A Comparison of Automatic Protocol Generation Techniques[†]

R. De Silva, L. Dairaine, A. Seneviratne, M. Fry

May 1996

UTS, Sydney
School of E. E.
P.O. Box 123, Broadway
NSW 2007 Australia

ENSICA
1, place Émile Blouin
31056 Toulouse Cedex
France

Abstract: *Due to the increasing complexity of applications and the availability of high speed networks, classical protocols have become the main bottleneck in communication systems. Although tailored protocols are able to respond to the needs of a given application, their development is expensive in terms of time and effort. An automatic protocol generation environment is most desirable. Two approaches currently used are the stub compilation and the runtime adaptive techniques. We have studied these two approaches and the behaviour of the resulting tailored transport protocols. Relative performance, comparisons and discussions about these two approaches are presented in this paper.*
Keywords: *Tailored Protocol, Automated Implementation.*

1. Introduction

The development of flexible and efficient communication protocols is an important step towards the realization of high performance distributed applications running on new high speed networks. Developments in high speed networking are influenced by two major factors: the increasing capability of end-systems and communication networks, and the diversity and dynamism of new applications. Recent networks such as FDDI and ATM allow for high speed transmission of digital data and have low latency characteristics, thus enabling the design of new types of applications such as multimedia. High level protocols, like TCP and TP4, have not evolved to take developments in applications and networking into consideration, resulting in them becoming a bottleneck in the communication system (Clark and Tennenhouse, 1990).

Protocols can be tailored to application needs and network conditions. Different approaches can be taken to tailoring protocols.

One method is to hand-craft the tailored protocol. This approach often leads to high performance protocols for a given application and network, but demands a lot of effort and time to design, implement and maintain. The alternative to hand-crafting protocols is to automatically generate application specific protocols. A number of research projects are currently in

[†]Published in Australian Computer Journal, Volume 28, Number 2, May 1996

progress, studying the possibility of automatically implementing a protocol based on a specification of application requirements and network resources. This paper reports a comparison of two approaches used for the automatic generation of tailored protocols.

The paper is organized as follows. In section 2 we study the evolution of protocol development, discussing the main techniques used in protocol design. Two automatic protocol tailoring approaches—namely compilation-based and runtime adaptive—will be studied in section 3. Section 4 presents our testing environment and results, both in terms of quantitative performance measurements and methodological aspects. Concluding remarks are given in section 5.

2. Tailoring Protocols in High Speed Environments

Protocols should be designed to handle the varying requirements of multimedia applications and to take into consideration the underlying network support. The main techniques that can be used to tailor a protocol are: the optmization of a current implementation, the development of new mechanisms, and the development of new protocols.

Implementation optimizations can be applied to existing protocols to improve performance. Such techniques do not change the functionality of the protocol but simply reduce the processing cost of the protocol.

An alternative technique is to develop new mechanisms. This is often achieved by reengineering an existing protocol to reduce a bottleneck that arises out of a particular characteristic of the operating environment. These implementations often present a large number of options. Only a limited set of these options are used by a particular application.

The third technique is the development of new protocols built to exactly match a given application's requirements. This technique would normally also utilise the previous two techniques discussed above. The main drawback of this method is the development cost for a specialized protocol. Automatic protocol generation permits a high level of tailoring without the high costs of time and manpower.

3. Automated Communication Protocol Generation

Automated tailored protocol generation can be achieved by defining, for a given application, the functionality the protocol should provide and the associated mechanisms. The overall process involves three basic tasks : specification, selection and synthesis.

During the specification phase, the application developer lists all information that characterizes the application and the environment. This list should contain all the relevant information needed by the Automated Communication Protocol Generator (ACPG) to create the appropriate tailored protocol. Such a specification should contain, for example, the structure and characteristics of the data being transfered, ordering constraints, reliability, timing criteria, possibilities of having self contained data packets and possible integrable processes.

The selection of mechanisms is the second phase of the automated process. Using information that characterizes the application, the ACPG decides the overall functionality required to build the tailored protocol. If the protocol is intended to be dynamic, then decisions on when to switch protocol functionality can be decided at this stage.

Finally, the synthesis phase involves the implementation of the protocol. It has been shown that, for efficient implementations, the principles of Application Level Framing (ALF) and Integrated Layer Processing (ILP) should be adopted (Clark and Tennenhouse, 1990). The implementation can be static or dynamic. Dynamism can be introduced by using dynamic linking of protocol functions as required, or statically implemented in a state machine which

changes states when changes in protocol functionality are required. The latter can result in "code bloat" if a large number of dynamic states are defined.

