Implementation and Interpretation of Protocols in the COMSCRIPT* Environment

Murhimanya Muhugusa, Giovanna Di Marzo, Christian Tschudin, Eduardo Solana and Jürgen Harms
Centre Universitaire d'Informatique, University of Geneva

Abstract

In this paper, we present COMSCRIPT, *a language designed for protocol entity implementation and protocol stack manipulation. The* COMSCRIPT *language is an interpreted language derived from* POSTSCRIPT *and enriched by concurrent processes which synchronize their execution and exchange data in a controlled and restricted way. The language adheres to an event driven programming approach which is very suitable for the implementation of both low and high level protocols. Communication between processes takes place through manipulable and flexible links created and configured dynamically.*

This paper presents the COMSCRIPT *language and its execution environment from a programming point of view. It introduces all its basic concepts showing how they can be used in the implementation of an application.*

Keywords: *computer communications, protocol implementation, protocol stack configuration,* COMSCRIPT.

1 Introduction

Classical computer communication software is structured as a stack of protocol entities which offer a well defined service to an application. However, in this approach the protocol stack is considered as a "black box" completely decoupled from the application. The application has no means to tailor the stack to its own requirements. Moreover, two hosts must have identical preconfigured protocol stacks to be able to exchange data; it is difficult to achieve interoperability of different protocol stacks.

This static approach to protocol stack implementation seems very restrictive. Many research works have shown the necessity of more flexible, reconfigurable protocol stacks [10] and new approaches have been proposed:

- The Unix System V STREAMS approach [14] allows different STREAMS modules to be dynamically pushed and/or popped on a STREAMS head. All the modules must however be precompiled in the kernel, the adjunction of new modules requires to recompile the kernel;
- A highly layered stack architecture using both "micro protocols" and "virtual protocols" has been implemented with similar or even better performance than monolithic protocol stacks[5];
- The three layered DaCaPo approach [8] allows an application to request, at initialization time, the desired quality of service; the middle layer ensures that the offered service permanently fulfills the application requirements.

We present here the COMSCRIPT approach to flexible protocol stacks. This approach brings more flexibility because the application can, at any moment, configure the protocol stack according to its needs. The interpreted aspect of COMSCRIPT and its interprocess communication model are the basic mechanisms used for achieving the desired stack flexibility.

The configuration process is not limited to the initialization phase, and it is even possible to (re)configure a remote protocol stack [13]. The possibility to download COMSCRIPT code for execution in a remote host is the basis for realizing communication between two hosts not having the same protocol stack, and thus to achieve interoperability. Moreover both interpreted COMSCRIPT modules and existing precompiled modules can be combined to build and to configure a protocol stack.

The COMSCRIPT approach to flexible protocol stacks uses a uniform way to both the implementation and the reconfiguration of protocol stacks: the same interpreted language is used to implement and reconfigure protocol stacks. Other approaches decouple the two aspects of this problem handling the reconfiguration differently from the implementation.

In section 2, we show the link between COMSCRIPT and POSTSCRIPT; section 3 presents COMSCRIPT processes, section 4 describes the interprocess communication in COMSCRIPT, while section 5 explains how COMSCRIPT platforms are connected to the outside world. Section 6 discusses some points relevant to COMSCRIPT, and some concluding remarks are given in section 7.

2 From POSTSCRIPT to COMSCRIPT

The COMSCRIPT language is derived from the POSTSCRIPT[1] language and thus shares its execution model, some of its operators and data structures. The following reasons have lead to the choice of POSTSCRIPT as a starting point: (a) POSTSCRIPT is widely used, (b) it has a very simple execution model, (c) the need for an interpreter for protocol implementation and manipulation, and (d) the

*This work is supported by Swiss National Fund for Scientific Research (FN-SRS) grant 20-34'070.92. E-mail: muhugusa@cui.unige.ch

[1]POSTSCRIPT© is a registered trademark of Adobe System Incorporated.

source code of GHOSTSCRIPT [2], a public domain interpreter was available.

We first removed all graphic related operators from GHOSTSCRIPT to get a bare COMSCRIPT interpreter. Then, we incrementally added new features for concurrency, interprocess communication, timeout handling and for access to the external world to the COMSCRIPT environment. We assume in this paper that the reader is familiar with the POSTSCRIPT stack execution model [1].

