COMSCRIPT*: An Environment for the Implementation of Protocol Stacks and their Dynamic Reconfiguration

Murhimanya Muhugusa, Giovanna Di Marzo, Christian Tschudin,
Eduardo Solana and Jürgen Harms
Centre Universitaire d'Informatique, University of Geneva
muhugusa@cui.unige.ch

Abstract

The need for flexible protocol stacks in communication software, instead of static, predefined protocol stacks, has been more and more asserted these last years. We present here a new environment, COMSCRIPT*, which addresses the implementation of flexible protocol stacks directed by the application.* COMSCRIPT *is a new programming language, derived from* POSTSCRIPT*, which follows an interpretative approach to perform protocols.* COMSCRIPT *is also an interpreter, that lets execute concurrently event driven processes, whose communications occur, either synchronously or quasi-asynchronously, through synchronization points linked to gates. This paper explains these concepts and shows that the* COMSCRIPT *language is suited for the implementation of communication protocol entities. It also shows how the* COMSCRIPT *environment uses these concepts to achieve dynamic protocol stack configuration.*
Keywords: *computer communications, protocol implementation, protocol stack configuration,* COMSCRIPT.

1 Introduction

Computer communication software is currently based on the classical OSI [Rose 90] and TCP/IP [Comer 86] models. These models structure the communication software linearly in layers, forming a protocol stack. Each layer is responsible for a predefined set of communication services. For two hosts to communicate, they must have identical preconfigured protocol stacks.

The whole protocol stack is seen by an application as a black box capable of realizing a certain service. This means that an application can neither modify the protocol stack, nor configure it to meet its own requirements. As different applications have different communication requirements, these classical frameworks for computer communication software seem very restricted. Moreover, they are very rigid due to the fact that identical preconfigured stack must exist in the two communicating hosts.

More interest is now given to the appropriate usage of 'useful' protocols; i.e, their configuration to achieve the required communication service efficiently. Thus the approach of static, preconfigured stacks is increasingly questioned and new approaches have been proposed:

- Inside existing operating systems, a need has been identified to separate communication software more clearly from the OS's kernel, the STREAMS concept represents an important example of this approach [Strea 90].
- Dynamically adaptable protocol stacks are proposed as a means to overcome the problems of interworking of different protocol architectures [Tschu 91].
- A highly layered stack architecture and the use of both 'micro protocols' and 'virtual protocols' have been devised and implemented, showing similar or even better performance than monolithic protocol stacks [OMall 91].
- In the area of high speed networking, the assembly of protocol entities in a entire communication stack based on an application's requirements is proposed as a means to overcome the performance bottleneck of protocol software [Plage 92].

All these approaches to communication protocol implementation go beyond the classical reference models which assume a restricted and fixed number of layers through all implementations; they surely bring some degree of flexibility. Section 5 presents and discusses current work in this area and shows how it relates to COMSCRIPT.

The COMSCRIPT approach brings more flexibility by allowing an application to dynamically (re)configure an entire protocol stack to its specific needs. An application can not only choose *what* services the protocol stack has to offer, but the application can also specify *how* the protocol stack can best fulfill its needs. For example, an application can download a whole protocol stack into a remote host before starting the data exchange.

The COMSCRIPT philosophy is based on the two following principles:

1. the possibility of protocol interpretation instead of restriction to precompiled protocol functionalities;
2. the application is able to configure the best protocol stack.

The interpretative aspect of COMSCRIPT allows then an application to dynamically configure the protocol stack if necessary.

*This work is supported by Swiss National Fund for Scientific Research (FNSRS) grant 20-34'070.92

Section 2 gives an overview of COMSCRIPT; section 3 discusses interprocess communication in COMSCRIPT; section 4 shows how COMSCRIPT achieves protocol stack configuration and we conclude this article in section 6 after discussing related work in section 5. We give in the appendix an example of COMSCRIPT code showing a simple protocol stack (re)configuration.

2 An overview of COMSCRIPT

We named our environment COMSCRIPT, by analogy with POSTSCRIPT[1]. COM stands for computer communications to stress the fact that COMSCRIPT is a language designed for network programming. SCRIPT stresses two things (a) the way protocol entities can be implemented: the language allows a simple, rapid and incremental implementation of protocol entities similar to shell scripts and (b) the way protocol stacks are (re)configured using an interpreted language.

As we said before, COMSCRIPT is both a language and an executing environment.

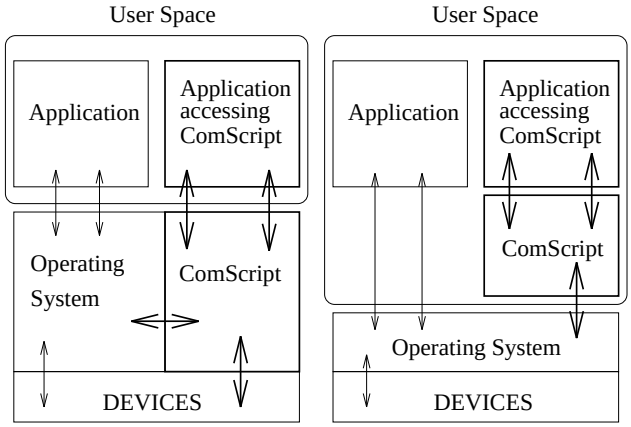
- The COMSCRIPT language is derived from POSTSCRIPT [Adobe 90]. Just like POSTSCRIPT is used for page description and for printing, COMSCRIPT is intended to be used for protocol implementation and stack (re)configuration. Being derived from POSTSCRIPT, COMSCRIPT inherits its simple execution model, its syntax, and some of its data structures and operators. All POSTSCRIPT graphics related operators have been removed and new operators have been added to support concurrency and interprocess communication.