Currently there are a number of projects being conducted on automated approaches. These include DaCapo (Vogt, Plattner, Plagemann and Walter, 1993), F-CSS (Zitterbart, Stiller and Tantawy, 1993), ADAPTIVE (Schmidt, Box and Suda, 1992), the Runtime Adaptive Approach (Universal Transport Service) (Richards, 1995), and the Stub Compiler (STRL) (Castelluccia and Dabbous, 1994). The first four models provide runtime configuration while the fifth model provides configuration at compilation time. The first three approaches tailor the whole communication environment while the remaining two create application tailored protocols. The rest of this paper is devoted to these two last techniques, which have been selected to contrast the runtime and compilation approaches.

3.1. Runtime Adaptive Approach

The runtime adaptive approach has been developed at UTS (University of Technology, Sydney) in Australia. As shown in Figure 1, the conceptual architecture of the runtime adaptive model follows the general model for ACPG.

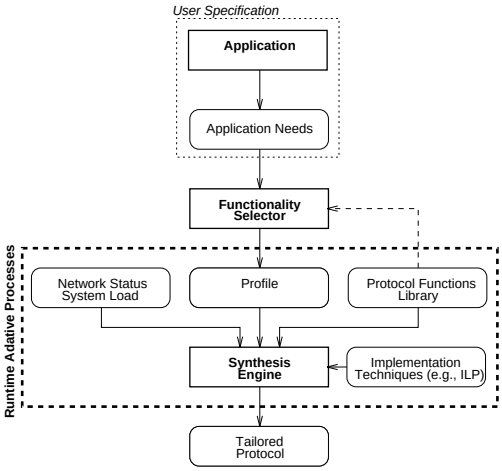

Figure 1: The runtime adaptive model.

The realisation follows the 3 phases as defined above. At runtime, the application indicates its requirements to the *Functionality Selector*, for example via a QoS management entity. The *Functionality Selector* then determines the protocol functions that will be required to satisfy the application's requirements. A profile is then generated that indicates to the *Synthesis Engine* the optimal choice. The *Synthesis Engine* then takes into account the environment status (e.g. network status and system load) and chooses the appropriate mechanisms to provide the requested functionality. It then uses implementation techniques to optimize the transport system's performance. In addition, the *Synthesis Engine* continuously monitors the status of the network and host system and, where possible, dynamically chooses the protocol mechanisms that will best suit the given conditions.

The runtime adaptive model differs most from the stub compiler model in its inherent dynamicity. The configuration engine creates/modifies the protocol as necessary following changes in any of its inputs (network status, profile, etc.). Dynamicity is achieved by dynamically linking in and out the appropriate protocol functionality as determined by the *Synthesis Engine*.

The adaptive approach takes into account the ALF and ILP principles, although our implementation currently does not apply ILP.

3.2. Stub Compilation Approach

The stub compilation approach has been developed at INRIA (Institut National de la Recherche en Informatique et Automatique) in France. This approach is a preliminary step in the development of a new generation of remote procedure call models (Diot, Chrisment and Richards, 1995). In this model, a distributed application can specify its own communication requirements, which are to be associated to a dedicated communication system. This is realized by means of an application specification, which is used by a protocol compiler to integrate communication facilities with the application by generating client and server stubs.

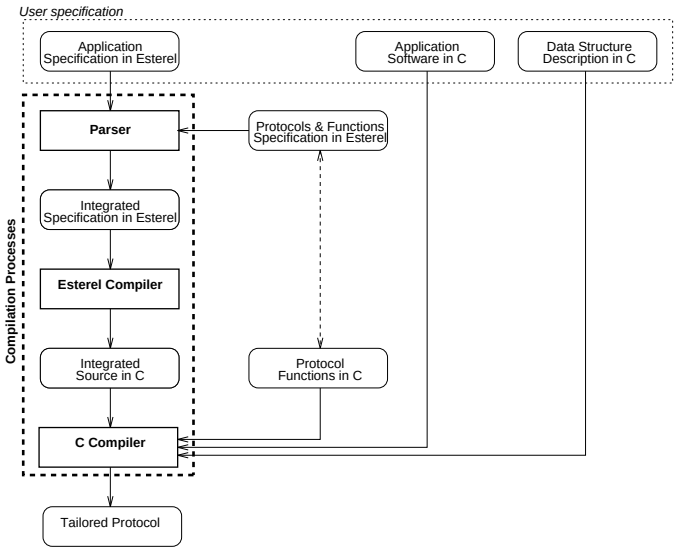

Figure 2: The stub compiler model.

