving packets.

4.2.1. Init

This function is called without parameters and used for initializing local communication members (i.e., the communication system). This is the first function the application must call before it can begin opening or accepting connections. Failure to initialize the communication channel will lead to inability to send or receive data. The sending thread(s), the receiving thread(s) and the timer thread(s) are created and their ids are stored globally.

4.2.2. Stop

This function is called without parameters and signals that the communication system should be stopped. The execution units are stopped.

4.2.3. Connect

This function is called with two parameters: a vector of hostname and port pairs and connection flags. If the application uses the connection as a client, it must specify

to what hosts it will communicate. This function receives a vector of hostname and port pairs and connects to all of them, encompassing them in a single connection. A number of streams (by default 1) is opened for every endpoint and the Connection Open handshake is executed. Currently a TCP-like three way handshake is used. The connection flags explain the retransmission model to be used in case of packet loss for one of the connected peers. Possible values include:

- CANY to indicate that the sent message must reach at least one connected peer. In case the message is lost for some of them, but one peer still received it and sent an ACK, transmission will continue normally. In case no ACK is received, the packet is retransmitted for all peers.
- CALL to indicate that the sent message must reach all of the connected peers. All peers must receive the message and reply with an ACK. In case an ACK is not received from some peers, the message is retransmitted for those peers only.
- CBAL to indicate that the load balancing mechanism should be used.

Since all streams are independent, a common synchronization structure is created for the streams under the same connection. This will enable the retransmission behavior described above. A communication channel object is created and stored internally and a pointer to it is returned by the function.

4.2.4. Close

This function is called on a communication channel. When the application finished sending or receiving data on the communication channel, it has to call this function. This will enable all the transmission buffers to be emptied and close the communication channel. All the streams inside this channel from all associated endpoints will be closed. It will also close the operating system UDP socket and enable it for reuse. The function also removes the connection from the global client list.

4.2.5. WaitForClient

This function is called on a communication channel, with one parameter: port number. If the application uses the connection to act as a server, it must specify a port onto which clients will connect. This function listens for data on that port and then waits for a client to connect to it. The client and the server establish the connection via the connection open handshake. Currently a TCP-like three way handshake is used. When a client successfully connects to the server, a communication channel object is created and a pointer to it is returned.

4.2.6. Send

This function is called on a communication channel with the following parameters: pointer to message, message length and flags. The application instructs the protocol to send a message of a specific size to the connected peers. If the application is acting as a client, the message is sent to the ones specified at connect, following the retransmission model described by the connect flags. If the application is acting as a server, the message is sent to the client that connected to it in the WaitForClient method. If the message is sent as one big chunk and is lost, the protocol retransmits the message again. For large messages this will prove inefficient. Therefore, the message is first broken into chunks of a maximum size MaxData and packets are made with each chunk. Also, markers are placed for the start of the message and the end of the message in the corresponding packets to enable the reconstruction of the message when received. The flags parameter is used to specify different communication behaviors. The function returns only after all the data was placed into the send queue/buffer (a condition variable is used for waiting until room is available in the send queue/buffer).

4.2.7. WaitForSend

This function is called on a communication channel with one parameter: stream number. The send function only inserts data into the queues and returns. The actual sending of the data is achieved asynchronously. This function is used if the application desires to block until the data is sent. It waits on a synchronized condition until all the packets inside the sending queue associated with the stream have been sent and acknowledged.

4.2.8. Receive

This function is called on a communication channel with the following parameters: pointer to buffer to store the message, length of buffer, stream number and flags. The application instructs the protocol to retrieve a received message from a certain stream and store it in the buffer passed as a parameter. This function blocks until a new message is ready to be given to the application. This implies waiting until all the packets corresponding to that message have been received and stitching them back together using the start of message and end of message markers. The flags parameter is used to specify different communication behaviors.

4.2.9. AddDestination

This function is called on a communication channel in order to add a new destination to it. A new stream towards the destination is created and the connection open procedure is used. If the connection already contains this destination as its peer, the effect is that a new stream is created towards the destination.

4.2.10. RemoveDestination

This function is called on a communication channel in order to remove a destination from its peers. The stream towards that destination is closed. If there are multiple streams towards the destination, only one of them is closed.

