Standardized Protocol Interfaces

Gerard J. Holzmann

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

A traditional protocol implementation typically consists of at least two distinct parts, a sender and a receiver. Each part runs on a distinct machine, with the implementation provided by a local expert. At best, the two machines are of the same type and the protocol implementations are provided by the same person. More likely, however, the machines are not of the same type and the implementations of the two halves of the protocol are provided by two different people, working from an often loosely defined protocol specification. It seems almost unavoidable that the two implementations are not quite compatible.

In this paper we consider an alternative technique. With this method, one of the two implementors can design, formally validate, and implement all the relevant protocol parts, including those parts that are to be executed remotely. Each communication channel is now terminated on the receiving side, by a single standard protocol interface, which can be called a Universal Asynchronous Protocol Interface or UAPI.

Though it is likely that the UAPI is most efficiently implemented in hardware, it can also trivially run as a software module, e.g. under a standard UNIX® operating system (in our case under 10th Edition Research Unix). This paper introduces the concept of a UAPI and explains how the sample software controller was constructed.
[Bell Labs Technical Memorandum, March 26, 1991; submitted for publication.]

October 15, 1992

Standardized Protocol Interfaces

Gerard J. Holzmann

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

1. INTRODUCTION

A universal protocol interface is a scheme in which the initiator of a communication can define in detail what the rules of the communication shall be, irrespective of any previous knowledge about those rules at the remote side of the connection. In particular, the rules can be chosen to conform to known international standards for data exchange, such as CCITT Recommendation X.25, or they can be based on a protocol known only to the initiator of the connection. To make this work, all we need is a standardized language for defining general data transmission and manipulation, and a device that interprets that language, which we shall call a *UAPI.*[1] The UAPI's are used to interface host computers to data links. In their least interesting mode, these UAPI's can be used transparently when connecting to remote hosts that do not speak the new language. In that mode, the local host is accessible as before, through the use of predefined hard−coded protocol implementations, that must be trusted precisely as they are provided. For remote hosts that do speak the new interface language, however, the restriction to the hard−coded protocols can be side−stepped, and any trusted set of protocol rules can now be negotiated on the fly, as deemed necessary.

At first sight, there seems to be one disadvantage, and two potential advantages to the usage of universal, or non−protocol dependent, protocol interfaces. The advantages are the increased reliability and flexibility of a communication link. The increased reliability comes from the fact that a sender no longer has to trust an unknown remote receiver to be precisely compatible with its operation. The disadvantage, however, can be a loss of performance in comparison with the traditional hard−coded protocols. The objective of this paper is to outline a basic scheme for defining a language and for programming a UAPI. The precise details of the language and the implementation given here are meant as example only, not as the final word on the possible design choices that could be made. The example implementation will, however, allow us to study the potential advantages and disadvantages of the UAPI scheme quantitatively.

In Section 2 we compare the idea of a universal protocol interface with a range of other ideas that have been pursued before. Section 3 considers the basic structure of a sample UAPI controller. Section 4 shows an example of a small application protocol that was converted into the language accepted by this controller. Section 5 gives the result of some performance tests with this implementation, and Sections 6 and 7 summarize the proposal and the results obtained. Two Appendices contain more detailed listings of the relevant parts of the controller and the sample protocol that was implemented on it.

Sections 3 to 5 can safely be skipped on a first reading. We begin with a simple example to explain the basic setup.

ON THE FLY PROTOCOL DEFINITION

Consider an arbitrary data transfer protocol, that is unknown to the receiving machine when a data transfer connection is established. The definition of both the sender and the receiver part are stored in the applications program at the originating side. In the initial connection setup handshake the originating side transfers the required control tables for the remote receiver into the remote UAPI, as illustrated in Figure 1.

1. A UAPI resembles a UART both in name and in function.

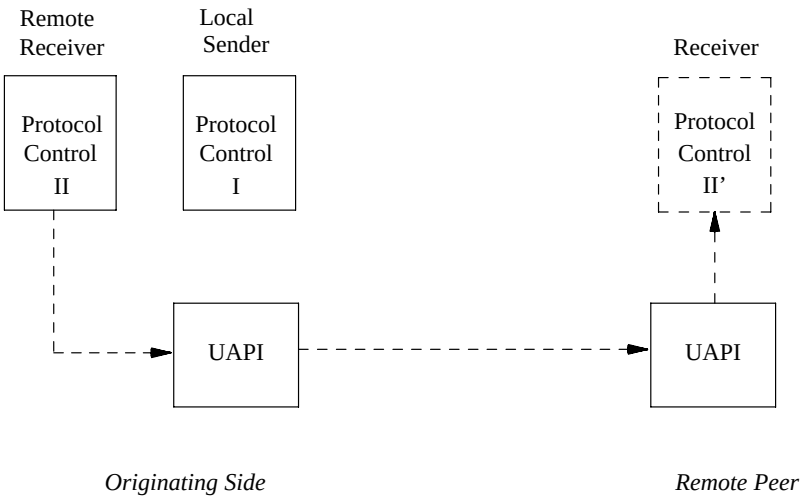

Figure 1 — Connection Setup Phase

This protocol setup is typically completed in one or two data transfers, depending on the complexity of the protocol selected. The originating side then switches to normal operation by loading the matching protocol control tables for the sender into the local controller and starts the data transfer (Figure 2).

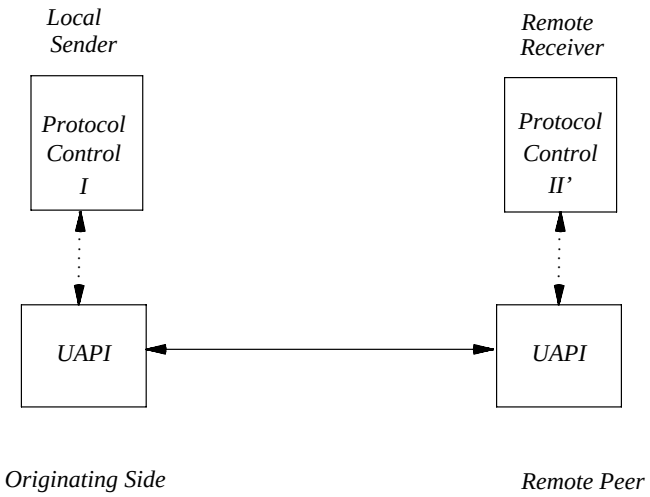

Figure 2 — Data Transfer Phase

At the end of the data transfer, both UAPI's fall back into their default controller bootstrap mode, ready to accept the new protocol controllers for either incoming or outgoing data transfers.