The specification step is achieved using both Esterel and C code. Esterel (Berry, 1992) specifies the control and synchronization aspects of the communication, while data structures and application software are directly coded in C. More specifically, the Esterel specification describes the application's behavior, and the level of reliability required for the transmission of the Application Data Unit (ADU) – the smallest unit of data that the application can process to completion. The services required are described by a set of predefined Esterel signals expressed in the specification, such as *selective_ retransmission*, *flow_ control*, *checksum*, etc.

The selection step is realized through the parsing of the specification as shown in Figure 2. From the Esterel description of the application's behavior, the parser extracts possible synchronization points and any parallelism that exists between the different modules that compose the description. These synchronization points will be used in the synthesis stage to construct an efficient implementation. The predefined Esterel signals present in the application specification are extended by the parser to integrate the requested communication facilities. The result of this step is the integrated specification, still expressed in Esterel.

The new Esterel specification is then compiled through an Esterel compiler which produces C-code for a finite state machine defining the different protocol states. This C-code is then combined with the appropriate application code and the appropriate protocol functions to create the executable application. The C and Esterel compilers form the implementation stage of the ACPG model.

The main strength of the stub compiler model is that it provides a high degree of "tailorability". The main reason for this is the use of Esterel - a formal language, which allows for an accurate description of the application's requirements. The stub compiler model can realise Application Level Framing. One weakness of this model is that the stub compiler is

based on a state machine which changes state to provide different services.

Currently, this model does not properly consider dynamic adaption as it fails to take into consideration external factors such as network and system loads.

4. Experimentation

To compare the runtime adaptive and stub compiler approaches, a common application was selected and two tailored protocols were generated. Quantitative and qualitative measurements of the resulting protocols' performance, and their development methodologies, were then evaluated in different environments.

4.1. The JPEG Image Server

The JPEG Image Server (JIS) is a client/server-based application allowing the visualization of images stored at the server. JIS constitutes a system that is sufficiently simple to be ported quickly to several distinct platforms, and easily modifiable to run over different implementation environments, but is still sufficiently complex that it can benefit from application specific tailoring.

The main property of JIS that can be exploited by tailoring is its *partial ordering* of the JPEG image elements. The client does not require the ordered reception of image data messages. The only real ordering requirement is to receive the JPEG specification tables before the image data (these tables specify parameters of the compression such as quantization). Then, during the data transfer phase, the the communication protocol can present the image data blocks to the application regardless of the order in which they are received (this assumes that the application has structured its transmission according to ALF principles). Thus, JIS can be realized using two different protocols, namely one for the specification phase and another for the data transfer phase.

The runtime adaptive and stub compilation ACPGs were used to develop protocols tailored to JIS. The following sections discuss the results obtained from the comparative study of the two automated approaches. The protocols developed by the runtime adaptive and stub compiler ACPGs will be referenced as UTS and STRL respectively. The performance of the protocols is presented in terms of qualitative issues and quantitative measurements. Only prototype versions of both ACPGs are currently available. These were used to obtain the experimental results.

4.2. Qualitative ACPG Issues

The two major aims of an ACPG are to offer easy and rapid development of protocols and to produce high performance implementations. These two issues are discussed in this section.

Protocol Development Issues

>From the user point of view, the runtime adaptive and stub compilation techniques take completely different approaches to the development of tailored protocols and provide different interfaces. The former is based on a protocol configuration using its own interface, which is currently based on a mix between a BSD socket interface and function-call interface. The latter provides a formal protocol description using both the Esterel and C languages.

The two ACPGs also differ in the manner of expressing application needs and in the management of protocol adaptations. With the compiler-based ACPG, the JIS application defines a protocol using an Esterel specification indicating the two different stages in the data transfer and their corresponding functionality. The parser and compiler stages of the ACPG then

creates a finite state machine which will change states depending on input signals from the application. Changes of state occur at set points in the data transfer and are agreed at compilation time. We therefore do not require a meta-protocol to signal changes at runtime between the client and server.

Considering the JIS client side, the Esterel code is based on two main parts specifying the two phases. The first part basically waits for two specification tables, using a parallel construct. When both tables have reached the client, a signal causes the state machine to terminate the specification phase and begin the data transfer phase. This enables out-of-order reception of the different image data messages. From the user point of view, the image data ADUs are displayed as soon as they reach the client, resulting in a possible mosaic-style effect as the image is being constructed.

The runtime adaptive model protocol requires the application to request functionality through an interface. With the current prototype of the model, the application specifies the required functionality through the use of function calls. Since functionality changes occur at runtime, and because the protocol has no prior knowledge of any changes, we require a meta-protocol to coordinate the changes between the client and server. UTS currently realises the meta protocol within the packet header. As a result, there will normally be a round-trip delay before functionality is switched.