4.2.11. FullyRemoveDestination

This function is similar to the previous one, except that all the streams towards the given destination are closed.

4.3. Internal Members

4.3.1. Send Queues

These queues act as a buffer space where data from the client application is stored before the sending threads pick them up. Each stream has its own queue and the API function Send inserts data here. There exists one sending queue for each communication channel. Moreover, there is one sending queue for each stream.

4.3.2. Receive Queues

These queues act as a buffer space where data for the client application is stored after the receiving threads get data from the network. Each stream has its own queue and data resides here until the client application decides to make an explicit read for the data using the Receive function. There exists one receive queue for each stream of a communication channel.

4.3.3. Socket Descriptor

The socket descriptor is an abstraction used by the operating system socket to transmit data. Each UDP "connection" has a socket descriptor and each comunication channel uses just one socket for a connection, regardless of how many peers it has connected to it. All communications send by the protocol for this connection pass through this socket. All the streams use the same socket. It is initialized when a connection is first opened, either via a Connect or via a WaitForClient function call. The socket will remain valid until the connection is closed via a Close call.

4.3.4. Destination Address Vector

A vector containing destination structures particular to the operating system. They describe the communication peers. It is initialized in the Connect or WaitForClient functions and modified in the AddDestination, RemoveDestination and FullyRemoveDestination functions. There exists one such vector for each communication channel.

4.3.5. Synchronization Primitives

These primitives are used for synchronization with the protocol threads that do the actual communication. In case of synchronized network communication, the client thread has to wait for the sending or receiving thread(s) to finish transmission. A synchronized network transmission always occurs in the handshake as no communication is possible until it has completed successfully. The synchronization is achieved through mutexes and condition variables. Whenever the application has to wait for data to be sent, it waits for the send condition to be fired. Whenever the receiving thread obtains an ACK for a packet and there are no more packets to be sent for that particular stream, a send condition is broadcasted. If the application wants to receive data, but there is no more data in the receive queue, it waits until a receive condition is fired. Whenever a receiving thread adds more data for a particular stream, it also broadcasts a receive condition. Each communication channel has its own set of synchronization primitives.

4.4. Global Elements

In order for the threads to know what clients (communication channels) are managed, a list of all the clients must be stored. Access to this list must also be synchronized and also flags must be raised when the process exists. All streams use a packet queue for receiving and another for sending. These queues are accessed by both the API and the transmission threads and access to them must be synchronized. Additional operating system elements must also be stored globally to be accessed from all components of the protocol.

4.4.1. Client List

A global client list needs to be maintained for the sending threads to know where to look for data to be sent and for receiving threads to know where to insert captured data from the network. This list is stored in a vector available to all the elements of the protocols and contains pointers to the protocol client classes. Whenever a new connection is initiated/accepted, the Connect/WaitForClient function call inserts a pointer to the created communication channel class here. Access to this list is synchronized via a mutex.

4.4.2. Thread Flag

The thread flag is a special global variable is provided globally and always accessed by the internal threads before any new round of operations is started. This flag is used to signal when the internal components of the protocol needs to stop. Other notifications can also be implemented in the future via this flag, for example temporarily stopping the protocol for a temporary duration. Access to this flag is granted only after acquiring the global resource access mutex.

4.4.3. Packet Queues

The packet queue is one of the most important data structures of the protocol. It is designed based on the producer-consumer model where one entity inserts data into the

queue and another extracts data from the queue. From an implementation perspective we can compare the queue to a circular buffer where elements are always inserted in a clockwise direction and always the first element removed is the first element inserted. The queue holds packets and it is required that all the elements inserted into the queue have successive sequence numbers. Thus, the packet queue takes care of packet sequences and assigns a sequence number to every inserted element. To keep track of the elements inside the queue, 3 markers are used. One marker shows the next element to be removed, another marker shows the next position a new element will be inserted at and a third, send queue specific marker, shows the next packet to be sent.