2. RELATED WORK

Variations of the idea of a universal protocol interface are fairly easy to find. Before going in to too much detail on the way in which a UAPI could be realized, it may be helpful to explain in more detail what the fundamental difference is with each of these related ideas. We will first give a broad overview.

- The most obvious related idea is that of a *downloading* protocol for bootstrapping a remote device. The standardized language for defining protocol behavior is then the assembly language of the remote device, and the protocol interface is a standard serial or parallel link.
- A more closely related idea was applied in, for instance, the Postscript language. The idea here is to *encapsulate the instructions* for processing data within the data itself. A data file thus becomes a

program that can be executed to obtain the desired effect, such as the printing of a text−file. Troff text files, similarly, can be made ‘‘self−formatting’’ by writing them as UNIX® shell−scripts, with the text re−directed internally into the required preprocessing commands. A similar idea exists in image processing. Large image files can be compressed with special filters, and the decompression algorithm can be encapsulated into the data in such a way that the image file unpacks and uncompresses itself when executed. The idea can obviously be used to minimize the time needed to transmit arbitrary large data files.

- In the area of network protocols, many examples can be found of protocol functions, or *grades of service*, that can be set on demand by a remote party during the connection setup phase. Fraser’s Universal Receiver Protocol (URP) embodied that idea [3], for instance by allowing the sender to select the specific type of error control and flow control to be used during the actual data exchange. Remnants of this idea can be traced to many modern protocols, e.g. Chesson’s XTP [7].
- A next category of related work includes, for example, Ritchie’s work on *I/O streams* [6]. The goal there is to build a uniform, standardized interface for protocols used within an operating system. The streams idea provides a clean interface definition for combining stream protocols in arbitrary ways.
- Probably the most closely related previous work in this area is that of Falcone’s *network command language* NICL [e.g. 2], and Tschudin’s description of ‘communication messengers’ in [8]. Both ideas define how services can be defined by a local host, transmitted in a standardized language, and executed as a process on a remote host.

UAPI’s

The standardization of protocol interfaces can be compared to the idea of standardizing the physical characteristics of electric outlets. This feature allows us to plug any device that has the right interface into any outlet that conforms to the standard, quite independent of the location or the maker of the outlet. The maker of the outlet does not have to know anything about the precise functionality of the devices that will use its services. Although we have gotten quite used to this, the same is not true for computer interfaces, and none of the above cited ideas can make it true, nor aims to do so.

In computer networking today each different device (protocol) is allowed to define its own non−standard interface. There are different outlets, so to speak, for toasters and clocks. A clock (e.g. an XTP sender) cannot be plugged into (communicate with) an outlet that is defined for toasters (e.g. an X.25 receiver). With this analogy in mind, it is easy to see the fundamental differences between the UAPI and the ideas summarized above.

Downloading Data —Downloading protocols are asymmetric. They typically assume a master−slave relationship between devices, where the master device loads a specific data format into the slave. It would be quite rare for the slave device to take over control and download new software back into its master once it has started up, because it wasn’t pleased with the original binaries in its master. The UAPI has a different purpose. It is intended as an interface between peers, that does not assume any specific machine architecture or binary format. It allows for a general negotiation between peers about the specific way in which they shall interact from that point on.

Encapsulated Data —The encapsulated instructions of languages such as Postscript are statically stored and are not meant to allow for any negotiation about data exchanges between senders and receivers. The formats used here can be understood as standardized for the correct interpretation of static data. It is much harder to understand it as a format for the exchange and manipulation of arbitrary data, irrespective of its meaning.

Grades of service —The difference with idea of selecting grades of service on the fly within protocols such as URP or XTP is quite important. An option within a given protocol is of course no more than just that. It can change an aspect of a protocol behavior, but it cannot change the protocol itself, e.g. a URP parameter will not allow the remote user to switch to XTP, X.25, or Johnny’s private version of NETBLT. A UAPI implementation allows all such choices.

I/O Streams —The difference in philosophy with I/O streams [6], NICL [2], communication messengers [8], or general remote procedure call mechanisms is worthwhile to consider here, although mostly to illustrate what a UAPI is *not* meant to be. A UAPI interface is not an operating system concept but a

data communication concept. A UAPI has value only when it is connected to a data link, connecting peers that do not (wish to) know each others capabilities a priori. It is distinct from I/O streams in that it is meant to solve a problem between operating systems, not a problem within operating systems. A UAPI message is also not intended to be a process. It is meant to be a protocol definition, that is specified in a pre−arranged notation.

Network services —A network service in NICL, or a communication messenger in [8], is defined to be either an algorithm that a remote host can execute, or a process that the remote host can run on behalf of the sender. Implementing such a service requires changes in the operating system of the remote host, and it requires knowledge at the sender of the precise capabilities of a remote host (e.g., which services it implements). In contrast, the protocol interfaces studied here are operating system and machine independent. They are defined outside the host machine, and their operation is ideally confined to the interface boards of a machine. The language of a UAPI controls only the rules for data exchange, and little else. The UAPI standardizes the interface between machines that do not know, and perhaps do not even trust each other. Specifically, they do not, or need not, trust each others implementation of specific data exchange protocols. They certainly do not attempt to offer services to each other, beyond mere standardized accessibility.

The standardization of protocol interfaces is a problem that, justly or unjustly, has been ignored in the past. We have, of course, learned quite well to operate computer networks without the luxury of such standards. Let us, however, consider how it could be done differently. We begin by taking a closer look at the structure of a UAPI, and what it would take to implement one in software.

3. UAPI STRUCTURE

In the following we will consider one particular way to implement a UAPI. No particular significance, however, should be attached to the precise choice of instructions or data structures; they are only meant to give us a vehicle for discussing the main ideas in more detail.