- The COMSCRIPT environment is an interpreter executing processes written in the COMSCRIPT language. In the COMSCRIPT environment, a communication protocol entity can be implemented as a process; but a COMSCRIPT process can also be used to implement a whole protocol stack. Both low level (e.g ARP) and high level (e.g FTP) protocols have been implemented in COMSCRIPT.

Applications which need to communicate with others should access all the communication functionalities through the COMSCRIPT environment. To achieve this, the COMSCRIPT environment should interface with both (a) the host's Operating System—to get access to the communications facilities implemented in the OS—and (b) the applications—to allow them to (re)configure the communication facilities as needed. The current implementation of COMSCRIPT runs in user space; actually, the application runs on top of the COMSCRIPT environment and is seen as an extension to it. Figure 1 shows (a) the way COMSCRIPT should interface with the OS and the applications and (b) the current software structure within a host running COMSCRIPT.

In this section, we present the basic concepts underlying COMSCRIPT; the first part is concerned with the process hierarchy adopted, the second presents how events are signaled to a process with the synchronization points, the third part covers the configuration links between processes and the notion of gate, the fourth part explains how events are handled by processes. Finally, the last part describes the access to the world outside COMSCRIPT using device drivers.

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



(a) Planned software structure *(b) Current software structure*

Figure 1: The software structure within a host running COMSCRIPT.

2.1 Processes hierarchy

As stated before, COMSCRIPT allows multiple processes to execute concurrently. COMSCRIPT enforces a hierarchical structure on processes; each process other than the *RootProcess* (the first process created by the environment itself) is explicitly created by another process executing the *fork* operator. Adequate operators are provided to allow a process to manipulate a whole subprocess tree.

2.2 Synchronization Points and Guards

A COMSCRIPT process can be seen as an agent with attached synchronization points called *s-points* which allow it to interact with internal and external events. These s-points can be seen as sensors by which events are signaled to the process. Each event can convey one of the following meanings:

- a requested data exchange, with another process or with the outside world, has occurred;

- a requested synchronization with another process has occurred (no data exchange);

- the timeout period requested by the process has elapsed.

S-points are either external or internal. External s-points deal with events external to the process, those in which a process other than a child process is involved. Internal s-points are only concerned with events which result from the interactions between the process and its children or the COMSCRIPT environment.

Each s-point has associated to it a *guard*. This is a flag which determines whether the s-point is activated or not. Only 'activated' s-points can trigger events; all others are ignored by the

COMSCRIPT environment. This mechanism allows a process to choose at any moment the set of events to which it wants to respond.

2.3 Gates and Links

Processes, that want to communicate, must be connected. These connections are established of links between s-points and *gates*. A gate implements a queue of length over or equal to zero. In a communication between two sibling processes, the parent process is responsible for correctly configuring communications. The parent links an s-point of one of the children to a gate. It then links an s-point of the other child to the same gate. Once this operation is completed, the children are able to either synchronize their execution or exchange data.

Figure 2 shows a configuration link between two processes P_1 and P_2, linked in order to exchange data. The s-point `/!out` of P_1 is linked to gate `g`, the s-point `/?in` of P_2 is linked to the same gate `g`. P_1 is offering data, while P_2 is accepting data. Note

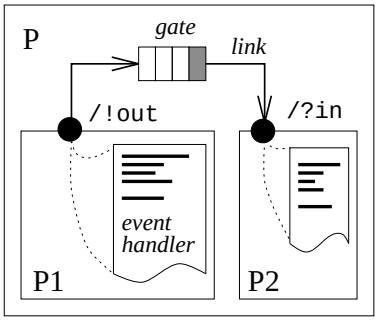


Figure 2: Processes linked through s-points and gates.

that two processes may be involved in a communication, without being aware of their partner. Only the parent process knows the configuration links between its children, and may change them when it wants.

2.4 Event driven programming

Another view of a COMSCRIPT process is that of a finite state machine (FSM) [Hopcr 79]. COMSCRIPT processes are event driven. This is to say that most of the time, a process is blocked waiting for an event to occur on one of its s-points; when this happens, the process is woken up to handle the event and subsequently goes back to a wait state. A process is thus composed of two logically distinct parts:

- an initialization code which is executed once after the process is created, and which is responsible for the creation of s-points;
- a number of *event handlers:* an event handler being the code associated to an s-point and which is executed by the process to handle an event occurring on the s-point. Thus, handling an event is synonymous with executing or activating the event handler associated to the s-point where the event occurs.

For any running process, only one event can occur at a time, and thus only one event handler is active inside a process. Moreover, while executing the event handler, the process is allowed to block on some other events if it wishes to do so. Figure 3 shows the different states of a COMSCRIPT process.

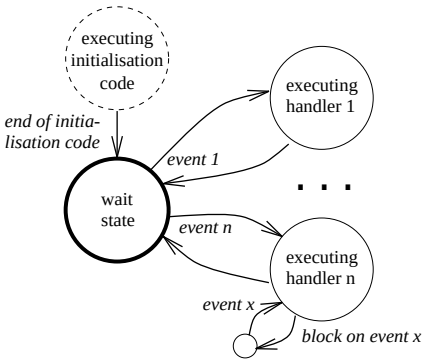


Figure 3: The states of a COMSCRIPT process

This process model is well suited to the implementation of FSMs which themselves are used to describe and to implement a large number of computer protocols. Like a COMSCRIPT process, a computer communication protocol entity is waiting for an incoming service request. The request is handled and the protocol entity returns to a wait state.