To improve performance, instead of defining two stages of unordered transfer separated by a synchronization point, the UTS protocol was defined as an ordered protocol for the first stage of the transfer (i.e. the specification stage) followed by an unordered stage for the image data. The change from ordered delivery to unordered delivery is a reduction in service. Hence it is unnecessary to stop the transmission of data until the functionality has changed. In contrast, if we were to change from unordered to ordered delivery, we would have to guarantee that every packet the user specified as ordered arrived in the correct sequence. When changing in the reverse direction we provide a higher level of service while the change in functionality occurs.

The user also specifies through a function call that they wish to use ALF. This ensures that the data written by the application through the socket interface will be handled as an integral unit by the communication system. This is required to enable unordered delivery of ADUs.

In both approaches the effort required by the user is minimized to defining the requirements of their application.

Resulting Code

Experiments were carried out on DEC 5000/240 Workstations using a dedicated Ethernet network. Experiments were executed over two different operating systems, namely Ultrix v4.3 and the Mach 3.0 micro-kernel.

With the Mach micro-kernel, TCP can be implemented using two possible methods. The first is the Mach 3.0 UX server, which provides a kernel level implementation of TCP/IP within the Unix Server. The second is a user level library implementation of TCP (Maeda and Bershad, 1992).

UTS is currently based only on the Mach 3.0 user level library implementation. Therefore the results for UTS were only measured for this implementation. The results for TCP (which was used as a benchmark) and STRL were obtained using all three platforms: Ultrix 4.3, Mach 3.0 using the UX server and Mach 3.0 using the user level socket library.

The code sizes of the resulting protocols varied. Table 1 shows the sizes of TCP, STRL and UTS in the different environments.

TCP code is optimal in all situations. The STRL code is significantly larger. This is because it includes its own protocol that runs over UDP/IP. This is in contrast to UTS, which

replaces TCP. Hence for user level protocol implementations, UTS is the same size as TCP, while STRL runs above the user level implementation of UDP/IP adding another level of protocol functionality. The current version of UTS simply uses internal switches to change functionality. The final version will dynamically link the required functionality and hence the overall code size is expected to be smaller.

Table 1: Architecture-oriented sizes in kilobytes for the client and server executables.

Client Sizes	**TCP**	**STRL**	**UTS**
Mach 3.0 SC	539.6	819.4	539.6
Mach 3.0 UX	285.3	546.2	*n/a*
Ultrix 4.3	278.4	547.3	*n/a*

Server Sizes	**TCP**	**STRL**	**UTS**
Mach 3.0 SC	510.6	802.6	510.6
Mach 3.0 UX	256.2	445.8	*n/a*
Ultrix 4.3	249.4	446.0	*n/a*

4.3. Quantitative Performance

Three experiments were carried out to capture how well the implementations of the JIS tailored protocols performed. The first compared the behavior of the three protocols with various ADU sizes. The second compared STRL and TCP performance across the different platforms. The third showed the behavior of the protocols when transmission errors are introduced.

Comparison of the Three Protocols

The Experiments were carried out on the Mach 3.0 Kernel using the user level implementation of TCP/UDP/IP for all the protocols. We experimented using two MCU ADU sizes (MCU = Minimum Coded Unit, an atomic unit of compression/decompression). One was the maximum size for a MCU ADU such that any ADU can be contained without IP fragmentation (i.e., 640 bytes). The other was the maximum size for a MCU ADU without it begin fragmented at the MAC layer (i.e., 1460 bytes). Table 2, shows the results of the first experiment.

Table 2: Protocols Throughput under Mach 3.0 micro-kernel System with Maeda's Socket Code in Kb/s.

MCU ADU Size	**TCP**	**STRL**	**UTS**
$\leq$ **640 bytes**	3531	2428	3389
$\leq$ **1460 bytes**	5607	5507	5188

The overall results between the three protocols are very similar for both MCU ADU sizes. The hand-coded version of TCP and the two ACPG protocols show the same range of through-

puts in the same Mach 3.0 socket code library environment. This suggests that automatic protocol implementations give good implementation results and yield high performance.

The throughput results attained using the smaller MCU ADU size are low. This is because of inefficient usage of the underlying network. STRL is based on UDP, and hence a large number of UDP packets are sent across the network. Similarly, with TCP and UTS (which is based on TCP) the small sized MCU ADU are buffered to try and fill a packet, hence introducing delays. When we increase the MCU ADU size to the maximum for the network, we notice that there is a large increase in the throughput.