Figure 3 shows a general outline of a UAPI implementation. It communicates with a host system on the one side, and with a data link on the other. In principle, of course, UAPI devices could also be stacked, allowing for layered operation, but no such generality will be needed for the current discussion.

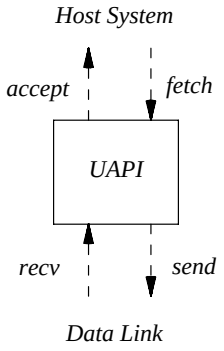

Figure 3 — Context

The UAPI contains a generic controller with an instruction set that includes specific commands related to the task of protocol control. The UAPI controller we have implemented has a generic set of data structures, depicted in Figure 4, that is used to enforce the user−defined protocol rules for manipulating incoming and outgoing data.

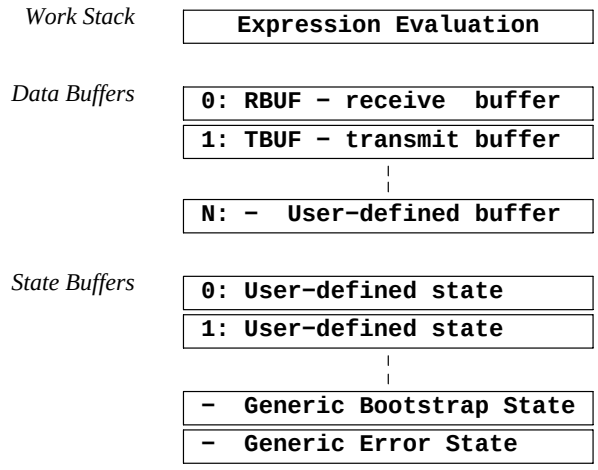

Figure 4 — Data Structures

There is a predefined (pointer to a) buffer for storing incoming UAPI messages, with the required size allocatable on the fly. Similarly, there is a predefined buffer for outgoing messages. Both buffer locations can be overriden by the user, via the UAPI controller. A set of scratch data buffers is included for that purpose.

The protocol rules are encoded into an extended state machine format. For that purpose, a range of 'state buffers' is available. Each state buffer can be loaded with behavior definitions that encode the desired behavior of the UAPI protocol machine at that state. Two such states are pre−defined: the bootstrap state, and a generic error state. Not discussed here are options that would be required for making the UAPI controller transparent to remote users that do not wish to make use of UAPI capabilities, or for the caching of UAPI protocols on the fly. The selection of the UAPI mode of behavior for the device could, for instance, easily be made dependent on the transmission of a magic sequence, protected by default integrity checksums to avoid unintentional triggers. The failure to detect such trigger sequences could put the device in a conventional, transparent mode of operation. We will also not attempt to provide a detailed definition of the interface between a host machine and a UAPI device. To explain the operation of the UAPI device it will suffice to assume no more than a minimal interface to the host (the upper interface) and to the data link (the lower interface), where arbitrary data messages can flow in either direction.

First, we will now have to consider what precise capabilities the UAPI controller should have to allow it to operate as intended. The simplest choice is to give the controller a full set of arithmetic, logical, and bitwise operators for expression evaluation. In our implementation we have allowed the usage of arbitrary expressions involving constants and variables of two basic types (bytes and words), as shown in Table 1.

Table 1 – Sample Instruction Set − General

0. Stack Management and Expression Evaluation	
PLUS,MINUS,UMIN,TIMES,DIVIDE,MODULO	arithmetic operations
AND,OR,GT,LT,GE,LE,EQ,NE,NOT	boolean operations
SHIFTL,SHIFTR,BIT_AND,BIT_OR,XOR,BIT_COMPL	bitwise operators
I_PUSH_BYTE,I_PUSH_WORD	push a constant onto the stack
I_PUSH_BYTE_VAR,I_PUSH_WORD_VAR	push a variable onto the stack, constant operands
PUSH_BYTE_VAR,PUSH_WORD_VAR	push a variable onto the stack

The stack we implemented is a standard integer size push−down stack for runtime expression evaluation. All stack−errors,such as underflow or overflow, and all arithmetic errors, such as divide by zero and integer overflow, cause a transition of the controller to the predefined error state.

Our controller has sixteen specific protocol control commands, grouped into four categories, as shown in Table 2. We want the behavior of the controller to be machine independent, both on the sender and the

receiver side. The definition of a ‘‘word’’ for the PUSH_WORD and PUSH_WORD_VAR operations in Table 1 is therefore under user−control with the help of instructions BYTEORDER and WORD_SZ. For obvious reasons, these two instructions should be the first commands issued by the originating side, before the downloading of a controller begins. The two instructions themselves are byteorder and wordsize independent. The defaults are MSB (most significant byte transmitted first) and two bytes per word, respectively.

Table 2 – **Instruction Set − Protocol Related**

MNEMONIC	PARAMETERS	PURPOSE
	1. FSM Control	
LOAD	2 (buffer nr,state nr)	assign a new state definition
NXT	1 (state nr)	perform a state transition
IF,ELSE	1 (value)	conditional execution of commands
	2. Upper Interface	
ACCEPT	1 (buffer nr)	pass a message to the host
OBTAIN	1 (buffer nr)	fetch a new message from the host
	3. Lower Interface	
RECV	1 (buffer nr)	receive a message from the data link into buffer
SEND	1 (buffer nr)	send a message to the data link from buffer
CKSUM	1 (buffer nr)	calculate a checksum on buffer contents
BYTEORDER	1 (constant)	define byte−order of data link
WORD_SZ	1 (constant)	number of bytes per word on data link
	4. Buffer Management	
ALLOC	2 (buffer,maxsize)	allocate bufferspace for buffer
SETSIZE	2 (buffer,size)	define the length field of a message in buffer
SETTIMO	2 (buffer,time)	define a timeout period for a message in buffer
CPY_BYTE	3 (buffer,index,value)	set a byte−value in a message
CPY_WORD	3 (buffer,start_index,value)	set a word (N bytes)