2.5 Access to the outside world

"Device drivers" allow COMSCRIPT processes to synchronize their execution and to exchange data with the "world" outside the COMSCRIPT environment. All device drivers are accessed in a uniform way: they are seen as "external" processes with synchronization points attached to them and accessible to processes running in the COMSCRIPT environment. Instances of these synchronization points are dynamically created on demand by a COMSCRIPT process.

Actually a device driver is seen as an array of s-points each implementing a defined functionality of the driver. A device for accessing the file system for example, would contain among others, an s-point for opening a file, and another for reading from the opened file. Reading the file is only possible if the file is already open. This means that the synchronization with the s-point responsible for reading from the file would be possible only after the synchronization with the s-point responsible for opening the file.

Although some devices are likely to be implemented in every COMSCRIPT environment, such as devices for accessing the file system or the system console, the number of devices available and the complexity of the services they offer can vary greatly from one environment to another. The *DeviceDictionary* is a global data structure containing all the device types known in the system. A COMSCRIPT process can define and add new devices to the *DeviceDictionary*, in order to make them usable by other processes. Similar to POSTSCRIPT fonts, each device has a name which uniquely identifies it.

Currently, our COMSCRIPT environment contains devices accessing:

1. the file system for creating, reading and writing files and directories;
2. the connectionless sockets for datagrams;
3. the connection oriented sockets for the implementation of client processes and the connection oriented sockets for the implementation of server processes;
4. the NIT device for access to the raw Ethernet on SUN.

3 Interprocess Communication in COMSCRIPT

We described above the COMSCRIPT components necessary to perform interprocess communication (IPC). These elements, s-points and gates, when correctly linked, establish channels between processes. The exchanged data can then transit through these channels. This same means allows for the synchronization of processes. Let us now see, exactly how data moves from one process to another process, and how two processes synchronize their execution.

This section first presents how synchronous and asynchronous communications are realized, it then explains the IPC mechanism, i.e. exactly how a synchronization or a data exchange is performed, and finally some additional considerations concerning this IPC are presented.

3.1 Synchronous and Asynchronous IPC

Beyond the task of linking s-points, a 'gate' contains a queue which is used to buffer data exchanged between processes. When pure synchronization or synchronous data exchange is needed, a gate of zero length queue is used. On the contrary, when asynchronous data exchange is desired, a gate of non-zero length queue should be employed.

3.2 How does interprocess communication take place?

After the preparatory work has been done, i.e. complementary s-points linked through an appropriate gate, the two processes can then request to be synchronized or to exchange data through these s-points.

If a gate of zero length queue is used (case of pure synchronization or synchronous data exchange), the first process to execute its synchronization request is blocked by the COMSCRIPT environment until the other process executes the equivalent synchronization request. Then a rendez-vous takes place between the two processes and if requested, data is moved from the 'outputting' process to the appropriate buffer in the 'inputting' process. The two processes then resume their execution; this is done by activating in each process the event handler associated to the s-point used to catch this event.

If on the contrary, a gate of non zero length queue is used, the gate buffers the data to be exchanged. In this case, the 'inputting' process is blocked only if the gate queue is empty; and the 'outputting' process is blocked only in the case of a full gate queue. In this way, some degree of asynchronism is achieved.

3.3 The flexibility of the COMSCRIPT interprocess communication model

We have so far presented how basic interprocess communication is realized in COMSCRIPT. The power and the flexibility of this interprocess communication model lie in the following aspects:

1. **The possibility to request synchronization on multiple events:** Multiple s-points can be involved in a synchronization request or data exchange request. If one or more events can be realized when the process request is handled by the COMSCRIPT environment, one of these events is picked up in a non deterministic way and is returned to the requesting process. The other events are discarded. If on the contrary, the requesting process is blocked because none of the events can be immediately realized, the first event to occur later on is transmitted to the requesting process and the others are discarded. This functionality is similar to that offered by the *select* construction in Ada programming language [Cohen 86] with the difference that the COMSCRIPT *select* is symmetric, i.e. it can handle both data input and data output requests, while the Ada *select* can handle only data input requests (accept).
2. **The possibility to link more than two s-points to the same gate:** There is no one-to-one correspondence between s-points. An s-point can be linked to only one gate, but it is common to have multiple s-points linked to the same gate. In this case, an s-point is logically seen as 'attached' to many others. This configuration is used in COMSCRIPT to synchronize one process or to allow it to exchange data in a non deterministic way with one process belonging to a group of processes. With this configuration no "broadcast" is achieved: COMSCRIPT does not allow either a synchronization or a simultaneous data exchange involving more than two processes. Figure 4 shows three s-points linked through the same gate. A data value outputted by P_1 is input by either process P_2 or P_3; the same value is never read by the two processes.
3. **Control over the communication patterns of child processes:** In COMSCRIPT, direct interprocess communication is restricted (a) between a process and its parent or children and (b) between children of the same process. In each case, a process is responsible for providing the necessary gates and also for realizing the appropriate links between the s-points of its children and the gates. This gives a COMSCRIPT process complete control over the interactions of its children. The process can dynamically alter the different s-point-gate links used by its children without the children being aware of that change. This restriction of

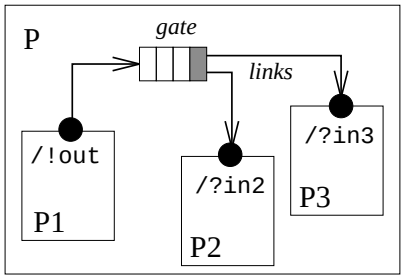


Figure 4: Multiple s-points linked through the same gate.

direct interprocess communications to some processes does not impose a great penalty on COMSCRIPT for the following reasons:

(a) In computer communication software, each layer of a protocol stack usually communicates only with the neighbor layers using the encapsulation mechanism.

