Modular Communication Subsystem Implementation using a Synchronous Approach

Claude Castelluccia Walid Dabbous

INRIA, 2004 Route des Lucioles
BP-93, 06902 Sophia Antipolis Cedex, FRANCE
e-mail: Firstname.Lastname@inria.fr

Abstract

The lack of flexibility and performance of current communication subsystems has led researchers to look for new protocol architectures. A new design philosophy, flexible and efficient, referred to in the literature as "*function-based communication model*" is emerging and seems to be very promising. It consists of designing application-tailored communication subsystems adapted to the specific requirements of a given application. The flexibility of such a solution leads to very efficient implementations integrating only required functionalities.

In this paper, we propose a flexible model which uses a synchronous language to synthesize communication subsystems from functional building blocks. We prove the feasibility of our approach by implementing a data transfer protocol using Esterel, a synchronous language. Communication subsystem specifications in our model are very modular; they are composed of parallel modules, implementing the different functionalities of the communication subsystem, which synchronize and communicate using signals. The Esterel compiler generates from this parallel specification a sequential automaton by resolving resource conflicts. The design flexibility of our approach is demonstrated; modules are selected according the application requirements and compiled to generate an integrated implementation.

1 Introduction

1.1 Motivation

The lack of flexibility and performance of current communication subsystems has led researchers to look for new protocol architectures [CT90, HP88, SSS+93, OP91, ANMD93].

On one hand, the avalanche of new applications with new requirements and the rapid change in network technologies place new demands on communication services. Integrating all possible services into a monolithic general-purpose communication subsystem seems to be an unrealistic solution, because it would be practically difficult to implement and maintain.

On the other hand, the inefficiency of current layered architectures is admitted [SSS+93, BSS92, AP92b, HP88]. There are several reasons for this lack of performance, among them the replication of functions in different layers, the overhead of control messages and their inadequacy for the parallelization of protocol processing.

A new design philosophy, flexible and efficient, referred to in the literature as "*function-based communication model*" is emerging and seems to be very promising. It consists of dynamically developing application-tailored subsystems which adapt to the specific requirements of a given application. The flexibility of such a solution leads to very efficient implementations integrating only required functionalities.

The generation of "tailored" subsystems addresses others important issues and questions that have to be studied concurrently : (1) How to specify the application requirements, (2) How to select the right mechanisms to suit application needs, (3) How to generate and implement the "tailored" communication subsystem.

In this paper, we mainly focus on the third issue.

1.2 Goals

The idea of "function-based" communication subsystems has been extensively discussed in the research literature but, to our knowledge, no working and efficient experiments have been performed.

In this paper, we present a practical solution for combining elementary building blocks in order to generate and implement a "function-based" communication subsystem using synchronous languages. Our approach is general: it can be applied to all the communication functionalities required for a selected application.

Our final goal is to propose a modular and yet efficient way to implement communication subsystems. However in this paper, we mainly focus on the modularity aspects. We propose a highly structured and flexible communication subsystem architecture that generates integrated implementations. Our approach is based on the use of synchronous languages to synthesize communication subsystems from basic building blocks. In order to validate our approach, we implemented a data transfer protocol by the synthesis of elementary building blocks. We thus show that synchronous languages such as Esterel may be used to express the protocol control part, and to generate an optimal (in terms of task scheduling) automaton for the protocol execution.

The rest of the paper is organized as follows. Section 2 presents the benefits of the function-based approach. Section 3 briefly introduces synchronous languages and shows their adequacy for flexible subsystem synthesis. In section 4, we prove the feasibility of our approach by implementing a data transfer protocol, and analyze the results obtained. Section 5 offers conclusions based on our experience and suggests direction for future research.

2 Communication subsystems design

New applications (e.g. audio and video conferencing, shared whiteboard, supercomputer visualization etc.) with specific communication requirements are being considered. Depending on the application, these requirements may be one or more of the following: (1) high bit rates, (2) low jitter data transfer, (3) the simultaneous exchange of multiple data streams with different "type of service" such as audio, video and textual data, (4) a reliable multicast data transmission service, (5) low latency transfer for RPC based applications, (6) specific presentation encodings, etc.