For the above basic instructions the parameters are variables that are popped from the stack, and that can therefore be calculated and modified at run−time. Most instructions also have a slightly faster variant (all with prefix I_) for use with constant operands, avoiding some of the stack operations. The data buffers are used to store all temporary information used in the protocol: message bodies, byte and word variables, timeout counts, checksums, etc. With the LOAD instruction, the contents of a data buffer can be transferred to a state buffer and subsequently used for protocol control. The following program, for instance, first sets the byteorder, overrides the default wordsize (two byte per word), then transfers data (presumably UAPI instructions) from data buffer RBUF to state buffer S, and transfers control to that newly defined state.

```
BYTEORDER,    1,          /* Most Significant Byte transmitted first */
WORD_SZ,      3,          /* 3 bytes per word */
I_LOAD,       S, RBUF,    /* assign RBUF to S */
I_NXT,        S,          /* goto state S */
```

This looks very much like programming a stack−machine in assembler. We would not want to describe any substantial protocol in this format by hand. For a higher−level format we can use any formal protocol specification language that can be translated into the UAPI assembler. Even a regular programming language would do. If the protocol is specified in a language such as PROMELA [4] there is an added advantage that the complete protocol can be validated exhaustively before it is converted into UAPI assembler. We have certainty in this case that both sides of the protocol are implemented in precise accordance with the validated model.

There needs to be an agreement between sender and receiver on the format used for defining protocols (i.e., on the instruction set of the UAPI and its encoding). The main difference with hard−coded protocols is that

the UAPI standardizes only the way in which protocols are defined, and not the protocols themselves.

THE CONTROLLER

The complete implementation of the UAPI system with the instruction set from Tables 1 and 2 is less than 700 lines in C. The controller itself is roughly 200 lines of source. The following fragment illustrates how it works.

```
transition(n)
{       BYTE b1, *cur_state = State[n].cont;
        static   int Stack[SMAX+1];
        register BYTE *prot;
        register int w0, w1, w2, i;
        register int *sp = &Stack[SMAX];

        prot = cur_state;
        while (prot) {
                switch (*prot++) {

                /***** FSM CONTROL *****/

                case NXT:
                        assert(sp < &Stack[SMAX]);
                        w0 = POP;
                        debug("next %d\n", w0, 0, 0, 0);
                        if (w0 < 0 || w0 > SMAX || !State[w0].cont)
                                return ERRORSTATE;
                        return w0;
                case I_NXT:
                        w0 = *prot++;
                        debug("next %d\n", w0, 0, 0, 0);
                        if (w0 < 0 || w0 > SMAX || !State[w0].cont)
                                return ERRORSTATE;
                        return w0;
                ...

                default:
                        debug("Error <%d>\n", *prot, 0, 0, 0);
                        return ERRORSTATE;
        }       }
}
```

The procedure `transition(n)` executes one state transition, starting with the behavior that was specified for `STATE[n]` (initially the predefined bootstrap state, see below). The execution of every command that is recognized starts with an assertion that performs a sanity test on, for instance, the state of the stack, and it includes a debug statement that prints some useful information for a system designer. The assertions and the debug statements can be disabled at compile time.

The NXT command pops one operand of the stack and if it defines a state in the correct range that has been initialized it returns it to the calling function as the next state. If it is invalid, a transition to the error state is made. Similarly, the I_NXT command reads a constant operand from the program string, and proceeds like NXT.

THE UAPI BOOTSTRAP

Upon connection setup, the UAPI controller contains only two predefined states, the *bootstrap* state and the *error state* mentioned earlier. Since these two states are unavoidably outside the scope of user−defined protocols, they are the only ones that enforce a predefined format on incoming messages, and that must require, for instance, the presence of a checksum field. Immediately after receiving the required response, these two states hand−off protocol control to user−defined portions.

The UAPI is started with a single call `run(BOOTSTRAP)`. Procedure `run()` is reproduced completely below.

```
run(s)
{       int n = s;
        while (n >= 0 && n <= SMAX && State[n].cont)
                n = transition(n);
        return n;
}
```

The definition of the code for the bootstrap and the error state illustrates most of the concepts of the working of the UAPI. A full definition is therefore included in Appendix A. For a general understanding, however, the flow−chart for the bootstrap behavior shown in Figure 5 suffices.

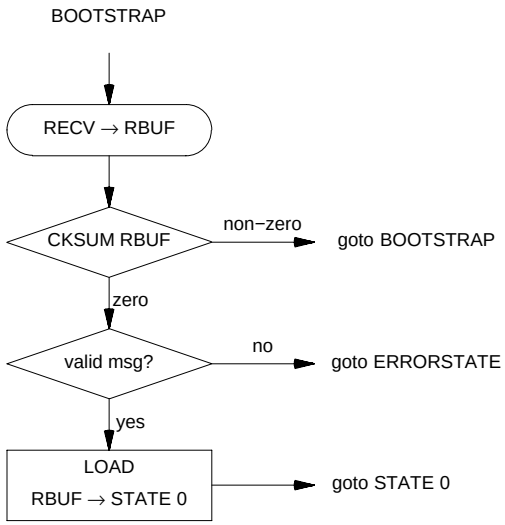

Figure 5 — Bootstrap State

The bootstrap routine waits for the initial setup message from the remote sender. A checksum is performed, to make sure the message is uncorrupted. If the checksum is non−zero, a transmission error has occurred, and the machine returns to the start of the bootstrap state. If the checksum is zero, a check is made if the message has the correct type. The first instruction is required to be the definition of the BYTE-ORDER for the lower interface (no word operations could be carried out otherwise). This byte−order definition specifies the order in which the bytes in a word are transmitted across the lower level interface: most or least significant byte first. It need not match the byte−order on either the receiving or the sending machine.

If the message is a valid bootstrap operation, the contents of receive buffer is assigned to the initial state of the newly loaded protocol, and control is transferred to that state. From that point on, the remote machine is in charge and can continue the loading and protocol execution at will.

If the first instruction is an invalid one, control is transferred to a generic error state. Like the bootstrap state, this error state is not part of any protocol proper. It is only meant to provide a standard mechanism for responding to errors in the operation of the UAPI, such as stack−overflow, memory allocation errors, arithmetic errors (e.g., divide by zero), etc. A flowchart for the error state is given in Figure 6. The full definition is included in Appendix A.