There is a difference here on how the queue is used. If it's used for a sending buffer, the API functions insert elements into the queue where the second marker is positioned and increment it every time. When the sending thread wants to transmit a new packet, it takes the one pointed by the third marker, sends it, and increments the third marker. An element is removed from the queue only if the receiving thread obtains an ACK from the network for that particular packet and thus increments the first marker. When the queue is used as a receiving queue, only two markers are used. The receiving thread inserts into the queue a new packet and increments the second marker. The API function that receives elements removes packets from the queue and increments the first marker.

Since there is more than one execution unit accessing the data structure, all operations with the data structure is synchronized with an access mutex. Whenever an element is inserted or a marker incremented, the mutex must first be acquired and then released afterwards. Because of performance reasons, the queue must have an upper limit for how many elements it can hold. If an attempt is made to insert elements in a full queue, a return code of *QUEUEFULL* is returned. Furthermore, queries can be done to check if the queue is full or empty.

Given the circular nature of the queue, all marker operations are done modulo the size of the queue. The current size is 128 elements. This may be increased or decreased to obtain better performance according to experiment results. The size of the queue is practically the size of the sliding window in the *Go-Back-N* protocol. No more packets will be sent if there are already 128 sent that haven't received an acknowledgement. Once an ACK is received, the element is removed from the sliding window (from the packet queue).

4.5. Transmission Threads

To properly process packets and prepare them for sending over the network or receiving them from the network three types of threads are implemented. Each thread is implemented as a function that executes in a continuous while loop. In order to limit 100% processor usage and avoid a busy-waiting situation, synchronization mechanisms are implemented that wake a thread only when there is work for it to do.

4.5.1. Sender Thread

Since we may use multiple sender threads, we considered the following inmplementation. There is a global data queue, from which each sender thread extracts data in a synchronized manner. When a new packet is placed in the send queue of a stream, information regarding the communication channel and stream number are placed in the global data queue (e.g., a pointer to the stream's send queue). The global data queue's condition variable is signalled and a sender stream is woken up (if it was waiting at the global data queue's condition variable). Each sender stream continuously checks if there are any elements in the global data queue. If there are no elements, then it waits on the queue's condition variable. Otherwise, it removes the first element from the global data queue. Afterwards, using the extracted information, the sender stream sends the packet identified by the information from the global data queue to its destination.

After a packet is sent, the timer thread is announced of the need for a timeout after a certain amount of time. The current value of the timeout is 50 ms. Retransmission occurs if an acknowledgement is not received before the timeout is fired.

4.5.2. Receiver Thread

A receiving thread waits for data to be available for reading from any network socket the protocol manages. The UDP sockets are distributed among the existing receiving threads in a balanced manner. Dynamic redistribution is also possible (e.g., if a thread is very busy with receiving packets from some sockets, then some of the other sockets may be redistributed to other receiving threads, in order to avoid starvation or simply to improve performance), but was not implemented in out protocol.

Waiting is achieved using the provided operating system function called "select". If there is nothing to be read, the thread blocks and the execution scheduler gives CPU time to another thread. Because of the need for the thread to respond to events such as application shutdown, this waiting is not indefinite. Waiting is limited to 100 ms. After every waiting round, the thread checks if it needs to respond to any miscellaneous event such as shutdown.

Whenever a packet is received, the ACK message corresponding for that packet is prepared for sending. Currently the protocol sends an ACK message for each message it receives, but future iterations of the protocol can wait for multiple packets to be received and only send one ACK for the whole group.

4.5.3. Timer Thread

The timer thread acts as a ticking clock and constantly processes events every defined interval of milliseconds. This interval was chosen at 50 ms. More fine values are supported, but the protocol needs to consider other processes

that use CPU time. Because a clock that ticks without anybody listening to it is not efficient, the timer is only active while there is a packet timeout that needs an alarm. The timer maintains a list of timeout events that will be fired in the future. If this list of events is empty, the timer waits on a synchronized condition. The API of any client wanting to send data also signals the timer thread and unlocks it if it was waiting for this condition.