3 COMSCRIPT and processes

In a COMSCRIPT environment processes execute concurrently and are structured in a process tree. The first process created by the COMSCRIPT environment is the root of this tree and is called the `Rootprocess`. New processes are created with the `fork` operator. This operator takes as argument a procedure which will become the code to be executed by the newly created process and returns a reference to the created process.

```
1      /code1 { (ComScript is fun) print flush } def
2
3      /go { % create 2 new processes
4         % name the first process /child1
5
6         /child1 /code1 load fork def
7         { (Hello World) print flush } fork pop
8      } def
```

The interpretation of the above code is straightforward. Procedure `/code1` is defined on line 1: it just prints a message on the screen. The `/go` procedure is defined on lines 3 to 8. It creates two new processes (lines 6–7) and calls the first one `/child1`. After the execution of the `/go` procedure, the `Rootprocess` executes concurrently with the two created processes.

4 Process Synchronization and Data Exchange

COMSCRIPT allows concurrent processes to synchronize their execution and/or to exchange data. Data exchange between two processes can take place synchronously or asynchronously. COMSCRIPT restricts the interprocess communication (IPC) to a direct communication between a process and its children or between sibling processes. This section presents the ingredients used by COMSCRIPT IPC.

4.1 Synchronization Points, Event Handlers and Guards

The COMSCRIPT environment recognizes three kinds of events:

- a data exchange with another process or with the outside world has occurred;
- a synchronization with another process has occurred (without data exchange);
- a timeout period has elapsed.

A COMSCRIPT process can request that these events be signaled to it through synchronization points (s-points) attached to it. There are three kinds of s-points corresponding to these three kinds of events: (a) *input* and *output* s-points for synchronous or asynchronous data exchange, (b) *pure* s-points for synchronization without data exchange, and (c) *timer* s-points for synchronization with the system clock.

A procedure called an *event handler* is associated with each s-point. When an event occurs at a given s-point, its associated event handler is executed by the COMSCRIPT environment to handle the event.

A flag, called a *guard*, is associated with each s-point, allowing the process to *enable* or *disable* the s-point. Disabled s-points are ignored by the COMSCRIPT environment, they can not convoy an event. Thus, a process can choose at any moment the events to be signaled to it by the COMSCRIPT environment.

A process can wait for events on more than one s-point. When this configuration is used, the COMSCRIPT environment passes to the process only the first event to occur for handling. If multiple events can be realized when the process passes its request to the COMSCRIPT environment, the environment picks up one of the possible events in a non deterministic way.

Processes are event driven; most of the time they are blocked waiting for one of the requested events to occur. For each process, only one event can occur at a time. As a consequence, a COMSCRIPT process is logically structured in two parts: (a) an initialization code which is executed once when the process is created and (b) a collection of event handlers associated to the process' s-points. Amongst other, the initialization code is responsible for creating s-points and providing them with an event handler. After the initialization code is executed, the process enters an infinite loop where it is waiting for an event to occur on one of its s-points. Whenever an event occurs on an s-point, the associated handler is executed and the process waits for the following event. This continues until the process is explicitly killed.

4.2 Gates and Links

Processes that want to synchronize their execution or to exchange data must be "connected": for this, s-points are connected through *gates*. A gate implements a message queue of length over or equal to zero. A gate with a zero length queue is used to link two processes that want to exchange data in a synchronous way or that synchronize their execution without data exchange. A gate with a queue of length greater than zero is used to obtain an asynchronous data exchange between two processes.

In a communication between a process and its child, the parent process is responsible for providing a gate and configuring the links between its s-point and the associated s-point belonging to its child. In a communication between two sibling processes, their parent is still responsible for providing a gate and configuring the links of its children. In fact a process is never aware that it is communicating with its parent or with its sibling process. A parent process can also change at any moment the configuration links of its children in a transparent way.