The above requirements imply the necessity of revising the service model in order to fulfill the specific application needs. The applications must be able to specify the control mechanisms of a complete communication subsystem and not only the parameters of a single transport service. Recent research activities in this domain propose the synthesis of the so-called "communication subsystems", tailored to provide the service required by the application, from "building blocks" implementing elementary protocol functions such as: flow control, error control, and connection management (e.g. [SSS+93, AP92b, BSS92, OP91]). The synthesis of "fine grain" protocol functions should replace the coarse grain protocol choice (e.g. TCP or UDP).

However, these synthesis activities are focused on transport level functions, and do not consider the application synchronization and data presentation requirements. We argue that the integration of all the application communication requirements (including transmission control, synchronization and presentation encoding), in a single optimized protocol graph will result in increased performance. This is in line with the ALF (Application Level Framing) architecture proposed by Clark and Tennenhouse in [CT90].

This approach puts the application inside the "control loop". We still need to define how the application level parameters will be mapped onto network parameters and control functions in order to build specialized communication subsystems. The most promising approach in our opinion consists of the following combination: the use of a suitable language to express the application synchronization requirements (or the application dynamics) and the design and implementation of a tool (a compiler) that derives the complete communication subsystem automatically based on a set of optimized building blocks implementing the protocol functionalities. These functions should be combined in a way that minimizes the memory access cost.

In this paper, we mainly focus on the synthesis of the communication subsystem from functional building blocks; we are not detailing the application-communication integration part. In order to test the feasibility of the communication subsystem synthesis we used the Esterel language to describe the control part and to generate an implementation of a TCP-like protocol based on functional building blocks. The generality of our approach extends to the support of the complete subsystem's implementation and not only to transport functionalities.

3 Synchronous languages

In this section, we review the basic concepts of synchronous languages, and in particular those of Esterel, and show why they are appropriate to our final goal [Ber89].

Programs can basically be divided into three classes : (1)transformational programs that compute results from a given set of inputs, (2) interactive programs which interact at their own speed with users or other programs and (3) reactive programs that interact with the environment, at a speed determined by the environment, not by the program itself.

Synchronous languages were specifically designed for implementing reactive systems. The Esterel language is one example [BdS91, BG89]; others include languages such as Lustre, Signal, Sml and Statecharts.

Protocols are good examples of reactive systems; they can be seen as "black boxes", activated by input events (such as incoming packets) and reacting by producing output events (such as outgoing packets).

Esterel programs are composed of parallel modules, which communicate and synchronize using signals. The output signal of a module is broadcast within the whole program and can be tested for presence and valued by any other modules. This communication mechanism provides a lot of design flexibility, because modules can be added, removed or exchanged without perturbing the overall system. A module is defined by its inputs (the signals that activate it, they can potentially be modified by the receiving module), sensors (input signals used only for consultation, they can not be modified) and outputs (signals emitted). The inputs of a module can be the outputs of another one (modules executed sequentially) or external inputs (such as incoming packets). The design of an Esterel program is then performed by combining and synchronizing the different elementary modules using their input, sensor and output signals. Synchronous languages are used to implement the control part of a program, the computational and data manipulation parts are performed by functions, implemented in another language (C for example).
Data declaration are encapsulated, so that only the visible interface declarations must be provided in Esterel: type/constant/function/procedure names. These declarations can then be freely performed independently of the Esterel program design. It will be linked with the automaton generated in

the last phase, when executable code is produced. This separation between the control and data manipulation makes the integration of new implementation techniques such as *Integrated Layer Processing*[CT90, CJRS89] easier and allow to defer data handling to full-scale existing compilers. However the use of opaque declaration precludes reasoning on actual values prior to the linking phase, since the meaning of data operation is left uninterpreted.

Esterel makes the assumption of perfect synchrony : program reactions can not overlap. There is no possibility of activating a system while it is still reacting to the current activation. This assumptions makes Esterel programs deterministic, since behavior are then reproducible; the generated automaton can then be tested for correctness using validation tools [RdS90].

