

Implementing Abstract Protocols in C¹

Tommy Marcus McGuire, M.A.
The University of Texas at Austin, 1994

Supervisor: Mohamed G. Gouda

The goal of this work is the creation of a system to aid the implementation of a network communication protocol specified in the Abstract Protocol notation. The requirements of this system are primarily faithfulness to the formal notation and at least a semblance of efficiency. To this end, a suite of functions has been written to allow the creation of the processes involved in the protocol, originally specified in the Abstract Protocol notation, in the programming language C.

The basic idea of the suite is the registration of functions whose execution affects the protocol state and which are enabled or disabled by the protocol state. The core of the suite is the function `engine()`, which invokes the functions in response to state changes.

Several example protocols and their implementations will be presented, including an evolution of a simple request/reply protocol into a more complex form. Also, a reference has been provided for the functions in the suite.

¹This thesis was presented to the faculty of the Graduate School of The University of Texas at Austin in December, 1994. This edition has also been published as Technical Report TR96-31 from the Department of Computer Sciences at The University of Texas at Austin.

Contents

Abstract	i
Chapter 1 Introduction	1
1.1 Definitions	2
1.2 The <code>nuweb</code> system of structured documentation	3
Chapter 2 Language Issues	7
2.1 Abstract Protocols	7
2.1.1 AP syntax	7
2.2 The C language	9
2.2.1 Function pointers	10
2.2.2 “Subclassing” variables	10
2.2.3 C implementation issues	11
Chapter 3 Abstract Protocols in C	13
3.1 Implementing processes	13
3.1.1 Implementing actions	14
3.2 Vending machine protocol	15
3.2.1 Process identifiers and message types	16
3.2.2 The vendor	16
3.2.3 The customer	20
3.2.4 Elaborations	23
3.3 The request/reply protocol	24
3.3.1 Process identifiers and message types	25
3.3.2 Process <i>p</i>	26
Chapter 4 Parameters, Timeouts, and Process Arrays	31
4.1 Request/reply protocol revisited	32
4.1.1 Process identifiers, message types, and new constants	33
4.1.2 Process <i>p</i>	34

4.2	Reliable request/reply	39
4.2.1	Constants and message types	40
4.2.2	Process p	40
4.3	Request/reply using multiple processes	48
4.3.1	Constants and message types	49
4.3.2	Process p	50
Chapter 5 Reference Guide and Conclusion		59
5.1	The APC engine	59
5.1.1	Initializing and executing the engine	59
5.1.2	Receive and local actions	61
5.1.3	Timeout actions	63
5.1.4	Sending messages	65
5.1.5	Types and variables	67
5.1.6	The APC engine header file	69
5.2	Compiling and executing APC protocols	69
5.2.1	Compiling the programs	70
5.2.2	Executing the protocol	70
5.2.3	APC driver internals	72
5.3	Conclusion	73
5.3.1	Design and implementation	73
5.3.2	Future work	74

Chapter 1

Introduction

The Abstract Protocol (AP) notation solves many problems involved in the formal specification of networking protocols. Its syntax is very simple. Combined with protocol properties stated in its notation, it offers a set of proof obligations that make ensuring that a protocol satisfies a certain property relatively easy. For more information, see [Brown 91], [Burns 93], [Gouda 91], and [Gouda 93].

However, the gap between the specification and the implementation remains. For the most part, this is due to the relatively low-level nature of the communication primitives supplied by many operating systems and programming languages. The most advanced of these languages, such as Ada [Barnes 82] and Distributed C [Pleier 93], provide primitives for sending and receiving messages as well as structures such as **select** for building a networking protocol from those primitives. However, they are based on the idea of communicating sequential processes rather than on reactive systems which respond to state changes with action. It is not obvious that translation between AP and the structures such as **select** would be easy. More widely available systems, such as the Unix socket interface [Stevens 90], do not even provide the framework available in Ada; the focus in these systems is only on the sending and receiving primitives.

The large gap between an AP specification and its implementation in C using TCP/IP sockets lies primarily in bookkeeping, syntax, and structure. Initializing a socket in order to use it to send and receive messages is a multi-step procedure requiring considerable information that is not available (and should not be available) from the specification. Additionally, due to the lack of structure from the language and the socket interface, the implementation is required to provide such structure as is required. This structure actually has nothing useful to do with the protocol itself, but its presence in the implementation can cause many bugs and much confusion. It is very easy, when implementing an AP specification, to be led down a bad path

which results in complex, difficult, specialized, and almost unmaintainable code.