(b) Data exchanges with other processes (which do not belong to the restricted set of processes with which direct interaction can occur) can be done using a chain of intermediary data exchanges.

(c) COMSCRIPT offers a *bypass* mechanism by which a process can delegate to a child the handling of events occurring at one of its s-points. Figure 5 shows a chain of bypassed s-points. Process P_1 delegates the handling of events occurring at s-point `/?in1` to its child P_2 which itself delegates the handling to its own child P_3. In this way, s-point `/!out`, belonging to *P*, and s-point `/?in`, belonging to P_3, are linked through the gate. Now a communication can take place between those two processes, without the intervention of the intermediary processes (P_1 and P_2).

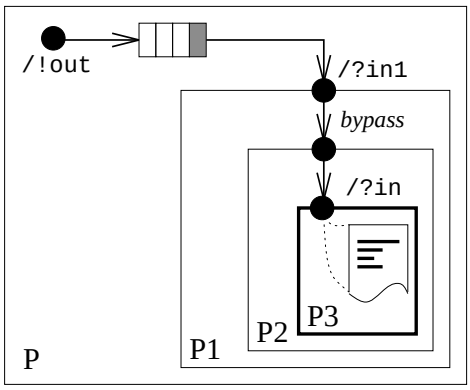
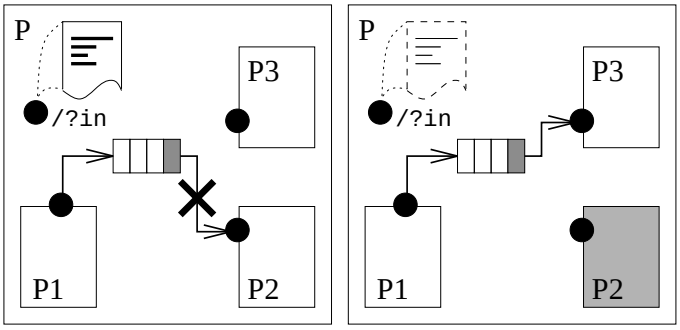


Figure 5: A chain of bypass between s-points `/!out` and `/?in`

4 From basic concepts to stack (re)configuration

In COMSCRIPT, the "gate" concept plays a central role in the reconfiguration process. The gate is the stable part of a *link* between processes through their s-points.

When two sibling processes communicate (synchronize their execution or exchange data), they are linked through a gate belonging to their parent. None of these two processes is aware of its partner in the communication under way. The parent process can configure the link of its children i.e. attach new s-points to the link or detach some of them in a completely transparent way to its children. The parent process can even replace one of the partners in the communication without the other being aware of that fact. Figure 6 shows a dynamic reconfiguration of a communication link. An event is triggered by s-point `/?in` belonging to process *P*, and its handler is executed. First, process *P* breaks the link between process P_1 and P_2 by detaching the s-point of P_2 from the gate. Then, *P* links P_1 with process P_3. This simple reconfiguration mechanism is the basis for building flexible protocol stacks [Tschu 91].



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

Figure 6: A dynamic reconfiguration of a communication link of process P_1 by its father *P*

A simple example will show how a COMSCRIPT process can configure its local protocol stack to meet its own requirements; a second one will present how configuration of a remote protocol stack is achieved.

4.1 Local stack (re)configuration: A simple example

Building a flexible protocol stack is a simple and straightforward task in COMSCRIPT as the following example illustrates. We will build a protocol stack where protocol entities can be dynamically added, removed or replaced by others in an efficient way.

Our example mimics and extends the UNIX SYSTEM V STREAMS concept. The STREAMS approach allows a user to customize a protocol stack by pushing and/or popping the adequate packet processing modules on the stream head. A number of such modules (example buffering module, filtering module) are compiled in the kernel. They all have a uniform interface. In each module there is a two way flow of information. An up flow to move data and control information from the network interface (STREAMS bottom) to the application which interfaces with the stack at the STREAMS head; and a down flow to move information in the opposite direction. Using a common and uniform interface, makes it possible to either add or remove a module in

a transparent way to the peer modules since the interface they use to exchange data remains stable.

In our COMSCRIPT example, the user is not limited to push/pop operations, he can add or move any module at any level of the stack. Our protocol stack is represented as an array of processes controlled by a stack handler; each process representing a protocol entity. The stack handler interfaces with the application and carries out on the stack the operations requested by the application. All the processes share the same interface: four s-points. `/?upin` and `/!downout` to move information towards the bottom of the stack, and `/?downin` and `/!upout` to move information to the top of the stack. A dictionary, let us say, `/stackgates,` associating a protocol entity with an array of gates linked to the module, is maintained by the stack handler.

The first operation is a `reset` operation: it is requested through the `/?reset` s-point, in order to initialize the stack by specifying the device driver which is used to connect the stack to the network. The `reset` operation opens the specified device and links it through two gates to the application interface. The application interfaces with the stack modules by two s-points, one for inputting `/?in` and the other `/!out` for outputting. The gates are put in the `/stackgate` dictionary to reflect the current links.