At compile time, the Esterel program is translated into a sequential finite automaton; the code of the different modules is sequentialized according the program concurrency and synchronization specifications.

However the synchronous approach cannot be considered as a stand-alone solution, principally because the synchronous assumption is not a valid one in the real implementation world. A so-called "Execution Machine" is required [AMP91]. This machine is aimed at interfacing the asynchronous environment to the synchronous automaton. It collects the inputs and outputs and activates the automaton only when it is not executing; the "synchronous assumption" is then respected.

4 Communication Subsystem implementation with Esterel: a study case

In this section, we describe the implementation of a data transfer protocol using Esterel. The specification of the protocol that we implemented is very similar to that of TCP protocol (we omit however the connection establishment and termination phases). We address how to design the building blocks and how to combine them to generate the required protocol. The goals of this case study were to test the validity of our approach and to give an insight on building-blocks contents.

Reusability and flexibility are our main goals here. The building blocks must be designed so that they are meaningful to the designers and so that changes in the protocol specification only induce local changes in the architecture and the code.

Communication subsystems are mainly structured in 3 parts [KTZ92]:

- the send module, that handles outgoing frames
- the receive module, that processes incoming frames
- the connection module, that handles connection variables and states

Each of these components can be decomposed into finer grain modules. The protocol functions are considered as atomic and their dependencies are defined by their input, sensor and output signals.

4.1 Building Blocks Description

We implemented this example incrementally in order to satisfy the modularity property that we are aiming for. We started from a simple protocol and added modules step by step until we accomplished all required functionalities. Following the precepts of Object Oriented Programming we followed the rule *one functionality-one module*. An overview of the whole subsystem is shown in figure 1 . For clarity purposes, only the principal modules have been described and displayed. Our data-transfer protocol is structured in 3 main concurrent modules :

The Send Module

This module is composed of 2 concurrent submodules:

The *Input_Handler* module receives data from the application. If enough space is left in the internal buffer, they are copied to it and a “Try-to-Snd” signal is broadcast, otherwise incoming data are unauthorized until an acknowledgment signal frees up some buffer space.

The *Emission_Handler* module transmits packets on the network. It can received two types of inputs: the ”Try-to-Send” input will try to send packets by evaluating the congestion window size, the silly window avoidance algorithm and the number of bytes waiting to be sent; it may then send one or several packets. The ”Send-Now” input will force the sending of a packet even though the regular sending criteria are not satisfied (this is used to acknowledge data for example). If the module decides to send packets, the checksum is performed and the header is completed.

The Receive Module

This module processes the incoming packets. It is composed of several submodules that can be executed concurrently:

When a packet is received, its header is scanned by the *Scan-Handler* module and all the header fields are broadcast.

The *Validate-Packet* module waits for “Checksum” and “Off-bit” signals to validate the incoming data (checksum and Off-bit field conformance tests are performed). If the packet is non-valid it is rejected at this point, otherwise the data are processed (the acknowledgment field is processed concurrently by the *Connection module*).

A test is performed (in the *Process-Path-Check* module) to decide whether the packet should follow the ”header prediction” path (*header-predictor* module) or normal path (*Normal-Process-Handler module*).
In the normal path, the data are processed and delivered to the application; the flow control parameters are updated.
In the fast path, two cases are possible:
(1)A pure acknowledgment packet is received. Buffer spaces are fred up and the sequence number of the next data to be acknowledged is updated. (2)A pure in-sequence data packet is received. The data is directly delivered to the application. The sequence number of the next expected data is updated.

The Connection module

This module is composed of the following submodules that are executed concurrently: the *RTT_Manager* module, the *Timer_Handler* module, the *Window_Manager* module and the *Acknowledgment_Manager* module.

The *RTT_manager* module computes the round trip time of the connection. When a packet is emitted, no other packet belonging to this connection is in transit and we are not in a retransmission phase then a timer is started. When this packet is acknowledged, then the timer is stopped and a new RTT value is computed.