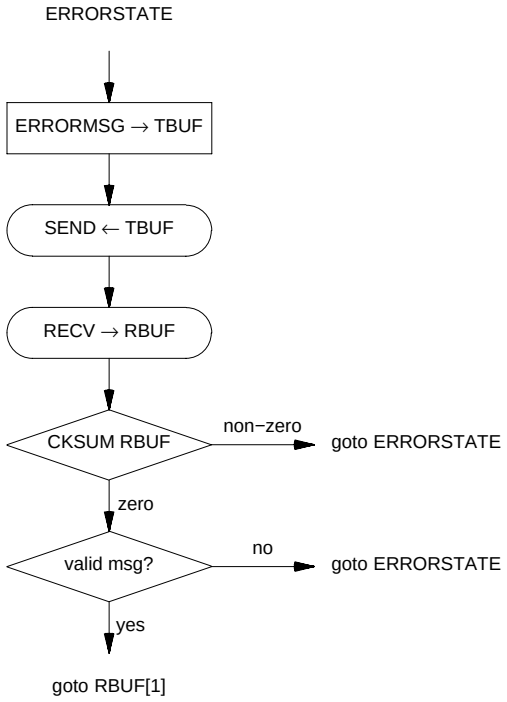

Figure 6 — Error State

The error state first notifies the remote UAPI of an error condition by sending a predefined message. It then awaits a response that it will receive into the default receive buffer RBUF. If the message was uncorrupted, and of the right type, control is transferred to the state that is specified in the message, using the NXT command. In all other cases, the error state is reentered from the top.

4. A SMALL APPLICATION

The feasibility of the UAPI scheme was tested with the implementation of the well−known alternating bit protocol [1]. As illustrated in Figure 1, the originating process initializes both the sender and the receiver side of the protocol. In the code below, some message buffers are initialized, the state descriptions for the receiver and the sender are loaded, and a checksum is calculated for the first (bootstrap) message to go out.

The generic UAPI receiver simply executes

```
run(BOOTSTRAP);
```

quite independent of the type of protocol that is to be used. The ABP sender executes

```
run(0);
```

after initializing the protocol−specific data structures as follows.

```
abp_init()
{       extern Buffer Buf[];
        extern Buffer State[];
        unsigned short w0;

        /* message buffers */
        Buf[M0].size = sizeof(Msg0); Buf[M0].cont = Msg0;
        Buf[M1].size = sizeof(Msg1); Buf[M1].cont = Msg1;

        /* Receiver states − loaded by Abp_snd_ini into remote UAPI */
        Buf[R_ini].size = sizeof(Abp_rcv_ini); Buf[R_ini].cont = Abp_rcv_ini;
        Buf[R_run].size = sizeof(Abp_rcv_run); Buf[R_run].cont = Abp_rcv_run;

        /* attach checksum, required on first transmission */
        w0 = cksum(Buf[R_ini].cont, Buf[R_ini].size−2);
        Buf[R_ini].cont[Buf[R_ini].size−1] = w0>>8;
        Buf[R_ini].cont[Buf[R_ini].size−2] = w0&255;

        /* Transmitter states */
        State[0].size = sizeof(Abp_snd_ini); State[0].cont = Abp_snd_ini;
        State[1].size = sizeof(Abp_snd_run); State[1].cont = Abp_snd_run;
}
```

The header file that contains the definitions for the sender and the receiver is 92 lines long. It is listed for reference in Appendix B.

5. PERFORMANCE

A performance test was done on a VAX−8550 with cross−connected user−level UNIX processes (connected with in−core pipes). The send and receive operations were implemented with read and write systems calls. The expense of these read and write calls is a peculiarity of the specific implementations used in the test, and not the direct objective of our comparison. More typically these interface routines will be interrupt−driven device driver routines. The send and receive code was therefore kept the same in both the hard−coded and the UAPI implementation of the protocol, and the expense of their use was subtracted from the run−times reported below, giving the expense of just the controller portions of the implementations (the only part that was changed). The message length thereby became irrelevant (note that the read/write times go up with message length and tend to make the controller overhead disappear).

Compared are the receiver portions of the UAPI implementation of the alternating bit protocol shown in Appendix B and the hard−coded version listed in Appendix C. A total of 32765 messages was transmitted and acknowledged per test run. All times are averaged over five testruns, and give user−time plus system−time in seconds.

Table 3 – Runtime Comparison (Time in Seconds)

Version	Avg. CPU−time	Avg. R/W (1Kb Msgs)	Difference
UAPI	31.9	18.3	13.6
HARD	28.7	18.3	10.4

The software controlled UAPI incurs a time penalty of about 30% over a hard−coded version.

Table 4 shows another measurement of the ten most expensive operations of a run of the UAPI implementation (receiver side) with 1 Kbyte messages and four byte acknowledgements. Some 66% of the time goes to the cost of read and write system calls (irrelevant to this test), and over 18% of the time is spent in the controller. This overhead can be reduced by a third by switching to a hard−coded protocol.

Table 4 – Performance (1 Kbyte Msgs)

%	Time	Calls	Name
36.6	10.533	0	_write
29.4	8.467	0	_read
18.5	5.317	32766	_transition
3.8	1.083	0	_signal
2.1	0.617	0	_alarm
1.9	0.550	32765	_getheader
1.7	0.500	98298	_cpyval
1.6	0.450	32765	_recv
1.0	0.283	32767	_send
0.6	0.183	1	_run

The software implementation of the UAPI controller clearly increases the overhead. Not counting the expense of read/write calls, the expense came out at 30% for the alternating bit example. Taking the cost of read/write access to upper and lower layers into account this means that a *software* implemented UAPI controller could reduce the maximum transmission speed that by roughly 10% (i.e., 30% of the 18% spent in the controller). For more control−intensive protocols the difference will be greater and can become prohibitive.