This system provides a framework which makes the implementation of an AP protocol straightforward, if not easy. With the suite of functions provided by the APC system, the implementation should closely resemble the specification in format. Much of the bookkeeping required to handle TCP/IP sockets is hidden from the protocol implementation, as well.

1.1 Definitions

One of the major problems with discussing networking protocols and their implementation is that the terms seem to have multiple meanings. This work uses several specific definitions in order to avoid confusion.

- A *protocol* is collection of two or more processes which are involved in a scheme for communicating with each other. This collection can either be of abstract specifications of processes or running programs. This contrasts with the definition of “protocol” which involves only the list of messages passed between processes and their hypothetical transactions—the definition of “protocol” which is used here implies that information, but does not specifically focus on it as it is not overwhelmingly useful when looking at the behavior of a group of processes.
- A *process* is a list of actions along with supporting definitions. In general, the single term “process” is used to mean an abstract specification, particularly using the Abstract Protocol notation.
- A *program* corresponds with an abstract process specification; it is a concrete specification written either in a programming language or in machine instructions (presumably after having been compiled from a programming language).
- A *running process* is a program which is actually executing; more precisely, it is the thread of machine states which is following a program’s instructions. This term corresponds to the normal operating system definition of “process.”

The Abstract Protocol notation is a system for defining the abstract specification of a protocol. The system described in this paper is a toolkit for implementing a protocol as one or more programs, starting from the abstract specification of the protocol in AP.

1.2 The nuweb system of structured documentation

Large portions of this work are, in fact, the source code to various programs and functions. This source code has been split up and the pieces rearranged and formatted in an effort to make them easier to understand, both for the person maintaining and updating the source code as well as the person reading the material without any intention of working with the code.

In order to support the formatting effort, a tool called **nuweb** was used to preprocess the \LaTeX input file before it was passed to \LaTeX . **nuweb** reads the input file (which has the extension “.w”) and parses it into two different sets of output. The first set is a \LaTeX file with the documentation (this text) passed through unchanged. In this file, the embedded source code is changed to a typewriter font and notations are added which allow the relations between the source code sections to be followed.

The second set of output is one or more source code files, suitable for passing to a compiler if necessary. In the code files, the source has been patched together in the order specified by the relations between the sections.

Source code sections, called “scraps” or “macros” in **nuweb** terminology, look something like:

\langle Clear the arrays 3 $\rangle \equiv$

```
for (k = 1; k <= n; k++)
{
  a[k] = 0;
  str[k] = "";
}
```

◇

Macro referenced in scrap 4a.

The code section effectively defines a macro, in this case called “Clear the arrays,” with the replacement text of “**for** (...) { ... }”. When discussing the section in terms of the name and replacement text, the term “macro” is generally used; when referring to the whole element as part of a document, the term “scrap” is used.

When the name of a macro is seen inside another macro, the replacement text is substituted into the replacement text of the second macro. If the scrap,

`<Initialize the data structures 4a> ≡`

```
sum = 0;
<Clear the arrays 3>
◇
```

Macro referenced in scrap 4b.

is seen while `nuweb` is processing a file, it will set the replacement text for “Initialize the data structures” to something like

```
sum = 0;
for (k = 1; k <= n; k++)
{
a[k] = 0;
str[k] = "";
}
```

and this whole text will be substituted for the use of “Initialize the data structures.”

The numbers which follow the macro name indicate the page number on which the scrap referred to can be found; the optional letter appended to the number indicates the scrap on that page.

If several scraps have the same name, their replacement texts are concatenated before they are substituted into the scrap using the name.

Finally some scraps have names in a typewriter font surrounded by quotation marks rather than angle brackets. The names of these scraps are used as the file names of the source code files which can be passed to a compiler or another tool. For example,

`"example.c" 4b ≡`

```
/*
 * Preliminary material...
 */
<Initialize the data structures 4a>
/*
 * Following material...
 */
◇
```

produces the file `example.c` which has the useful, compilable source code in it.

Normally within this document several related scraps (perhaps defining variables along with the code that uses them) have been grouped together, along with

some text which discusses the scraps. These groups are marked off from the surrounding text by thin horizontal lines.

For more information on structured documentation, see [Knuth 92], [Sewell 89], [Knuth 86], and [Knuth 93].

Chapter 2

Language Issues

2.1 Abstract Protocols

This section will present a brief look at those aspects of AP which are used by the example protocols described later—most of these aspects are either unique to AP or raise significant implementation issues.