The *Window_manager* updates (concurrently) the congestion window and the send window (corresponding to an evaluation of the space left in the receiving buffer of the remote host). When an acknowledgment signal is received (from the *Acknowledgment_Manager* module), the *Window_manager* increases the congestion window either linearly or exponentially according the slow-start algorithm, and the sending window is either update to the value of the "Win" field (emitted by the*Scan_Handler* module) or decreased by the number of bytes acknowledged. If a signal corresponding to a window shrink request (from the *Timer_Handler* module) is received, the congestion window is set to 512 bytes (value of the Maximum Segment Size).

The *Acknowledgment_Manager* module handles the acknowledgment information received from the incoming packet. If the acknowledgment sequence number (which corresponds to the last bytes received by the remote host) received is greater than the latest bytes sent or less than the last already acknowledged bytes then it is ignored, otherwise the acknowledged value is updated.

The *Timer_Handler* module manages the different timers. When a packet is emitted and no other packets are in transit, the Retransmission timer is set. When this packet is acknowledged and other packets are in transit it is restarted, otherwise it is reset. If the timer expires before the acknowledgment of this packet is received, retransmission and window-shrink requests are emitted.

All the modules just described have been implemented in Esterel and compiled into an automaton.

4.2 Application Programming

The protocol generated was then used to implement a file transfer application in order to validate the conformance of our protocol implementation with its specification. In our model, the protocol is implemented at the user level and linked with the application. The protocol is completely integrated with the application; actually the protocol is a task of the application. The application and the protocol can then share the same memory space. We implemented this interaction using Light-Weight Processes which allow the definition of multiple tasks within a single UNIX process and provide light-weight context switching between tasks.

Our application (contained in one process) is composed of 4 tasks :

- the application task : This is the task implementing the application. It notifies the protocol task when some data have to be sent.
- the I/O task : This is the task responsible for the incoming packets. When a new packet from the network is received, the I/O task notifies the protocol task that an incoming packet is waiting to be processed.
- the protocol task : This task processes the outgoing data (coming from the application) and the incoming frame (coming from the I/O task) according to the synthesized protocol. It performs what we called earlier the Esterel "Execution Machine".
- the timer task : This task manages the timers used by the protocols and the application.

All these tasks have the same execution priority, except the I/O task which has an higher one to avoid loss of incoming packet. A task with higher priority may preempt the others. Tasks with the same priority are executed on a round-robin discipline: a task is executed until is done or blocked on an I/O, the following task on the list is then executed.

All these tasks synchronize with each other. When a task has nothing to do, it sleeps. When another task needs its assistance, it is woken up. For example, when the protocol task has neither outgoing data nor incoming packet to process, it sleeps. When a packet arrives, the I/O task awakes it. The synchronization is then very natural.

4.3 Analysis of the Results

4.3.1 Esterel Compilation Phase

As explained in section 3, Esterel programs are composed of parallel modules, which communicate and synchronize using signals. An Esterel module gets activated on the reception of one or several signals (input signals), executes some actions and emits one or several signals (output signals). It uses local variables (invisible to the others modules) and communicates with other modules using signals (global variables).

The Esterel compiler generates from this parallel specification a sequential automaton by resolving resource conflicts. It assigns a code to every elementary action (e.g. assignment, test) of each module in the program text, and serializes these codes such that actions are always performed on the latest emitted signals within an instant. For example, if a *module A* needs the value of a signal *sig1* for its internal processing and *sig1* is modified and emitted in the same instant by *module B*, then Esterel compiler serializes theses two components such that the code modifying and emitting *sig1* be scheduled before the code using its values. If no schedule can be found, the program is rejected. This is what is called a causality error. Signals do not appear in the automaton. They are implemented as global variables, available to all modules that declare them as inputs or outputs. Emitting a signal consists of updating the corresponding global variable, reading a signal consists of accessing its value.

4.3.2 Modularity issues

The modularity of our approach has been demonstrated and illustrated by our case study. In fact we implemented our protocol incrementally and ran it at each step to test its correctness. As a result of this programming style, the program obtained is very modular; we can easily remove or exchange modules. For example by simply discarding the *Window_Handler* submodule we synthesize a sliding window data transfer protocol. If we set the window size to one packet, we implement a Stop-and-Go protocol. The window flow control can also be exchanged by a rate control mechanism by simply modifying the *Emission_Handler* module. The isolation of the different functionalities into separated and concurrent modules leads to very modular structures.