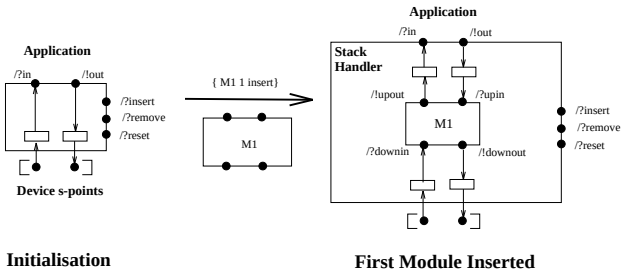


Figure 7: Initialization of a protocol stack and pushing of the first layer

The `add` operation allows the addition a module onto the stack. A process having the required interface is supplied, together with the level in the stack where the module it implements is to be inserted. The stack handler creates the necessary gates to link the new module. Figure 7 shows the initialization of the protocol stack and the adding of a first module M1. In figure 8 module M2 is added on top of the protocol stack and in figure 9 the middle module of a three layer stack is removed.

A `remove` operation is indicated to the stack handler through an event occurring on its `/?remove` s-point. The position of the module to be removed is specified. The stack handler breaks the links of the module, removes it from the stack, reconfiguring as appropriate the links of the neighbor modules and updating the `/stackprocess` and `/stackgate` data structures.

While in the STREAMS approach, a user is limited by the number of available modules compiled in the kernel, the number of different usable modules is not limited in COMSCRIPT. Indeed, the modules are dynamically created when needed and must not be preconfigured in the kernel. This is possible because the modules are defined in terms of a COMSCRIPT procedure that defines the module's behavior. Thus our example extends the STREAMS

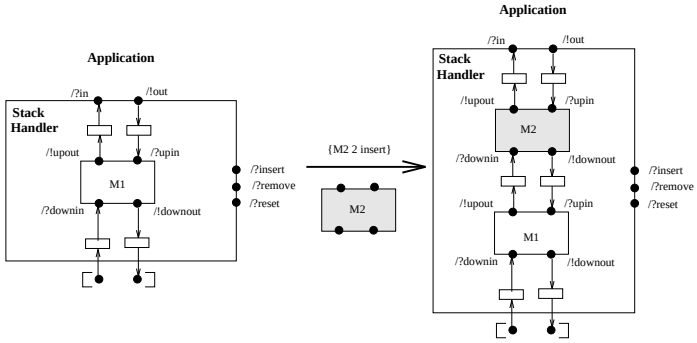


Figure 8: Pushing a layer on top of the protocol stack

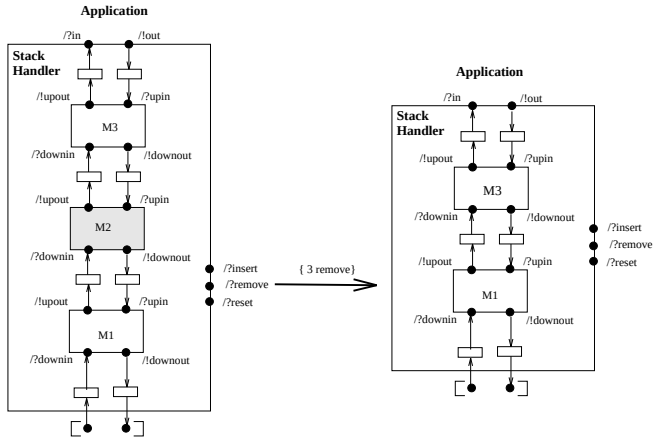


Figure 9: Removing the middle layer of a three layer stack

approach by allowing insertions and deletions of protocol entities at each level of the protocol stack and brings more power because the number of configurable entities is unlimited.

4.2 Configuration of a remote protocol stack

Now that we have seen how simple the reconfiguration of a local protocol stack is, let us see how a protocol stack residing in a remote host can be configured. To achieve this, the remote host must also be running a COMSCRIPT environment. COMSCRIPT allows an application running in one machine to access a COMSCRIPT environment running in another host. The remote COMSCRIPT environment is used as a server to which the application can send requests. Each request is itself COMSCRIPT code which is executed by the server when it reaches its destination.

Using the configuration mechanism presented above an application can configure a remote protocol stack. It even becomes possible to realize a data exchange between two hosts that do not have identical preconfigured protocol stacks. The following example shows how this can be done.

An application running on host A establishes a communication with a COMSCRIPT server (CS) on the remote machine B by opening two connections, one for control information and the other for data exchange. The control connection is used by the application to send requests to the remote server. The applica-

tion then downloads its own code to host B using the control channel. The execution of this code in the remote host results in the creation of a protocol stack which can then be used by the application to exchange data with host B.

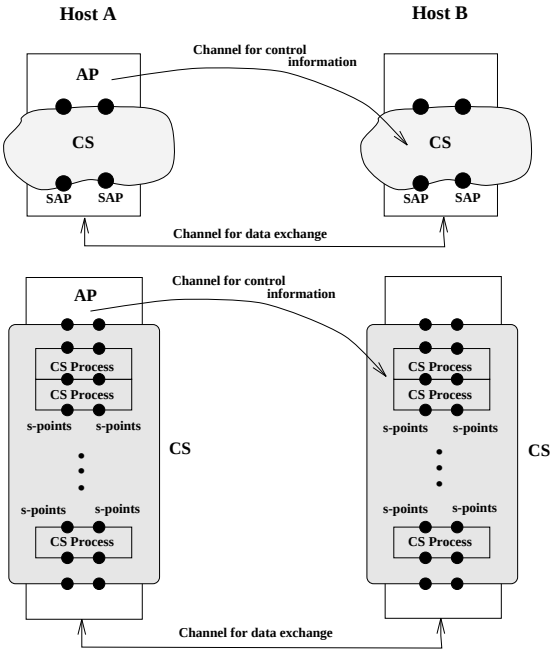


Figure 10: Configuring a remote protocol stack

The remote stack is controlled by the remote server which accepts requests from our application. Whenever the application needs a stack reconfiguration, it sends the appropriate request to the server, i.e. the code necessary to carry out the desired reconfiguration. The server initiates the reconfiguration in the remote host, while the application applies the same reconfiguration process to its local protocol stack (see figure 10).