Figure 1 shows two processes P1 and P2 linked through their s-points `/!out` and `/?in` and gate of a four length queue. The

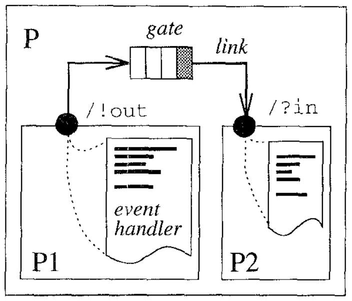


Figure 1: Two processes linked by their parent through a gate

gate belongs to their parent P which must *link* the two processes.

4.3 The Flexibility of the ComScript IPC Model

Although the ComScript interprocess communication model is simple, it is flexible and powerful as illustrated by the following considerations:

1. The use of gates in process communication links brings more flexibility. An s-point can be linked to only one gate but it is common to have multiple s-points linked to the same gate. ComScript does not handle this situation as a *broadcast*. On the contrary, only one process will receive the data item exchanged. This provides *non deterministic* data exchange. It becomes possible for a process to interact in a transparent way with another process belonging to a group of processes. Figure 2 shows three processes linked through the same gate.

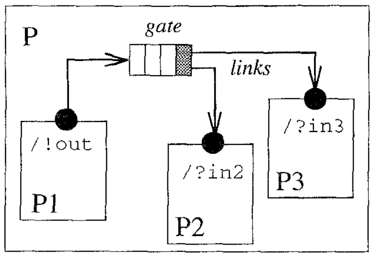
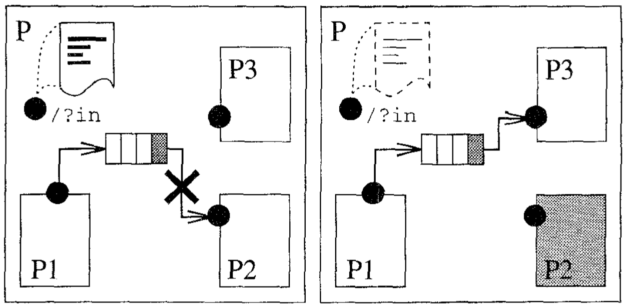


Figure 2: Multiple s-points linked through the same gate

2. It is possible to dynamically manipulate communication links between processes; a parent process can at any moment change the way its children are linked together, i.e. by (a) breaking their communication links, (b) replacing one of the communicating partners with another process and (c) reconfiguring the communication links so that subsequent interactions take place with the new configured process. This possibility is illustrated by figure 3. An event occurs on the `/?in` s-point of the parent process P and its event handler is activated (left part of the figure). The execution of the event handler leads the parent process to break the link between P1 and P2 and to establish a new link between P1 and P3. After the event handler has completed its execution (right part of the figure), P1 is ready to exchange data with P3. P1 is not aware of the fact that P2 has been replaced by another process.



a) before link reconfiguration *b) after link reconfiguration*

Figure 3: A dynamic reconfiguration of a communication link of process P1 by its parent P

5 Opening the ComScript environment to the outside world

ComScript processes can access the world outside the ComScript environment and exchange data with it via "device drivers". A device driver can be seen as an "external process" offering a well defined functionality. Examples of device functionalities are: receiving Ethernet packets, reading from a file or navigating in a file system.

Each device is uniquely identified by a name (just like PostScript fonts). Instances of these device drivers are created dynamically on demand by ComScript processes. As far as ComScript processes are concerned, all the device drivers are accessed in a uniform and consistent way; each device driver is an array of s-points. The s-points allow a ComScript process to access the well defined "services" of the device driver.

A device for reading a file, for example, would contain among others, an s-point for opening a requested file, an s-point for reading from the opened file, and another one to close the file after the read operation is completed.

Although some devices are likely to be implemented in every ComScript environment, the number of devices available and the complexity of the services they offer can vary greatly from one environment to another. All device types known in the system are contained in a global data structure called the *DeviceDictionary*. A ComScript process can define new devices and add them in the *DeviceDictionary*, in order to make them accessible by other ComScript processes.

Currently, our ComScript environment contains devices accessing:

1. the file system for creating, reading and writing files and directories;
2. the connectionless and connection oriented sockets (UDP and TCP);
3. the NIT (Network Interface Tap) device for access to the raw Ethernet on SUN.