4.3.3 Performance issues

Performance was not the main goal of this paper. In fact, in this work we mainly focused on flexibility and modularity aspects.

However in this section, we show that the use of the synchronous language, Esterel, does not necessarily imply bad performances. To demonstrate this point, we compared the C code generated by the Esterel compiler with a handcoded BSD-TCP version of the University of California.

Since we implemented our case-study at the user-level on UDP, we believe that throughput or latency comparisons are not good performance indicators; the overhead due to the OS support is too difficult to evaluate. We therefore compare the structure of the C code of both implementations.

Following the analysis approach described in [CJRS89], we focused on the input processing analysis, which seems to be the bottleneck in protocol processing.
Our protocol is not a fully functional TCP, so it is dangerous to compare the input processing costs too closely and use comparison benchmarks such as instruction count numbers. We instead compare the sequence of actions executed on packet input paths of both implementations. We believe that this coarse grain comparison should give a fair indication of the performance of the code generated by Esterel.

In both implementation, when a packet is received its checksum and offset flag are verified, the protocol control block is retrieved, some variables are initialized and the test for the header prediction is performed. If this test is positive the header prediction algorithm is performed, otherwise the normal processing path is followed. In the normal processing path, the data are possibly trimmed and flow control variables updated.

We observed that the actions, their order of execution and the code executed on packet reception are very similar for both implementations. No specific overheads appear in the Esterel C code. Within a state, the code produced by Esterel is very compact.
However despite this similarity, some differences exist in the way these two programs are structured; The BSD-TCP implementation is "Action Oriented", while the Esterel implementation is "State Oriented".
In fact, the BSD code is composed of a sequence of actions preceded by their condition of activation. This sequence of <condition, action > is processed each time an input event occurs. For example, the condition corresponding to the header prediction algorithm is tested each time the input processing procedure is invoked. This is not always a desirable and time optimal feature.
The Esterel code is structured into states. All these states are distinct and only contain the actions possible in that particular state. For example, in the retransmission phase state the code corresponding to the header prediction algorithm does not appear, because no header prediction is possible in that state.

The receive paths in the Esterel program should then be more time efficient, because no unnecessary tests are performed. However the price to pay for this performance improvement is a code size increase.
The code size of the BSD-TCP is optimum, code sharing is performed whenever possible. In the Esterel program, no code sharing is performed . So if an action is executed in several states, its code is duplicated in each of those states, which can lead to very large compiled programs. As a matter of fact, our Esterel program which is about 10000 lines long produces an 11 states-automaton; the compiled code size of this automaton is about 100 kbytes, 4 times larger than the BSD one.

These preliminary results are very encouraging.
Our analysis shows that there is no inherent reason to believe that the Esterel code should not perform at least as well as the current one. In addition there is still the possibility of tuning the Esterel code (or modifying the Esterel compiler) to combine these approaches in a more optimal way.
We are now expecting to gain performance from the exploitation of the application-level knowledge.

5 Conclusions & Future work

The next generation communication subsystems need to be flexible to adapt to the rapid change of application needs and network technologies. In this paper, we proposed a framework using a synchronous language, Esterel, to synthesize communication from functional building-blocks. Esterel language programming style, which consists of breaking a program into components communicating via signals, provides the flexibility and efficiency we are aiming for. It also facilitates the integration of new design principles and implementation techniques, such as ALF/ILP, because of the separation made between the control and the data manipulation paths of a program. The viability and modularity of our framework has been established by our case study.

Our final goal is to propose a modular and yet efficient implementation of a communication subsystem framework. The modularity of our approach has been fully established by the work described in this paper. Efficiency aspects have now to be considered in more details. To achieve this next goal, we are currency addressing some important issues, among them :

- How to integrate efficiently new architectural and implementation concepts such as *Application Level Framing* (ALF) and *Integrated Layer Processing* (ILP) [CT90, GPSV91, AP92a]

- How to exploit application-level knowledge to design optimal communication subsystems

- What OS support is required for efficient and flexible subsystem design [MRA87, MJ93, HP88, MB92, MJ93, ANMD93]

.