Because of the way Linux threads are implemented, *sleep* cannot be used. Using *sleep* will cause the whole process to wait for the given time. Therefore, a way was implemented to make only a thread of a process sleep for a certain period of time. The function *pthread_cond_timedwait* waits for a condition to be signaled. If the condition is not signaled until a specified time is reached, the thread continues operation. Using this function with a mutex and a condition that never gets signaled will always make the thread wait for until the specified time is passed. To wait for a relative amount of time, one can wait until the current time (given by gettimeofday) plus the relative time desired. After the wait begins, the wait time is not affected by changes to the system clock. Although time is specified in seconds and nanoseconds, the system has approximately millisecond granularity. Due to scheduling and priorities, the amount of time it actually waits might be slightly more or less than the amount of time specified. That is why very fine wait periods (e.g., 0.1 ms) have a high margin of error.

Currently, the timer thread is designed to work with a granularity no less than 10 ms. Experiments still need to be performed to check the error margins or performance gains if this value is changed.

5. Experiments and Practical Applications

To test the protocol, several applications were implemented and some tests were performed. All tests were realized over an 802.11g wireless connection. While the signal strength was at 95% and speeds are at maximum, packet losses can still occur. This is on purpose to check that the protocol shows no issues. All computers have an Intel Core 2 Duo CPU, but at different frequencies, have at least 1 GB of free RAM, and no other programs running at the same time. The desire was to have more than one core per computer. CPU frequencies are not that important as the CPU load never increases over 1% while using the test applications.

5.1. File Download

The experiment performed was the measurement of the time it took to transfer a file from a single server. The goal was to see if the time increased in a more than linear fashion if the file size increased linearly. Normally the increment in time should be proportional with the increment in file size. If this does not happen, that means there is a problem with the protocol when large quantities of data are transferred. This can be a synchronization issue that makes the protocol block for a small period of time, a small buffer that gets filled up quickly or something else. Also, using this simple experiment, one can check for any performance benefits by tweaking different parameters of the transmission algorithm. The transfer time was measured by measuring the time since the transfer application started and until the time the application finished. Linux's time utility was used in this regard. The experiment was run with random data files of sizes 1, 2, 4, 8, 16 and 32 MB (see the results in Table 1). In each case, a client transferred the file to a server in pieces of a certain size and received the same data back with an ACK. The client checked the data to validate it and make sure it is the same one sent and proceeded to the next chunk of data.

Table 1
Transfer time for 1024 B of data payload (untweaked)

File size [MB]	Transfer time [s]
1	1.743
2	3.375
4	6.784
8	13.735
16	27.694
32	55.394

For the first run of the experiment, the sliding window had space for 32 elements, each packet carried a maximum data payload of 1024 B and the timer retransmitted a packet after 50 ms. The protocol achieved an average speed of 4.6 MB/s. The speed was consistent regardless of the size of the file with only a very small improvement as the file size went up. This indicates that the protocol runs well and the larger the file size the less noticeable the handshake or shutdown procedure gets.

Table 2
Transfer time for 8096 B of data payload (tweaked)

File size [MB]	Transfer time [s]
1	0.463
2	0.867
4	1.654
8	3.166
16	6.279
32	12.563

For the second run of the experiment different parameters have been tweaked. The sliding window now has a size of 128 elements, each packet carried a maximum data payload of 8096 B and the timer retransmitted a non-acknowledged packet after 10 ms. The results can be seen in Table 2. The protocol achieved an average speed of slightly over 19 MB/s. Again, as the file size increased, the transfer speed increased slightly. Even though the packet will be

fragmented by the IP layer into pieces of MTU size, a speed improvement is still observed. Testing these values in a less reliable medium is mandatory to check if packet fragmentation increases packet loss rate.

We can see there is a great improvement in the speed of the protocol after modifying these parameters. The 19 MB/s (for both sending and receiving) is getting closer to the advertised speed of the 802.11g wireless standard.

Table 3
Transfer time for 8096 B of data payload and window size of 1

File size [MB]	Transfer time [s]
1	1.187
2	2.658
4	4.795
8	9.323
16	18.333
32	38.152

To verify the impact of the sliding window, a series of tests were performed where the sliding window accepted only one element. This basically means the protocol turned into a variant of the stop-and-go algorithm. All the other elements were left tweaked for improved speed. The results can be seen in Table 3. We can observe that speed has fallen back to around 6 MB/s of data transfer. The sliding window does have an impact on the performance of the protocol. This shows why, for the highest possible throughput, it is important that the transmitter is not forced to stop sending by the sliding window protocol earlier than one round-trip delay time (RTT). A different sliding window retransmission protocol might further improve performance. An example is selective repeat where the sender does not retransmit all packets starting with the lost one, but only retransmits the lost packet.