Cross Platform Comparisons

The second experiment compared the performance of the STRL protocol against TCP across the different platforms. The maximum size of MCU ADU was used for all experiments to achieve the best results. Table 3 illustrates the comparative behavior of TCP and STRL across the different environments. With the exception of the Mach 3.0 UX platform, where there is a drop in performance for the STRL protocol, the two protocols have very similar performance.

Table 3: Cross Platform Throughput with Maximum MCU ADU Sizes in Kb/s (1460 bytes)

	TCP	**STRL**
Mach 3.0 SC	5607	5507
Mach 3.0 UX	5192	3681
Ultrix 4.3	8972	8714

With Mach for both socket code (SC) and UX server (UX), the results are at least 2Mb under those of Ultrix. This can be explained by the user level location of the transport protocols and the better implementation of the Ethernet device driver in UX (Witana, 1994).

Performance under Erroneous Conditions

The selective retransmission and ALF architecture of the STRL protocol leads to a very good behavior in presence of transmission errors introduced by the network (both packet loss and bit error). Missing packet are requested when the receiver finds an out of sequence pattern, but this does not block the sending application. Thus the overall throughput can be estimated assuming that the transmission delay is just the time to send ADUs and retransmitted ADUs.

The same experiment with TCP gives poor results. TCP interprets packet loss to be caused by congestion, and therefore reduces transmission rate to allow the network to recover. TCP's slow start mechanism is responsible for this behavior.

5. Concluding Remarks

In this paper we have discussed the benefits of tailoring protocols to application needs. Currently, tailoring techniques are costly in terms of time and effort, and require highly skilled personnel. Automated protocol generation aims to keep the benefits of tailoring without the associated costs of development.

A number of different automated approaches for Automated Communication Protocol Generation have been proposed, of which we focused on two, namely runtime adaptive and stub

compilation. These two ACPGs were used to build protocols tailored to a JPEG Server Application. Both approaches were able to create protocols with tailored functionality, as would be done by a hand-crafted implementation.

Our results show that the two tailored protocols have similar performance although the implementations differ. Both protocols provide similar functionality with the exception of error recovery mechanisms. The main difference that currently exists between the two approaches is the method by which functionality is requested and supported.

Comparisons between the automatically generated protocols and a hand-coded TCP implementation resulted in the same range of throughput over a number of platforms. This indicates that the automated approach does not interfere with the quality of implementation of the resulted protocols. In addition, some engineering concepts such as ILP could be taken systematically into account, yielding better implementations. In conclusion, it has been shown that automated protocols are a viable solution for the future.

6. Acknowledgments

The authors would like to thank Isabelle CHRISMENT from INRIA (France) for her assistance in the JIS application. Laurent DAIRAINE thanks INRIA for its financial support.

References

BERRY, G. (1992), *The Esterel synchronous programming language: Design, semantic, implementation*, Journal of Science of Computer Programming, 19.

CASTELLUCCIA, C. AND DABBOUS, W. (1994), *Modular communication subsystem implementation using a synchronous approach*, Internal report, INRIA, Oct.

CLARK, D. AND TENNENHOUSE, D. (1990), *Architectural considerations for a new generation of protocols*, in Proceedings of ACM SIGCOMM Conference (Communication Architecture and Protocols), USA, pp. 200–208.

DIOT, C. CHRISMENT, I. AND RICHARDS, A. (1995), *Application level framing and automated implementation*, in IFIP HPN international conference, Sept.

MAEDA, C. AND BERSHAD, B. (1992), *Networking performance for microkernel*, internal report, Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA, Mar.

RICHARDS, A. (1995), *The Universal Transport Service, an Adaptive End-to-end Protocol Analysis and Design*, Ph.D. thesis, University of Technology, Sydney, Dec.

SCHMIDT, D. BOX, D. AND SUDA, T. (1992), *ADAPTIVE: A flexible and adaptive transport system architecture to support lightweight protocols for multimedia applications on high speed networks*, in Proceedings of the Symposium on High Performance Distributed Computing Conference, Amsterdam, Sept.

VOGT, M. PLATTNER, B. PLAGEMANN, T., AND WALTER, T. (1993), *A runtime environment for Da CaPo*, in INET.

WITANA, V., *TCP/IP protocol performance* (1994), Technical Report, University of Technology, Sydney.

ZITTERBART, M. STILLER, B. AND TANTAWY, A. (1993), *A model for flexible high performance communication subsystems*, IEEE Journal on Selected Areas in Communications, 11, pp. 507–518.