These issues are not specific to our approach but common to high performance subsystem design framework. However we believe that if the right OS support is provided, modular frameworks, which synthesize tailored and optimal subsystems, should perform better than monolithic general purpose architectures.

Acknowledgment

We would like to thank Isabelle Christment and Ellen Siegel for valuable suggestions and comments.

Biographies

Claude Castelluccia received the B.S. degree from the University of Technology of Compiegne, France, in computer sciences in 1989 and the M.S. degree from Florida Atlantic University in electrical engineering in 1991. He is currently working on a Phd at INRIA (Institut National de Recherche en Informatique et Automatisme), France, on high performance communication protocol design.

References

[AMP91] C. André, J.P. Marmorat, and J.P. Paris. Execution machines for esterel. In *European Control Conference, Grenoble*, July 1991.

[ANMD93] Chandramohan A.Thekkath, Thu D. Nguyen, Evelyn Moy, and Edward D.Lazowska. Implementing network protocols at user level. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, September 1993.

[AP92a] Mark B. Abbott and Larry L. Peterson. Automated integration of communication protocol layers. Technical Report TR 92-24, Department of Computer Science, University of Arizona, December 1992.

[AP92b] Mark B. Abbott and Larry L. Peterson. A language-based approach to protocol implementation. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, August 1992.

[BdS91] Frédéric Boussinot and Robert de Simone. The ESTEREL language. Technical Report 1487, INRIA U.R. Sophia-Antipolis, July 1991.

[Ber89] Gérard Berry. Real-time programming: Special purpose or general purpose languages. In *Information Processing IFIP Conference, Elsevier Science Publishers, B.V. North Holland*, September 1989.

[BG89] Gérard Berry and Georges Gonthier. Incremental development of an hdlc protocol in esterel. Technical Report 1031, INRIA U.R. Sophia-Antipolis, May 1989.

[BSS92] Donald F. Box, Douglas C. Schmidt, and Tatsuya Suda. ADAPTIVE - an object-oriented framework for flexible and adaptive communication protocols. In *Proceedings of the Fourth IFIP Conference on High Performance Networking*, December 1992.

[CJRS89] David D. Clark, Van Jacobson, John Romkey, and Howard Salwen. An analysis of tcp processing overhead. In *IEEE Communications Magazine*, June 1989.

[CT90] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, pages 200–208, September 1990.

[GPSV91] Per Gunningberg, Craig Partridge, Teet Sirotkin, and Bjorn Victor. Delayed evaluation of gigabit protocols. In *Proceedings of the Second MultiG Workshop*, June 1991.

[HP88] N. Hutchinson and L. Peterson. Design of the x-kernel. In *Proceedings of the ACM Symposium on Communications Architectures and Protocols*, pages 65–75, August 1988.

[KTZ92] O.G. Koufopavlou, A.N. Tantawy, and M. Zitterbart. Analysis of TCP/IP for high performance parallel implementations. In *IEEE Workshop on the Architecture and Implementation of High Performance Communication Subsystems*, pages 576–585, February 1992.

[MB92] Chris Maeda and Brian N. Bershad. Networking performance for mocrokernels. In *Proceedings of the third Workshop on Workstation Operating Systems*, April 1992.

[MJ93] Steven McCanne and Van Jacobson. A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX Conference*, January 1993.

[MRA87] Jeffrey C. Mogul, Richard F. Rashid, and Michael J. Accetta. The packet filter: An efficient mechanism for user-level network code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, November 1987.

[OP91] S. W. O'Malley and L. L. Peterson. A highly layered architecture for high-speed networks. In *Protocols for High-Speed Networks II IFIP*, 1991.

[RdS90] V. Roy and R. de Simone. Auto and autograph. In *Proceedings of Workshop on Computer Aided Verification*, June 1990.