Figure 4 shows the way the service offered by the NIT device driver is accessed by a COMSCRIPT process.

The following COMSCRIPT code shows how a process interacts with a device driver. In this example, we use the NIT device, the access to the other devices is done in a similar way.

```
1      /initdevice { % init (create a nit access)
2         /rwEthernet finddevice { } clonedevice
3         {
4            /ethernet exch def
5            [ 2 { 0 creategate} repeat ]
6            ethernet 1 index link1to1
7
8            /rnit /?r 1 createsync def
9            /wnit /!w 2 createsync def
10           [ rnit wnit ] exch link1to1
11        } {
12           (Error:  Can not clone rwEthernet device)
13           print flush
14        } ifelse
15     } def
16
17     /readpacket { % read one Ethernet packet
18        rnit input
19     } def
```

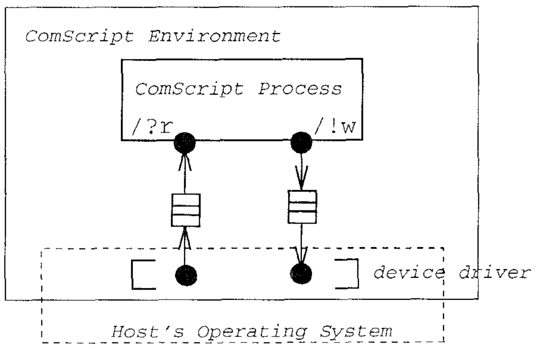


Figure 4: The view of the NIT device driver by a COMSCRIPT process

Two procedures are defined in the above code; the `/initdevice` procedure on lines 1–15 and the `/readpacket` procedure on lines 17–19.

The `/initdevice` procedure creates a new instance of the `/rwEthernet` device on line 2. The `finddevice` operator searches in the COMSCRIPT environment for the existence of the named device, (in this case the `/rwEthernet`) and returns a reference to the device if it exists in the environment. The `clonedevice` operator takes as arguments a device and a procedure, creates and returns a new instance of the device. The created device instance is an array of s-points. The procedure is used to control the access right to the device. In our example, no access control is done to create an instance of the device. The `rwEthernet` device has only two s-points, one for reading an Ethernet packet and one for writing a packet. On line 4, the created Ethernet device instance is saved in the `/ethernet` variable for later references. On line 5, an array of two gates of zero length queue is created. The `link1to1` procedure (code not shown here) links, on line 6, each s-point of the Ethernet device with the corresponding gate of the previously created array of gates. On lines 8–9, two s-points are created; an input s-point called `/?r`, stored in the `/rnit` variable, and an output s-point called `/!w` and stored in the `/wnit` variable. On line 10, the two created s-points are also linked to the gates created on line 5.

The `/readpacket` procedure reads one Ethernet packet. With the `input` operator on line 18, the process asks the COMSCRIPT environment to realize an interaction with the Ethernet device through the `rnit` s-point. As the `rnit` s-point is linked to the device's s-point implementing the 'read from Ethernet' service, this interaction results in the reading of an Ethernet packet.

6 Discussion

COMSCRIPT is not just another programming language. Its basic concepts are simple but yet powerful enough to allow a rapid and incremental implementation of protocol entities. Moreover, the COMSCRIPT environment offers the necessary ingredients to build in an elegant way very flexible protocol stacks. This section shows how COMSCRIPT can be used for the implementation of network applications.

6.1 Implementation of Protocol Entities

Communication software is classically structured in static layers of protocol entities forming a protocol stack. Each layer offers a well defined service to its upper neighbor layer and uses the service provided by its lower neighbor layer. The interaction between neighbor layers is done through the so-called service access points (SAP).

The COMSCRIPT language is well suited for the implementation of protocol entities and stacks. Each protocol entity module can be implemented either as a COMSCRIPT process or as a tree of communicating COMSCRIPT processes. A protocol stack is naturally implemented by concurrent COMSCRIPT processes and s-points are used to implement SAPs.

Moreover, as COMSCRIPT processes are event driven, it is quite straightforward to translate a finite state machine (FSM) protocol specification into its corresponding COMSCRIPT code. Each event occurring in the protocol machine can be captured by an s-point in an implementation. The action triggered by the event in the protocol machine is naturally implemented as the event handler associated to the s-point which captures the event in the implementation.