5 Related Work and Current State

With the development of high-speed networks and distributed multimedia, more and more applications will be developed having different and sometimes contradictory requirements and needs. Developing a specialized, dedicated and optimized protocol for each application or class of applications is a complex task and surely not a satisfactory solution to this problem. The configuration of 'useful' protocols for their efficient use is thus given more and more attention: many research proposals have investigated (many numerous studies propose) an approach of general and flexible light-weight protocol entities which must be configured into a protocol stack or protocol graph which realizes the desired communication service. The proposed solutions vary greatly in the way that configuration can be done.

5.1 Related Work and Discussion

O'Malley and Peterson [OMall 91][OMall 92] propose the construction of a protocol graph, implemented on x-kernel, from a set of *micro* and *virtual* protocols. The protocol graph represents the set of protocols which can be derived from the composition of the different micro and virtual protocols. The actual protocol to be used is determined at the initialization time of the communication service, by choosing the most efficient 'path' in the protocol graph, i.e, the path which best fulfills the requirements of the application. This approach brings a limited degree of flexibility, but no further reconfiguration is possible after the set up.

The UNIX System V STREAMS approach [Strea 90] is another operating system approach which offers a framework for the composition of protocols. Protocol modules can be dynamically pushed/popped to/from a protocol stack at run time to realize the desired communication service. The main goal of this approach is to simplify protocol implementation; its ability to dynamically push/pop protocol modules at run time can be used for the configuration of a protocol stack.

Plagemann and Plattner [Plage 93] use a three layer approach in Da CaPo. The middle layer is responsible for configuring a protocol stack which meets the requirements of an application, using the resources available in the host and the network. A negotiation protocol allows the two communicating hosts to choose, according to the application requirements and resources available in each host, a common protocol for data exchange. If the monitor embedded in the middle layer detects a degradation of the quality of service beyond the limits specified by the application, the reconfiguration process is initiated again, and another protocol stack is chosen to ensure that the application's requirements are always satisfied. The bottom layer called the T layer represents the existing and connected transport infrastructures and is determinant in the number of protocol graphs which can be configured, while the approach of O'Malley and Peterson presented above is constrained to only one protocol graph.

In [Plage 94], Plagemann and al. propose CoRA, a heuristic for the stack reconfiguration in Da CaPo. This heuristic is based on the classification of protocol entities and their resource usage. The aim of the approach is to ensure that the requirements of the application are met in a very fast way.

All these approaches require that the same software exists in the two communicating hosts. They allow, to different degrees, the hosts to combine the existing software in the most efficient way. Moreover, the constructed protocol graph/stack remains a black box offering a desired service with the required quality for the application; however, there is no way the application can reconfigure the protocol stack itself, should it wish to change its communication requirements. All the work is done in a totally transparent way to the application, without interaction with it.

Among all these approaches, the CoRA/Da CaPo one seems to be the most flexible implementation of flexible protocol stacks. Indeed, it allows the building of a protocol stack which meets the requirements of an application and ensures that these requirements remain satisfied even if the availability of network or host resources degrades. However, some aspects of the implementation considerably restrict its power and its flexibility:

- All selectable protocol modules must be precompiled and preinstalled in the host: the number of possible protocol

combinations is thus limited. Moreover, adding a new protocol entity can not be done at running time. The module must be compiled, its"quality" must be evaluated in some way, and installed to be accessible by the configuration system.

- All the configuration work is done by the middle layer having as a consequence that the configuration process is very complex. This single layer contains the connection management module, the resource management module and the heuristic module. All these modules are complex and interact in a complex way to achieve stack reconfiguration.

- The approach detects *what* are the application's requirements and *attempts* to offer in, a transparent way, a service which fulfills those requirements. To achieve this, a single negotiation protocol and one heuristic mechanism are implemented in the middle layer of Da CaPo. Thus, it becomes difficult to extend this approach to cope with all possible applications. There is no way to specify the use of an alternative 'heuristic function' nor the possibility to use a different negotiation protocol in the configuration process.

COMSCRIPT, on the other hand, tries to avoid transparency. Indeed COMSCRIPT uses a quite different approach allowing the application to "explicitly" configure itself its communication needs as necessary. The application uses the power of the COMSCRIPT programming language to "do" the configuration of the protocol stack "how" it likes. There is no fixed protocol to negotiate the configuration process, nor the necessity for a heuristic module to reduce the complexity of the configuration task as in CoRA. Each application establishes its own policy for the reconfiguration task, in this way, all kinds of applications can be dealt with. Any application can implement its own heuristics for use in the stack reconfiguration if necessary.

5.2 Current State of COMSCRIPT

We have implemented a COMSCRIPT prototype to experiment with the concepts of flexible protocol stacks presented in this paper. We modified the public domain POSTSCRIPT interpreter GHOSTSCRIPT [Ghost 91][2] by removing all graphic related operators. The addition of concurrency has been inspired from the NEWS extension to POSTSCRIPT.