CHECKSUMS

Certain type of functions of the UAPI can still be exploited without any degradation in performance over hard−coded protocols. The calculation of a CRC checksum can, of course, be done with the instruction set of the UAPI. It can, however, also be done much faster with a simple standard table lookup algorithm, such as the following (by Don Mitchell of AT&T Bell Laboratories):

```
cksum(s, n)
        register unsigned char *s;
        register n;
{
        register int crc;

        crc = 0;
        if (n)
                do
                        crc = crc_table[(crc ^ *s++) & 0xff] ^ (crc >> 8);
                while (--n > 0);
        return crc;
}
```

The contents of the `crc_table` can be stored in a buffer and changed, by downloading, without affecting the speed of the CRC calculations in any way. Lookup tables for any CRC polynomial can be generated with a simple generator program, as described in [4]. The above generic CRC calculation program is accessed with the CKSUM instruction from Table 2.

6. DISCUSSION

The software implemented version of a UAPI scheme we have described can easily replace relatively low−level machine to machine data copying services, such as the UNIX utility **uucp**. At present, the **uucp** program provides a single protocol for coupling arbitrary machines, disregarding the peculiarities of the specific links that may connect them. In practice, the specifics can range from two high−speed hosts standing side by side in a machine room connected by a fiber optic link, to machines at opposite sides of the globe, connected through switched telephone lines. At best, such a protocol would provide a service that matches the average of such extremes. In no specific circumstance, however, can it provide an optimal service. To do so, the sender must know the precise peculiarities of the specific link that is used for each connection, be it a virtually zero−delay, error−free channel, or a long delay error−prone channel. It is folly to standardize

on a single hard−coded protocol to solve both problems [cf. 4].

The attractiveness of the **uucp** example is that it can merely provides a data copying service, leaving it to host resident application programs to decide what, if anything, to do with incoming data. The method could, of course, be extended to also allow for remote parties to sparc off user level applications on remote machines, but such an extension may create undesirable security risks.

Leaving an *air gap*, in the form of data buffers, between the capabilities granted to a remote party and the services offered by a local host, restricts the flexibility of the UAPI idea, but it does avoid the potential security risks. An easy alternative would be to provide an independent authentication of requests that arrive via UAPI interfaces, before any local services are granted (such as taking over and clearing the file system on the remote host).

OPTIMIZATION

The optimal method for implementing a UAPI controller is probably to place it in special−purpose hardware outside the host machine, just like a UART controller. Note that such a chip or board would not only be protocol independent, but also on par with the performance of pre−compiled hard−coded protocols. As an added benefit, the reliance on untrusted remote protocol implementations is removed.

A software implementation like the one discussed in this paper can be useful only

- When performance is secondary to flexibility, (e.g. by the time a hypothetical new probe to Jupiter would reach its destination, we may have better checksum polynomials that we will want to use)
- When the protocol control portion is relatively small and the I/O overhead relatively small
- As a temporary measure, to allow communication between hosts that, for whatever reason, do not (want to) know each others protocols

In other cases, specifically when the performance degradation of a straightforward software implementation proves to be too large, several alternatives for optimization can be considered. In increasing order of effectiveness, these alternatives are:

- Optimizing the controller software, e.g. replacing function calls with inline code.
- The inclusion of a run−time optimizer on UAPI code, that could be invoked by a local UAPI controller, during the bootstrapping phase. In its simplest form this would be a rewriting program that tries to find expensive operations and rewrites them.
- On−the−fly compilation of UAPI code for the specific host on which the software implemented controller runs. A small builtin−compiler can produce optimized machine instructions that are written directly into memory, and executed upon demand (e.g. as in [5]).
- Caching the frequently used protocol descriptions at the receiving UAPI, to avoid at least the downloading of complete protocol descriptions in most cases. The UAPI then become a learning system that adapts to its environment. A check on the presence of a cached version could be performed by sending a unique protocol identifier (i.e. a sufficiently precise checksum over the protocol description) that is matched against protocol descriptions in the UAPI cache. Only on a cache miss would a full protocol description be loaded.
- A hardware implementation of the complete UAPI controller.

It is tempting to compare the adoption of a UAPI to the adoption of high−level programming languages. A good programmer writing in assembler may be able to speed up code by 10−20% over the code that a good optimizing compiler would produce. But, is it worth it? Sometimes, it is, but in most cases the convenience of the higher level language outweighs the loss in performance. The same may prove to be true for standardized protocol interfaces.

7. CONCLUSION

We have proposed the construction of a new type of controller for data links. In principle, the controller can be used wherever it is installed. If a UAPI is present only at the remote end of a data line, the local host can down−load trusted protocols to it and exploit them in data transfers. If it is present only on the local end of a line, the local host can down−load trusted protocols to it that match whatever protocol was hard−

coded at the remote end. If, in the best scenario, a UAPI controller is installed on both ends of the data link, the new scheme can be fully exploited by loading arbitrary protocols into remote and local schemes.

The UAPI can be configured to accept and execute any type of protocol. It has a single bootstrap mode that implements a minimal downloading protocol for protocol descriptions to initialize a session. Popular protocol descriptions can be cached inside the UAPI's to avoid repeated downloads, but the precise set of cached descriptions can change dynamically, following most frequent usage. As in normal disk−block caching schemes, this style of protocol caching can improve performance, but otherwise remains invisible to users. Once a protocol initialization is completed, the operation of a UAPI is indistinguishable from a hard−coded implementation of whichever protocol was selected. It is even possible to change part or all of the protocol logic dynamically in the middle of data transfer sessions, to exploit changing requirements or constraints. As an extreme example, a data transfer protocol could dynamically switch from one checksum polynomial to another, for instance to improve protection against burst errors, without requiring that the receiver has prior knowledge of the new polynomial.

Whether the adoption of a standardized protocol interface is feasible or desirable, is fortunately well beyond the scope of this paper; it rests comfortably in a non−computable domain.

ACKNOWLEDGEMENT

I am grateful to Dennis Ritchie, Gene Nelson and Geoffrey Brown for inspiring discussions and reflections on the ideas explored here. The idea to extend the UAPI with a cache for protocol descriptions is Gene Nelson's.

8. REFERENCES

[1] Bartlett, K.A., Scantlebury, R.A., and Wilkinson, P.T. ''A note on reliable full−duplex transmission over half−duplex lines,'' *Comm. of the ACM*, 1969, Vol. 12, No. 5, 260−265.