In the following example we show the code for the sender side of the well-known Alternating Bit Protocol (ABP). The code dealing with the external representation of PDUs has been omitted – we assume that some underlying entity will map the ABP PDUs (of the form `[true ackflag (datastring)]` for data packets and `[false ackflag]` for acknowledgements) to a flat string of bytes. Note also that the ABP entity passes the user data downwards regardless of its size and type.

The implementation is based on the finite state machine (FSM) model of figure 5. The FSM starts in the `idle` state and waits for a synchronization with a user entity. On reception of the user data, an ABP PDU is assembled and the FSM is ready to send it down the channel (state `rdy`). After synchronization with the channel, we have to wait for an acknowledge message or a timeout event (state `wack`). Only an acknowledgement with the right sequence flag

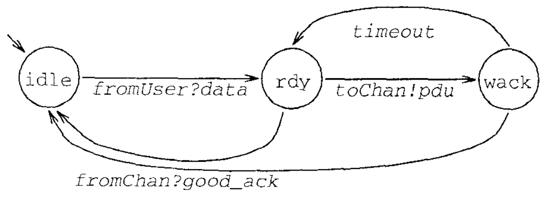


Figure 5: The Finite State Machine for the ABP Sender

leads back to the initial `idle` state, while a timeout requires the retransmission of the PDU. All other incoming acknowledgements are then discarded (this is not shown in figure 5).

```
1   /abpSender {
2      clear
3
4      /ddbegin { dict exch 1 index def begin } bind def
5      /findsync {
6         currentprocess /syncdict get exch get
7      } bind def
8      /installsync { createsync begin
9         /handler {
10           transdict state get 1 index get exec
11        } bind def
12        /guard {
13           transdict state get exch known
14        } bind def
15     end } bind def
16
17     /encodePDU { } def % currently empty
18     /decodePDU { } def % currently empty
19
20     /?fromUser 1 installsync /!toChan 2 installsync
21     /?fromChan 1 installsync /$timeout -1 installsync
22
23     /transdict 3 ddbegin
24        /idle 2 ddbegin
25           /?fromUser findsync {
26              /data get [ true seqflag 4 3 roll ]
27              encodePDU /pdu exch def
28              /!toChan findsync /data pdu put
29              /state /rdy def
30           } def
31           /?fromChan findsync { pop } def
32        end
33        /rdy 2 ddbegin
34           /?fromChan findsync {
35              /data get decodePDU 1 get seqflag eq
36              {
37                 /seqflag seqflag not def
38                 /state /idle def
39              } if
40           } def
41           /!toChan findsync {
42              pop /$timeout findsync /timer gmt
43              delta add put
44              /state /wack def
45           } def
46        end
47        /wack 2 ddbegin
48           /?fromChan findsync dup rdy exch get def
49           /$timeout findsync {
50              pop /!toChan findsync /data pdu put
51              /state /rdy def
52           } def
53        end
54     end
55
56     /state /idle def
57     /seqflag false def
58     /delta 1000 def % set timeout to 1 second
59  } def
```

The ComScript implementation uses a table driven approach: to each allowed combination of state (`idle, rdy, wack`) and event (`fromUser, toChan, fromChan, timeout`) we associate a procedure that has to be executed if that event "happens" in the given state. If there is no procedure defined for a <state, event> pair, it means that the given event should not be enabled in the given state. The transition table is stored in the `transdict` dictionary: to each state is associated a dictionary, in which the procedures are stored for the allowed events. What the implementor of a FSM based protocol entity has to do is to declare the set of possible events (lines 20–21), to set up the `transdict` (lines 23–54), and to initialize the value of the `state` variable and other protocol dependent values (lines 56–58).

How are activated the procedures defined in `/transdict`? How are set the guards? The small support needed for this can be found in the procedure `/installsync` on the lines 8–15. Every "event" is represented by an s-point. In fact, all s-points will have the same generic event handler and the same generic guard procedure. To enable a s-point we just have to check if a procedure is defined for the given <state, sync> pair. The generic event handler just looks up this procedure and executes it.