The aim of the current prototype is to gain confidence that we have made the right conceptual choices. Our implementation, being only a prototype, does not pay any particular attention to optimization considerations. Moreover, some important aspects which should be provided by a productive COMSCRIPT environment have not been implemented yet.

Currently, the COMSCRIPT environment runs in user space as a single process at the OS level that we will call the COMSCRIPT interpreter. This interpreter is responsible for the scheduling and the synchronization of all the COMSCRIPT processes running in the COMSCRIPT environment. All the COMSCRIPT processes can be seen as different "threads" of the COMSCRIPT interpreter. To schedule a particular process, the COMSCRIPT interpreter switches control to the appropriate thread. The scheduling used is non preemptive, i.e. a running COMSCRIPT process is not interrupted by the interpreter until the process explicitly releases control by executing the COMSCRIPT "pause" operator or until it blocks waiting for an event to occur.

Although the COMSCRIPT language contains the appropriate "hooks" to control access to devices such as the file system, the network etc., our prototype does not use them. No validation is done before executing COMSCRIPT code; a process executing invalid COMSCRIPT code enters the "error" state and is suspended by the interpreter. This behavior is not satisfactory in a community of COMSCRIPT environments where COMSCRIPT code can be downloaded for execution in a remote environment.

Experiments have been carried out on communities of COMSCRIPT environments. In such a community, a COMSCRIPT application running in some environment can "connect" itself to a COMSCRIPT server running in a remote host. It can send COMSCRIPT code which will be executed by the remote environment. Two types of servers have been implemented: one type of server listens to a TCP socket and is instantiated each time a client tries to connect to the COMSCRIPT service. The other server type is installed once and waits for UDP packets. Each UDP packet is assumed to consist of COMSCRIPT code which is simply executed. The main difference is that the second type of server is shared by many different applications while in the first case each client has its own server instance.

We will not close this section without a few words about performance considerations. Some preliminary measurements have been done using a simplified implementation of the FTP server in COMSCRIPT. This implementation uses the socket interface contained in the host's OS. What has been measured is the overhead introduced by the configuration process of COMSCRIPT and the interpretation approach. A 9MB file has been transferred (a) using the COMSCRIPT FTP server and (b) using the FTP server provided by SunOS. The measurements showed that the COMSCRIPT FTP server implementation has about 0.1% less performance than the implementation of the FTP server of SunOS. We conclude from this, that only little overhead is introduced by the configuration process of the COMSCRIPT implementation of FTP.

6 Conclusions

A primary goal of the COMSCRIPT approach is to make protocol stacks truly configurable at run time. We expect from this approach some solutions to interworking problems: applications can mix and match protocol functionality according to their requirements and network availability. Another contribution consists in the separation of a maximum of protocol logic from the rest of an application: application specific stack extensions can be run inside the stack instead of having to program protocols inside the application.

With the COMSCRIPT approach, an application can interact directly with its protocol stack by means of the COMSCRIPT programming language. The language itself is based on POSTSCRIPT

[2]We would like to thank Peter Deutsch for allowing us to use and modify the GHOSTSCRIPT sources.

but has been significantly extended with protocol specific constructions while all graphics related operators have been removed. Concurrent processes have been inspired by the NEWS [NEWS 90] extension to POSTSCRIPT. However, an event driven approach to processes has been privileged because it allows quite a direct translation between an FSM description of a protocol entity and its implementation in COMSCRIPT. Process interactions follow the process hierarchy and are achieved through s-points linked to gates. Self-containing protocol entities can be implemented and can be dynamically configured to meet the specific needs of a running communication software.

Experiments have been made with open communities of COMSCRIPT environments. In such a community, a COMSCRIPT application can access another COMSCRIPT environment running on a remote machine. It becomes possible to establish a data exchange between two hosts without having similar preconfigured protocol stacks installed (although we need a common transport infrastructure to exchange COMSCRIPT commands). Indeed, an application can download COMSCRIPT code to a remote COMSCRIPT environment which, when executed inside the remote host, will configure a whole protocol stack; afterwards, a classic data exchange can proceed between the two hosts, using the newly created communication environment.

Our first COMSCRIPT interpreter was implemented to experiment with flexible protocol stacks. Being an experimental tool, no particular attention has been given to optimization considerations, error handling and recovery. Memory management is rudimentary and security questions have currently not been addressed. For a tool which must be used in a real environment, all of these aspects need a more careful treatment. Nevertheless, the environment has proved to be suitable for the implementation of both low-level and high-level protocol entities. Successful experiments carried out with the environment suggest that the right choices have been taken at the conceptual level.

Acknowledgements

We would like to thank very much Mrs. Susan Warwick-Armstrong who corrected the text and suggested invaluable grammatical improvements.

References

[Adobe 90] Adobe Systems Incorporated, editor. *PostScript Language: Reference Manual*, Addison-Wesley, fifteenth edition, 1990.

[Cohen 86] Cohen N. H., *Ada as a second language*, McGraw-Hill, Inc. 1986.

[Comer 86] Comer D. E., *Internetworking with TCP/IP; Volume I; Principles, Protocols, and Architecture*, Second edition, Prentice-Hall International, 1991.

[Ghost 91] Deutsch L. P. (Aladdin Enterprises), *GhostScript—An Interpreter for the PostScript Language*, distributed under the GNU General Public License.

[Hopcr 79] Hopcroft J. E., Ullman J. E, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

[Hutch 91] 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.