[2] Falcone, J.R. and Emer, J.S. ''A programmable interface language for heterogeneous distributed systems,'' DEC−TR−371, August 1986. also in ACM Trans. on Computer Systems, Vol. 5, No. 4, 1987, pp. 330−351.

[3] Fraser, A.G., and Marshall, W.T., ''Data Transport in a Byte Stream Network,'' *IEEE J. on Selected Areas in Comm.*, Sept. 1989, Vol SAC−7, No. 7, pp 1020−1033.

[4] Holzmann, G.J. *''Design and Validation of Computer Protocols,''* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[5] Holzmann, G.J. ''Pico — a Picture Editor,'' *AT&T Technical Journal*, 1987, Vol 66, No. 2, pp. 2−13.

[6] Ritchie, D.M. ''A stream input−output system,'' *AT&T Technical Journal*, 1984, Vol 63, No. 8.

[7] Strayer, W.T., Dempsey, B.J., and Weaver, A.C. ''XTP — The Xpress Transfer Protocol,'' Addison−Wesley, Reading, Mass., 1992.

[8] Tschudin, C.F. ''Communication by messengers,'' Technical Report, 1992, Centre Universitaire d'Informatique, Groupe Telecom, University of Geneve, Switzerland.

APPENDIX A — THE TWO PREDEFINED STATES

THE BOOTSTRAP STATE

This is the full listing of the UAPI bootstrap state, shown in flowchart form in Figure 5. The commented numbers on left side of the listing below give the relative (byte) position of the first instruction on each line. The program is 27 bytes long.

```
BYTE    Bootstrap[] = {
/*0*/   I_RECV, RBUF,           /* receive message into RBUF        */
/*2*/   I_CKSUM, RBUF,          /* always checksum initial msg      */
/*4*/   IF, 10,                 /* if nonzero goto instruction #10  */
/*6*/   I_PUSH_WORD, BOOTSTRAP>>8, BOOTSTRAP&255,
/*9*/   NXT,            /* stay in bootstrap state                  */
/*10*/  I_PUSH_BYTE_VAR, RBUF, 0, /* get variable Buf[RBUF].cont[0] */
/*13*/  I_PUSH_BYTE, BYTEORDER,
/*15*/  NE,                     /* Buf[RBUF].cont[0] != BYTEORDER   */
/*16*/  IF, 22,                 /* if false goto instruction #22    */
/*18*/  I_PUSH_WORD, ERRORSTATE>>8, ERRORSTATE&255,
/*21*/  NXT,
/*22*/  I_LOAD, 0,      RBUF,   /* it checks out, define State[0]   */
/*25*/  I_NXT,  0               /* and execute it                   */
};
```

The bootstrap routine waits for the initial setup message from the remote sender (0). The message is received in a buffer, called RBUF here. A checksum is performed on the contents of that buffer (2), to make sure it's contents are uncorrupted. If the checksum is non−zero, a transmission error has occurred (4), and the machine returns to the start of the bootstrap state (6), trying its luck on the next message that arrives. If the checksum is zero, the first byte of the message is inspected (10). The first instruction is required to be the definition of the BYTEORDER for the lower interface (no word operations could be carried out otherwise). Note carefully that this need not match the byte−order on either the receiving or the sending machine. It simply defines the order in which the bytes in a word are transmitted across the lower level interface: most or least significant byte first.

If the first instruction is valid (16), the contents of buffer RBUF is defined as the initial state for the newly loaded protocol (22), and the machine transfers control to that state. From that point on, the remote machine is in charge and can continue the loading and protocol execution at will.

If the first instruction is an invalid one, control is transfered to a generic error−state, that is defined in the next section.

THE ERROR−STATE

In the same format, the definition of the error−state looks as follows. It is shown in flowchart form in Figure 6. The program occupies 33 bytes of memory.

```
BYTE    Errorstate[] = {
/*0*/   I_ALLOC,        TBUF, 1,
/*3*/   I_SETSIZE,      TBUF, 1,
/*6*/   I_CPY_BYTE,     TBUF, 0, ERRORMSG,
/*10*/  I_SEND, TBUF,
/*12*/  I_RECV, RBUF,
/*14*/  I_CKSUM,        RBUF,
/*16*/  IF, 22,                 /* if nonzero move to instruction #22 */
/*18*/  I_PUSH_WORD, ERRORSTATE>>8, ERRORSTATE&255,
/*21*/  NXT,
/*22*/  I_PUSH_BYTE_VAR,RBUF, 0,
/*25*/  I_PUSH_BYTE,    NXT,
/*27*/  EQ,                     /* Buf[RBUF].cont[0] == NXT          */
/*28*/  IF, 18,                 /* if false move to instruction #18  */
/*30*/  I_PUSH_WORD_VAR, RBUF, 1,
/*33*/  NXT
};
```

The error−state begins by allocating space for a outgoing error message. It needs a buffer of just one byte

long. Instructions (0−2) allocate that space, the length of the message is defined in (3−5), the message type is assigned in (6−9), and the message is sent in (10−11).

The error state then awaits a response that it will receive into the default receive buffer RBUF (12). If the message was corrupted (16), the machine transfers back to the beginning of the error state. If it is uncorrupted, the message type is checked to see if it holds a valid response to the error condition. The only valid response is a NXT command (27−28) followed by two bytes for its operand (a word), that may transfer control to any of the previously defined states (30), including the BOOTSTRAP state.