Unfortunately, it is not within the scope of this work to present the complete AP notation. It would seem that this is a requirement for writing a protocol in AP, which would be a prerequisite for implementing it from the AP specification. On the other hand, many portions of AP, particularly those which are common to many abstract programming notations, can be straightforwardly translated into the programming language of choice. Some of these portions are **do** loops, **if** statements, assignments—indeed, almost any local statements—arrays, constants, and other definitions. For further information on these topics, see many of the references in the bibliography. The purpose of this paper is only to examine those issues raised by the effort to implement a protocol.

2.1.1 AP syntax

This brief look at AP begins with the top level syntax, that of the process.

```
process ⟨process name⟩  
inp ⟨process constant name⟩ : ⟨type definition⟩;  
    ...;  
    ⟨process constant name⟩ : ⟨type definition⟩  
var ⟨variable name⟩ : ⟨type definition⟩;  
    ...;  
    ⟨variable name⟩ : ⟨type definition⟩
```

```

par ⟨parameter name⟩ : ⟨type definition⟩;
    ...;
    ⟨parameter name⟩ : ⟨type definition⟩
begin
    ⟨action guard⟩ → ⟨statement list⟩
  || ...
  || ⟨action guard⟩ → ⟨statement list⟩
end

```

The ⟨process name⟩ can be either a single identifier or the specification of an array of processes:

```

process  $p$  [ $i$  : ⟨lower bound⟩ ... ⟨upper bound⟩]

```

In the process array, the first identifier, p , is the array name and the second, i , is the index of the array that refers to the current process.

The **inp** section defines constants that are effectively local to the current process specification. Constants which are global to the protocol can be defined using a **glob** section, but this is not needed in the examples. **var** defines variables which are local to the process. The **par** section provides *parameters*, which are used in statements such as

```

ready $p$ [ $j$ ] → ⟨do something⟩

```

If j were a constant value, this guard would allow only one element of *ready* p ever to control the execution of the action. If j were a variable, the firing of the action would depend on the value of the j^{th} element, using the current value of j . However, if j is a parameter ranging across a set of values, $0 \dots N$, this action is virtually replaced by a set of actions

```

ready $p$ [0] → ⟨do something⟩
|| ...
|| ready $p$ [ $N$ ] → ⟨do something⟩

```

with j replaced by the corresponding member of the range. The parameter allows an enabled action to be nondeterministically picked based on the entire *ready* p array.

The guard of the action can have one of three forms:

- ⟨local guard⟩—a local guard is a predicate that involves only local variables.

- **rcv** \langle message \rangle **from** \langle process name \rangle —a receive statement which is true when a message of the correct form is at the head of the channel from the process named in the receive statement. If the action is executed, the receive statement removes the message from the channel.
- **timeout** \langle global guard \rangle —a global guard is a predicate that can involve all of the variables of all of the processes as well as the state of all of the channels. Timeout actions are difficult to implement and should be avoided where possible.

The lists of statements consist of either statements dealing with local variables or send statements of the form **send** \langle message \rangle **to** \langle process name \rangle . A message consists of an identifier plus zero or more optional fields. For example, $msg(val, seq)$, is a message msg with two fields, val and seq .

Between each pair of processes (p, q) there exists a shared variable, a channel, $C.p.q$, into which p can send messages and from which q can receive them. At any one time, $C.p.q$ contains the sequence of messages which have been sent by p but not received by q . The number of messages in the channel can be represented by $\#C.p.q$, and the number of any particular type of message, msg , can be represented as $msg\#C.p.q$.

At each time step, one action is picked from the set of all of the actions whose guard predicates are **true**. (A **rcv** predicate is **true** if the message at the head of the channel that it is receiving from matches the message that the predicate specifies.) This action is executed. (At which time the **rcv** predicate removes the message from the channel before the statements of the action are begun.) Fortunately, due to the semantics associated with AP, picking one action at a time from the whole protocol is equivalent to picking one action each from the processes—no synchronization is required between processes.

2.2 The C language

This work assumes that the reader is relatively familiar with programming in C. However, one technique is used that is normally regarded as “advanced” C and may not be familiar to all readers. Also, another “trick” is used which is not necessarily good C and which may not be portable. Finally, a couple of issues are raised regarding the relationship between C and formal notations in general and between C and network programming.

2.2.1 Function pointers

The “advanced” technique is the manipulation of function pointers. A C function `foo` has the definition syntax of

```
int foo(int i)
{
    /* C code */
}
```

`foo` takes an `int` argument and returns an `int`. `foo` is called by

```
val = foo( 4 );
```

If the program uses the bare identifier `foo` without the trailing parentheses and argument, its value is a pointer to the implementation of the function. Therefore, it is possible to write

```
int (*a)(int);

a = foo;
```

in which case the address of the start of `foo` is assigned to `a`. The definition of `a` shows it to be a pointer to a function taking an `int` and returning an `int`.