5.2. Segmented File Download

A practical implementation for the protocol is a segmented downloading program. A client connects to multiple servers at the same time and asks each of them for different segments of a file. This can also be done via existing protocols, but in these protocols a connection must be created for each server. With our protocol, a single connection can be created that touches every server and opens a stream with each of them. This makes implementation much simpler and error free. We only performed validation tests for this scenario.

6. Conclusions and Future Work

In this paper we presented a new approach for communication from one point to multiple endpoints. Our communication protocol is implemented on top of UDP and tries to provide the same facilities as TCP, but with extra functionality. The implementation of multiple endpoints is reliable, unlike multicast over UDP, and is not limited to only two endpoints that may have multiple IPs, like SCTP's implementation of multi-homing. Using the new protocol, applications can easily choose the type of connection they desire with the endpoints. Connection oriented multicast, anycast and load balancing are all fully integrated in our protocol. Experimental results showed that our protocol can provide good data transfer performance. However, as with any new protocol, further (extensive) testing is needed. We also intend to modify some of the components of the protocol, such as the connection open handshake and the limitation that a communication channel may use only a single UDP socket underneath (perhaps using multiple UDP sockets for sending and receiving data will be more efficient, due to having more buffer space allocated in the operating system kernel). Moreover, some of the parameters of the protocol (e.g., timeouts) still need to be tweaked for optimal performance.

Acknowledgements

The work presented in this paper was partially funded by the Romanian National Council for Scientific Research (CNCS)-UEFISCDI, under research grants ID_1679/2008 (contract no. 736/2009) from the PN II – IDEI program, and PD_240/2010 (AATOMMS – contract no. 33/28.07.2010), from the PN II – RU program, and by the Sectoral Operational Programme Human Resources Development 2007–2013 of the Romanian Ministry of Labour, Family and Social Protection through the financial agreement POSDRU/89/1.5/S/62557.

References

[1] W. R. Stevens, *TCP/IP Illustrated, Volume 1: The Protocols*. Addison Wesley, 1994.

[2] J. Postel, "RFC 768 – User Datagram Protocol", 1980 [Online]. Available: http://tools.ietf.org/html/rfc768

[3] R. Stewart *et al*., "RFC 2960 – Stream Control Transmission Protocol", 2000 [Online]. Available: http://tools.ietf.org/html/rfc2960

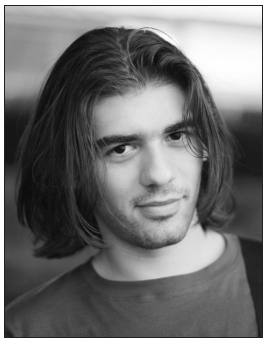
[4] M. I. Andreica, A. Drăguş, A.-D. Sâmbotin, and N. Ţăpuş, "Towards a (Multi-)User-Centric Stack of (Multi-)Point-to-(Multi-)Point Communication Services", in *Proc. 5th Worksh. Enhanced Web Serv. Technol. WEWST*, Ayia Napa, Cyprus, 2010, pp. 36–45.

[5] M. I. Andreica, A. Costan, and N. Ţăpuş, "Towards a Multi-Stream Low-Priority High Throughput (Multi)Point-to-(Multi)Point Data Transport Protocol", in *Proc. 6th IEEE Int. Conf. Intel. Comp. Commun. Proces. ICCP*, Cluj, Romania, 2010, pp. 427–434.

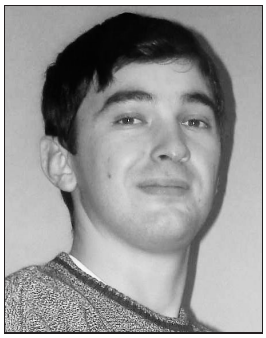
[6] T. Banka, P. Lee, A. P. Jayasumana, and V. Chandrasekar, "Application Aware Overlay One-to-Many Data Dissemination Protocol for High-Bandwidth Sensor Actuator Networks", in *Proc. 1stt Int. Conf. Commun. Sys. Software Middleware COMSWARE 2006*, New Delhi, India, 2006.