ComScript code can be "compressed" by using standard PostScript techniques to compact code. This requires the use of short variable names, the redefinition of often used and long keyword sequences and the inclusion of spaces only where necessary. This reduces the overhead when ComScript code has to be downloaded to a remote host. Using this compression technique, the ComScript code of a full duplex ABP protocol entity fits in less than 700 bytes.

6.2 The Configuration of Protocol Stacks

In the previous section, we have seen that it is quite simple to implement protocol entities in ComScript. It is also simple to implement an entire protocol stack in the framework of existing standards.

Moreover, ComScript contains the ingredients to allow the building and the (re)configuration of flexible protocol stacks. In ComScript, a protocol stack can be implemented by concurrent ComScript processes which synchronize their execution and exchange data through their s-points. At any moment, the "s-point-gate" links between processes can be broken and reconfigured differently leading to the protocol stack (re)configuration.

6.3 Code downloading in ComScript

Another attractive aspect of ComScript is that it allows code downloading. A ComScript process running in an environment can send ComScript code which will be executed by a ComScript environment running in a remote host. This is done by installing in each host, a ComScript server which executes ComScript code received from clients. Two such servers have been implemented in our prototype.

6.4 Current State

A first ComScript interpreter prototype has been implemented. Both low level and high level protocol entities have been implemented in the framework of existing standards. The environment proved to be also suitable for the (re)configuration of protocol stacks. Moreover, it has been also possible to configure an optimized protocol stack in ComScript, using optimized precompiled protocol entities implemented in other languages, and thus to overcome the disadvantages of protocol interpretation.

Two COMSCRIPT servers have been implemented: one server uses the UDP protocol and the other, the TCP protocol. While all requests directed to the UDP server are handled in the same COMSCRIPT environment, each connection established with the TCP server leads to the creation of a new COMSCRIPT environment which handles all subsequent requests.

Using these servers, we successfully realized code downloading of a whole protocol stack and the data exchange between two hosts not having identical preconfigured protocol stacks.

Being only a prototype, our implementation does not address neither performance considerations, nor security aspects which must surely be carefully addressed in a productive environment. Our effort has been focused on the implementation of the basic concepts underlying COMSCRIPT to show that they are feasible and suited for network programming.

7 Conclusion

COMSCRIPT is both an interpreter and an environment for the execution and the synchronization of concurrent processes. The primary aims pursued by the COMSCRIPT project is the implementation and the configuration of protocol stacks. The basic model underlying COMSCRIPT, i.e. concurrent event driven processes and their manipulable communication links formed of s-points and gates, proved to be well suited for the efficient realization of these aims.

Experiments with communities of COMSCRIPT nodes have been made. In such a community, it is possible to realize code downloading, communication between two hosts with different protocol stacks, as well as dynamic reconfiguration of a remote protocol stack directly by the application.

The future of protocol stacks lies in the flexibility and reconfigurability. We are convinced that the COMSCRIPT approach will bring some promising solutions in the areas of interworking between different protocol stacks and the tailoring of application-specific stacks.

References

[1] Adobe Systems Incorporated, editor. *PostScript Language: Reference Manual,* Addison-Wesley, fifteenth edition, 1990.

[2] Deutsch L. P. (Aladdin Enterprises), *GhostScript—An Interpreter for the PostScript Language,* distributed under the GNU General Public License.

[3] Hutchinson N. C., Peterson L. L., *The x-kernel: An Architecture for implementing network protocols.* In: IEEE Transactions on Software Engineering, Jan. 1991, pp. 64–76.

[4] Muhugusa M., Solana E., Tschudin Chr. F., Harms J., *ComScript — Implementation and Experiences,* Internal report, Université de Genève, Nov. 1992.

[5] O'Malley S. W., Peterson L. L., *A Highly Layered Architecture for High-Speed Networks,* In Marjory J. Johnson, editor, Protocols for High-Speed Networks II, pp 141–156, Elsevier, 1991.

[6] O'Malley S. W., Peterson L. L., *A Dynamic Network Architecture,* In: ACM Transactions on Computer Systems, Vol. 10, No. 2, May 1992, pp. 110-143.