[Muhu 92] Muhugusa M., Solana E., Tschudin Chr. F., Harms J., *ComScript — Implementation and Experiences*, Internal report, Université de Genève, Nov. 1992.

[NeWs 90] Sun Microsystems, Inc. NEWS *2.1 Programmer's Guide*, June 1990.

[OMall 91] 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.

[OMall 92] 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.

[Plage 92] 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.

[Plage 93] 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.

[Plage 94] 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.

[Rose 90] Rose M. T., *The open book: a practical perspective on OSI*, Englewood Cliffs: Prentice-Hall, Inc. 1990.

[Strea 90] Sun Microsystems, Inc. STREAMS *Programming Manual*, March 1990.

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

[Tschu 92] Tchudin Chr. F., Muhugusa M., Solana E., Tschudin Harms J., *ComScript — Concept and Language*, Internal report, Université de Genève, Nov. 1992.

[Tschu 93] Tschudin Chr. F., *On the Structuring of Computer Communications*, Ph.D Thesis no. 2632, Université de Genève, 1993.

Appendix: An Example of Simple Protocol Stack Configuration

We give in this section an example (see Fig. 11) that mimics the UNIX System V STREAMS approach, but in a simplified and simultaneously more powerful form. A 'stream' will be modeled by a COMSCRIPT process with a fixed set of external s-points. The s-points `/?in` and `/!out` are used to connect a client to the bidirectional data stream and represent the stream head. A synchronization with `/?push` allows to insert a new processing module, the `/_pop` s-point is used to remove the module at the top of the stack. The `/?reset` s-point is used to 'ground' the stream: an arbitrary device may be declared to be the bottom of the stack. The insertion of a new COMSCRIPT process which mimics a device is also permitted.

There are evident limitations in our 'implementation': there is no error treatment nor access control code built in, we also do not support parallel streams inside our COMSCRIPT process and multiplexors. However, the non-trivial but still compact example given below demonstrates the flexibility we gained by the fact that gates and s-points have become manipulable objects in COMSCRIPT.

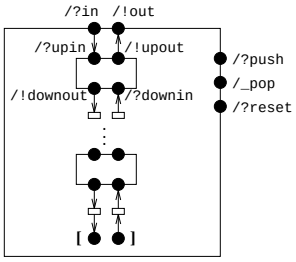


Figure 11: A Protocol Stack to Push and Pop Protocol Entities

For this example we do not use a FSM support – all event handlers and guards are directly defined inside the synchronization points. Rather than following the code line by line, we explain the basic organization of the data structures used and expected.

- The stack of modules will be represented by an array named `/stack` of fixed size. `/top` always points to the top element or is -1 if the stack is empty.
- The `reset` operation expects as exchanged data value a procedure, which must – once executed – return an array with two s-points. An array with these two s-points will be installed as the 'device driver' in the first stack position. Note that this procedure can either instantiate a device or create a new child process which acts like a device or installs one inside etc.
- The `push` operation will receive as data value the initialization code of the module to be inserted. Our main process will `fork` this code and create a new processing module which must install four external s-points with the well defined names `/?upin, /!upout, /?downin` and `/!downout`.
- Each stack entry corresponding to a processing module is an array with three elements: first we have the two upper external s-points followed by the module process itself.

```
1   /protocolStack {
2     clear
3
4     /getchildsync { exch /syncdict get exch get } def
5     /removetop {
6       % kills the top protocol entity and
7       % reestablishes the attachement
8       i detach o detach
9       stack top get 2 get kill
10      /top top 1 sub def
11      stack top get 0 2 getinterval dup { unlink }
12      forall
13      aload pop o exch attach i exch attach
14    } def
15    /linkwithtop {
16      stack top get exch get
17      3 1 roll getchildsync
18      0 creategate exch 1 index link link
19    } def
20    /attachhead { aload pop o exch attach i exch
21      attach } def
22
23    /stack 10 array def
24    /top -1 def
25
26    /?in 1 createsync dup /i exch def begin
27      /guard false def
28    end
29    /!out 2 createsync dup /o exch def begin
30      /guard false def
31    end
32    /?reset 1 createsync begin
33      /handler { /data get exec
34        top 0 gt { top { removetop } repeat } if
35        top 0 eq { i detach o detach } if
36        dup attachhead
37        stack exch 0 exch put
38        /top 0 def
39      } def
40    end
41    /?push 1 createsync begin
42      /guard { pop top -1 gt stack length top gt
43        and } def
44      /handler { /data get fork dup wait
45        [ 1 index /?upin getchildsync 2 index
46        /!upout getchildsync 3 index ]
47        dup stack exch top 1 add exch put
48        i detach o detach
49        0 2 getinterval attachhead
50        dup /!downout 0 linkwithtop /?downin 1
51          linkwithtop
52        /top top 1 add def
53      } def
54    end
55    /_pop 0 createsync begin
56      /guard { pop top 0 gt } def
57      /handler { pop removetop } def
58    end
59  } def
```

The interpretation of the code above is now straight forward. The `/?reset` event handler first executes the received device driver creation procedure (line 33), then removes any remaining old modules, and finally attaches the device to the external `/?in` and `/!out` s-points (line 36, procedure `/attachhead`). Pushing a new module is only possible if a device driver is installed and there is a free entry remaining on the stack (line 42). In this case we `fork` the received initialization code, extract the upper s-point and attach them to the 'stream head' (lines 44-49) and link the lower s-points via newly created gates to the former top s-points (procedure `/linkwithtop`).