[7] J. Liebeherr, J. Wang, and G. Zhang, "Programming Overlay Networks with Overlay Sockets", *Lecture Notes in Computer Science*, vol. 2816, pp. 242–253, 2003.

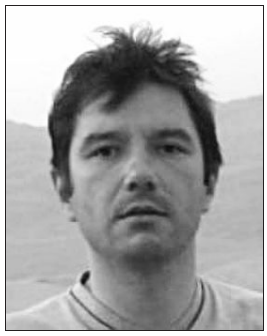
[8] M. Tuexen, and R. Stewart, "UDP Encapsulation of SCTP Packets", IETF Internet draft, 2011 [Online]. Available: http://tools.ietf.org/html/draft-ietf-tsvwg-sctp-udp-encaps-01



Valentin Stanciu is a graduate of Politehnica University of Bucharest and a teaching assistant at the same university. He obtained multiple prizes in different computer science competitions and participated at several internships with some of the biggest software companies (e.g., Adobe, Microsoft, Google). He is also an active board member of the Infoarena Association helping young students learn programming and train for competitions.
E-mail: valentin.stanciu@gmail.com
Politehnica University of Bucharest
Splaiul Independenţei 313
Bucharest, Romania



Mugurel Ionuţ Andreica, PhD, is an assistant professor in the Computer Science Department of the Politehnica University of Bucharest. His research topics include large scale distributed systems, P2P systems, distributed services, communication optimization and advanced, parallel and distributed data structures and algorithms. He was awarded the "Magna cum laude" distinction for his Ph.D. thesis and was granted an IBM Ph.D. Fellowship and an Oracle Research Scholarship for his Ph.D. research. He was the director of 2 national research projects and participated in 9 others (6 national and 3 international). He is (co-)author of 90 publications (books and papers in international journals and conferences). He obtained multiple prizes in algorithmic competitions (e.g., silver medal in the international olympiad in informatics and bronze medal in the ACM ICPC World Finals) and participated at several research internships (e.g., at Google Switzerland Gmbh and TU Delft) .
E-mail: mugurel.andreica@cs.pub.ro
Politehnica University of Bucharest
Splaiul Independenţei 313
Bucharest, Romania



Vlad Olaru holds a BS degree from the Politehnica University of Bucharest, Romania, an M.Sc. degree from Rutgers, The State University of New Jersey, USA and a PhD from the Technical University of Karlsruhe, Germany. His research interests focus on operating systems, distributed and parallel computing and real-time embedded systems. His Ph.D. work concentrated on developing OS kernel services for clusters of commodity-of-the-shelf PCs, namely cooperative caching for cluster filesystems and TCP connection endpoint migration for cluster-based servers. He was principal investigator within several EU-funded as well as national projects targeting the development of real-time OS software to control the next generation 3D intelligent sensors (with emphasis on power management and distributed control in ad-hoc wireless sensor networks), real-time Java for multi-core architectures, servers based on clusters of multi-core architectures, high-performance computing for computer vision programs.
E-mail: vlad.olaru@gmail.com
Politehnica University of Bucharest
Splaiul Independenţei 313
Bucharest, Romania

Nicolae Ţăpuş is a Professor of the Computer Science and Engineering Department, chair of this Department (1990–2008) and Vice-Rector of Politehnica University of Bucharest (UPB) since 2006. His main fields of expertise are Distributed Systems, Local Area Networks, and Computer Architecture. He is a Senior member of IEEE, ACM and Computer Society, chair of Romania Computer Society Chapter. He is director of ACM International Collegiate Programming Contest for Southeastern Europe since 1994. He received the award of the Romanian Academy, the award of Education Ministry and Innovation Award for High-Performance Applications from Corporation for Education Network Initiatives in California, CENIC. He is member of Romanian Technical Science Academy and Vice President of Information Technology and Communication Academy Section. He published more than 140 articles and papers at international conferences, 7 books and 12 university text books.
E-mail: nicolae.tapus@cs.pub.ro
Politehnica University of Bucharest
Splaiul Independenţei 313
Bucharest, Romania