[7] Plagemann T., Plattner B., Vogt M., Walter T., *A Model for Dynamic Configuration of Light-Weight Protocols,* In: IEEE Third Workshop on Future Trends of Distributed Computing Systems, Taipei, Taiwan, April 1992, pp. 100–106.

[8] Plagemann T., Plattner B., *Modules as Building Blocks for Protocol Configuration,* Proceedings International Conference on Network Protocols, ICNP'93, San Francisco, CA, Oct. 19–22, 1993, pp. 106–115.

[9] Plagemann T., Gotti A., Plattner B., *CoRA — A Heuristic for Protocol Configuration and Resource Allocation,* Submitted to IFIP Fourth International Workshop on Protocols for High-Speed Networks, Vancouver, Canada, Aug. 10–12, 1994.

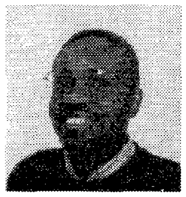
[10] Tschudin Chr. F., *Flexible Protocol Stacks,* In SIGCOMM'91 Conference on Communications Architectures & Protocols, pp. 197–204, Sept. 1991.

[11] Tschudin Chr. F., Muhugusa M., Solana E., Harms J., *ComScript — Concept and Language,* Internal report, Université de Genève, Nov. 1992.

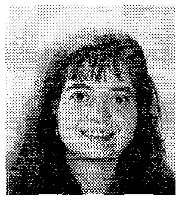
[12] Tschudin Chr. F., *On the Structuring of Computer Communications,* Ph.D Thesis no. 2632, Université de Genève, 1993.

[13] Muhugusa M., Di Marzo G., Tschudin Chr. F., Solana E., Harms J., COMSCRIPT*: An Environment for the Implementation of Protocol Stacks and their Dynamic Reconfiguration,* in the proceedings of ISACC'94, Monterrey, Mexico, Oct. 26-27, 1994.

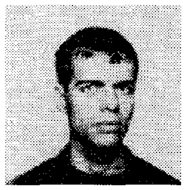
[14] Sun Microsystems, Inc. STREAMS *Programming Manual,* March 1990.



Murhimanya Muhugusa received the M.Sc. degree in computer science in 1991 and the B.Sc. degree in mathematics in 1994 all from the University of Geneva, Switzerland. He is currently member of the Teleinformatics research group of the Centre Universitaire d'Informatique (CUI) and a Ph.D. student in the "Département d'Informatique" of the University of Geneva. His research interests are in the fields of Teleinformatics, Distributed and Operating Systems.

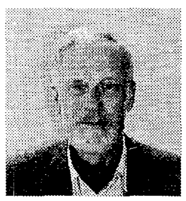


Giovanna Di Marzo received the M.Sc. degree in mathematics in 1993 and the M.Sc. degree in computer science in 1994 all from the University of Geneva, Switzerland. She is currently member of the Teleinformatics research group of CUI and a Ph.D. student in the "Département d'Informatique" of the University of Geneva. Her research interests are in the fields of Computer communications, Distributed and Operating Systems, and Formal Specification Languages.



Christian Tschudin received the M.Sc. degree in mathematics in 1986 from the University of Basel, Switzerland. Afterwards he worked for two years at the University's computing center before joining the Teleinformatics research group of the CUI of the University of Geneva. He received his Ph.D. in computer science in 1993 and is currently a "maitre assistant" in the CUI. His main research interests are communication protocol implementation and interpretation, mainly in terms of communication messengers, as well as the design of distributed operating systems.

Eduardo Solana received the M. Sc. degree in computer science in 1991 from the University of Geneva, Switzerland. He is currently member of the Teleinformatics research group of the Centre Universitaire d'Informatique (CUI) and a Ph.D. student in the "Département d'Informatique" of the University of Geneva. He is a member of the IBM technical staff of Geneva. His research interests are in the fields of Computer communications, Management of Distributed Applications, and Cryptology.



Jürgen Harms Studies in physics at the Polytechnical school of Vienna (Austria) and in electronic engineering at the University of California (USA). Since 1972 professor of computer science at the University of Geneva. President of the SWITCH foundation (Swiss national research network). Fields of interests: operating systems, teleinformatics.