`a` can be used to call `foo` (or any other function whose address is assigned to it) by writing

```
val = (*a)( 4 );
```

in which case `val` is assigned the same value as the statement above.

This technique is often used by graphical user interface systems to give an application the ability to set up handlers for input events from the user such as key presses and mouse actions.

2.2.2 “Subclassing” variables

The “trick” mentioned above is the use of casts of pointers to point to similar looking structures. For example, say the structure

```
struct bar
{
    int size;
};
```

is used by a program. Also, assume that the program uses a structure

```
struct mybar
{
    int    size;
    char *name;
}
```

to hold some extra information about a `struct bar`-like entity. It is possible to cast a pointer to a `struct mybar` to a pointer to a `struct bar`, use it as a pointer to a `struct bar`, and then recast it to a `struct mybar` without altering or losing the `name` field. For example,

```
bar    *b;
mybar *c = malloc( sizeof( mybar ) );
mybar *d;
char  *nm;
int    sz;

/* Assign the elements of c */

b = (struct bar *) c;
sz = b->size;
/* Use sz */

d = (struct mybar *) b;
nm = d->name;
/* Use nm */
```

This technique allows a `struct mybar` to be inserted into a list of `struct bars`, for example, manipulated by some functions expecting a `struct bar`, and eventually used again as a `struct mybar`.

2.2.3 C implementation issues

Two issues are raised by using C in relation to formal notations and in relation to networks of heterogeneous machines.

- An `int` is not an **integer**. Due to the fixed representation of an `int`, it has a limited range of values—a true **integer** does not. Due to the representation of a `double`, it is not a real **real** number; a `double` has limited precision and

limited range. In general, it is a good idea to ensure that **integers**, when used in formal notations that are aimed at non-trivial implementations, have upper and lower bounds.

- Different machines may have different representations for the same data types. For example, on one machine an **int** may be represented in 16 bits, with the least significant 8 bits at the lower addressed byte and the most significant 8 bits at the higher address. Another machine may use 32 bit **ints** with the order of bytes being, starting from the lowest address, most significant, second most significant, second least significant, and least significant. On most machines, a **short int** is 16 bits and a **long int** is 32 bits. Also, most implementations of the TCP/IP networking functions provide a set of macros **htonl**, **htons**, **ntohl**, and **ntohs**. It is best to use both specific types and the conversion macros when necessary.

Chapter 3

Abstract Protocols in C

The APC suite is designed to ease the transition from the abstract specification of a protocol in AP to the concrete implementation in C. It does this in two ways:

- Providing a simple structure for the implementation of actions, and
- Hiding some of the required but uninteresting bookkeeping.

For the moment, this discussion will ignore the **timeout** actions and a few other parts of AP. For information on those, see the next chapter.

The basic premise of the APC suite is to provide a central protocol “engine,” which is given C functions representing AP actions and which calls those functions when they are enabled. This call-back approach is used in several user interface systems where the application writer (or in this case, protocol writer) needs the ability to specify how a system will respond to an outside event, such as a key press or a message from another process.

3.1 Implementing processes

Each of the AP processes is mapped in the obvious way to a C program, complete with a function **main**. The program first initializes whatever variables are needed by the protocol, then initializes the suite. Then the program registers the functions implementing actions with the APC “engine.” All of this set-up work is simplified by the program only needing to know information about the local process. Once the set-up work is complete, the program activates the engine and starts the protocol proper. If and when the protocol terminates (another area not covered by the AP notation, but sometimes necessary), the engine cleans up after itself and the program exits.

Variables local to the process (and a few other things, as will be seen later) are translated into C variables at the file scope—global to the functions implementing the actions as well as the `main` function.

Initializing the suite is the task of the function `initialize_engine` which takes a single argument, an integer which is equivalent to the process name in the AP notation. This integer is used as an index into an array containing the information required to send messages to and receive messages from other processes. How `initialize_engine` determines this information is described in section 5.2.2. If `initialize_engine` discovers a problem, it exits with a return code. In order to simplify writing the program further, a macro, `INITIALIZE_ENGINE`, has been defined which calls `initialize_engine` and, in the event of an error, writes an error message to the standard error output and exits the running program.