APPENDIX B — UAPI−CODED ALTERNATING BIT PROTOCOL

This appendix gives the full listing of the UAPI data structures for the alternating bit protocol. Typically, a description like this would be mechanically generated from a higher level protocol specification language. The tables below, however, were produced manually in about one tedious hour of human time.

```
/* Receiver Buffers */
#define RBUF    0       /* receive buffer */
#define TBUF    1       /* transmit buffer */
#define VAR_E   2       /* variable 'e' − receiver side */

/* Transmitter Buffers */
#define M0      0       /* message M0 */
#define M1      1       /* message M1 */
#define R_run   2       /* Abp_rcv_run    */
#define R_ini   3       /* Abp_rcv_ini    */
#define R_ack   4       /* receive buffer − for acks    */
#define VAR_S   5       /* variable 's' − sender side   */
#define VAR_CNT 6       /* variable 'cnt' − sender side */

#define B_O     1       /* byteorder */
#define NR_MSGS 32765   /* number of test messages sent */

BYTE Abp_rcv_ini[] = {          /* initialization Buf[R_ini]; recvd in State[0] */
/*0*/   BYTEORDER,      B_O,
/*2*/   I_ALLOC,        TBUF, 2,
/*5*/   I_SETSIZE,      TBUF, 2,
/*8*/   I_ALLOC,        VAR_E, 1,
/*11*/  I_RECV,         RBUF,
/*13*/  I_LOAD,         1, RBUF,/* input becomes State[1] */
/*16*/  I_NXT,          1,      /* execute it */
/*18*/  0,              0,      /* room for the checksum; required on 1st msg */
};

BYTE Abp_rcv_run[] = {  /* abp receiver Buf[R_run]; recvd in State[1] */
/*0*/   I_RECV,         RBUF,
/*2*/   I_CPY_BYTE,     TBUF,   0,      'A',
/*6*/   I_PUSH_BYTE_VAR,        RBUF,   1,
/*9*/   H_CPY_BYTE,     TBUF, 1,
/*12*/  I_SEND,         TBUF,
/*14*/  I_PUSH_BYTE_VAR,        RBUF,   1,
/*17*/  I_PUSH_BYTE_VAR,        VAR_E,  0,
/*20*/  EQ,
/*21*/  IF,     34,             /* e == rbuf[1] */
/*23*/          I_PUSH_BYTE,    1,
/*25*/          I_PUSH_BYTE_VAR,VAR_E,  0,
/*28*/          MINUS,
/*29*/          H_CPY_BYTE,     VAR_E,  0,
/*32*/          I_ACCEPT,       RBUF,
/*34*/  I_NXT,  1               /* stay in same state */
};

BYTE Msg0[] = { /* message Buf[M0], received in Buf[RBUF] */
        'M', 0
};
BYTE Msg1[] = { /* message Buf[M1], received in Buf[RBUF] */
        'M', 1
};
```

```
/* sender behavior */
BYTE Abp_snd_ini[] = {                  /* becomes State[0] */
/*0*/   I_ALLOC,        VAR_S, 1,       /* the byte variable 's' */
/*3*/   I_CPY_BYTE,     VAR_S,0,0,      /* s = 0 */
/*7*/   I_ALLOC,        VAR_CNT,2,      /* the word variable 'cnt' */
/*10*/  I_CPY_WORD,     VAR_CNT, 0, 0,  /* cnt = 0 */
/*14*/  I_SEND,         R_ini,          /* send Buf[R_ini] == Abp_rcv_ini */
/*16*/  I_SEND,         R_run,          /* send Buf[R_run] == Abp_rcv_run */
/*18*/  I_NXT,          1               /* begin actual behavior */
};

BYTE Abp_snd_run[] = {                  /* becomes State[1] */
/*0*/   I_PUSH_BYTE_VAR, VAR_S, 0,
/*3*/   SEND,                           /* send from buf[s] */
/*4*/   I_RECV,         R_ack,          /* recv into buf[r_ack] */
/*6*/   I_PUSH_BYTE_VAR,R_ack, 0,       /* look at Buf[R_ack].cont[0] */
/*9*/   I_PUSH_BYTE,    'A',
/*11*/  EQ,
/*12*/  IF,             32,             /* #12 - #13 */
/*14*/          I_PUSH_BYTE_VAR, R_ack, 1,      /* Buf[R_ack].cont[1] */
/*17*/          I_PUSH_BYTE_VAR, VAR_S, 0,      /* s */
/*20*/          EQ,
/*21*/          IF,     32,     /* #21 - #22 */
/*23*/                  I_PUSH_BYTE,    1,
/*25*/                  I_PUSH_BYTE_VAR, VAR_S, 0,
/*28*/                  MINUS,
/*29*/                  H_CPY_BYTE,     VAR_S, 0, /* s = 1 - s */
/*32*/  I_PUSH_BYTE,    1,      /* #32 - #33 */
/*34*/  I_PUSH_WORD_VAR,VAR_CNT, 0,
/*37*/  PLUS,
/*38*/  H_CPY_WORD,     VAR_CNT, 0,             /* cnt = 1 + cnt */
/*41*/  I_PUSH_WORD_VAR,VAR_CNT, 0,
/*44*/  I_PUSH_WORD,    NR_MSGS>>8,     NR_MSGS&255,
/*47*/  GE,                     /* cnt >= NR_MSGS */
/*48*/  IF,             54,     /* #48 - #49 */
/*50*/                  I_PUSH_WORD,    255, 255,       /* -1 = exit */
/*53*/                  NXT,
/*54*/  I_NXT,          1       /* #54 - #55 stay in this state */
};
```

APPENDIX C — HARD−CODED ALTERNATING BIT PROTOCOL

```
/* hardcoded version of ABP receiver */

#include <stdio.h>
#include "serp.h"

extern Buffer   Buf[];
extern Buffer State[];

BYTE MSB = 1;

main()
{       BYTE twobuf[2];
        int E = 0;
        Buf[1].cont = twobuf; Buf[1].size = 2; Buf[1].cont[0] = 'A';

        while (recv(0))
        {       Buf[1].cont[1] = Buf[0].cont[1];
                send(1);
                if (Buf[0].cont[1] == E)
                {       E = 1 - E;
                        accept(1);
        }       }
}

/* hardcoded version of ABP sender */

#include <stdio.h>
#include "serp.h"

#define NR_MSGS 32765   /* number of test messages sent */

extern Buffer   Buf[];
extern Buffer State[];

BYTE Msg0[] = { 'M', 0 };
BYTE Msg1[] = { 'M', 1 };
BYTE MSB = 1;

main()
{       int S = 0;
        int cnt = 0;

        Buf[0].cont = Msg0; Buf[0].size = /* sizeof(Msg0) */ 1022;
        Buf[1].cont = Msg1; Buf[1].size = /* sizeof(Msg1) */ 1022;

        do {    send(S);
                recv(2);
                if (Buf[2].cont[0] == 'A' && Buf[2].cont[1] == S)
                        S = 1 - S;
        } while (cnt++ < NR_MSGS);
}
```