[SSS+93] D. Schmidt, B. Stiller, T. Suda, A.N. Tantawy, and M. Zitterbart. Language support for flexible application-tailored protocol configuration. In *LCN 93*, 1993.

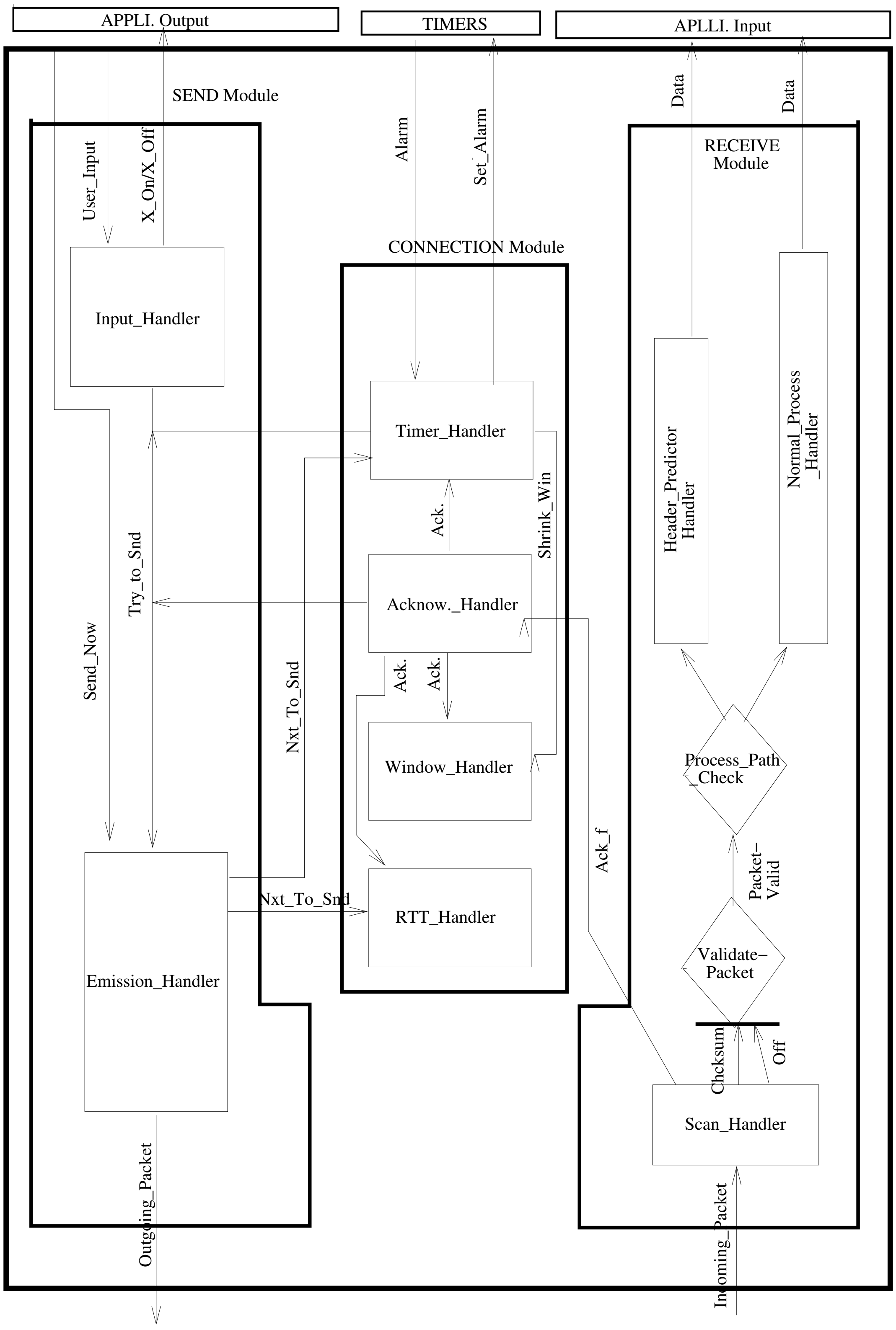

Figure 1: Protocol implementation Overview