Registering the actions is done by a pair of functions that are specific to the types of action; `add_local_action` is used for local actions; `add_receive_action` for receive actions. Both of these functions have error-handling macros, `ADD_LOCAL_FUNCTION` and `ADD_RECEIVE_ACTION`, respectively. Both of these functions take two arguments—the function pointers mentioned earlier. The first function argument implements the guard of whatever action is being registered. The second function argument implements the body of the action. Writing the guard and body functions is the subject of the next section.

Once the set-up work has been done, the program should call the function `engine` or its error-handling wrapper, `ENGINE`. `engine` takes no arguments.

3.1.1 Implementing actions

The implementation of an action consists of two functions, a guard function and a body function. For guard functions, there are two basic rules:

- The guard returns **true** if the guard predicate should evaluate to **true** and **false** otherwise. While there are some instances where the guard function will differ from the AP guard predicate, they should not differ radically.
- The guard function does not change the state of the protocol. It should not change the state of any variables (although the next chapter will consider a pair of seeming exceptions to this rule) and it should not send any messages.

If the guard predicate is simply the state of some boolean variable, a comparison of a variable and a value, or better still a boolean constant, the implementation of the guard is straightforward. Receive actions, however, require a little more work. The receive guard and body functions will be passed a structure when they are called

by the engine. This structure contains a process identifier, `sender`, which indicates the process sending the message, and a pointer to the message structure that was received. The receive guard will generally check the message type, sometimes check the sending process's id, and may also look at any additional fields that were part of the message as sent.

The body functions of both kinds of actions are very similar. These functions implement the right-hand side of the process's actions and change the state of the protocol by changing local variables and sending messages. The body function of a receive action does have available the structure passed to the guard in order to send replies or pick data out of the message.

Sending messages is handled by the function `send_message`, or its error-handling wrapper, `SEND_MESSAGE`. `send_message` takes two arguments, a pointer to a message structure that minimally has a `type` field and a `len` field. The `type` is used to send particular kinds of messages, and the `len` is used to inform the engine functions of how much data to send, i.e., the size of the message structure in bytes.

Receive functions, both the guard and the body, should not attempt to modify or delete their arguments. The memory for the arguments is managed by the engine itself.

When an action ascertains that the processing of the protocol is complete, for example that the data to be transferred has been transferred, it can set the variable `prctl_dne` to `true`. This will tell the engine that it should clean up after itself and terminate.

For more detailed information about the suite functions, see section 5.1.

3.2 Vending machine protocol

In order to demonstrate the basic use of these functions, a simple protocol will be defined and implemented.

The vending machine protocol consists of two processes, a vendor and a customer. The vending machine accepts money and item selections (in that order) from customers and returns items to them. The customer, a sort of generic individual, presents money and selections to the vending machine and accepts items in return.

The vending machine process is

```
process v
var readyv : boolean
begin

    rcv money from c → readyv := true
```

```

    || rcv selection from c → if readyv → send item to c
                               || ¬readyv → skip
                               fi;
                               readyv := false
end

```

and the customer process is

```

process c
var readyc : boolean
begin
    readyc → send money to v;
            send selection to v;
            readyc := false

    || rcv item from v → readyc := true
end

```

3.2.1 Process identifiers and message types

In order to properly implement the protocol, several constant values are required. C and V identify the process in question, the values of these constants are known as the process identifiers. Also, the protocol uses three different kinds of messages, MONEY, SELECTION, and ITEM. The value of these constants are the message types.

(Protocol constants 16) ≡

```

#define C    0
#define V    1

#define MONEY 1
#define SELECTION 2
#define ITEM 3
◇

```

Macro referenced in scraps 20a, 23a.

3.2.2 The vendor

In the specification of the vending machine protocol, the vending machine process (known from here on as the vendor) has one variable, a state **readyv**. The Boolean variable becomes an integer in traditional C.

The protocol specification does not mention process initialization. In order to properly start the process, it is reasonably clear that `readyv` should be **false**. The macro `FALSE` is defined to be zero, which in C is **false**.

⟨Vendor variables 17a⟩ ≡

```
int readyv = FALSE;
◇
```

Macro referenced in scrap 20a.

The vendor has two actions. The first allows the vending machine to accept money from the customer. The guard for the action (which is somewhat stricter than absolutely necessary—there is only one other process in the protocol) is **true** when the message type is `MONEY` and the message was sent by the customer, process id `C`.

The body of the first action sets `readyv` to `TRUE`, indicating that the vendor is ready to accept a selection and deliver an item. In the AP notation, the action is

rcv money from *c* → readyv := true

The C implementation is somewhat more complex, but in this instance is almost a direct translation. The value `rcvd` passed to the guard when the engine points to a structure with two components. The first component points to the message that was sent by the other process, `msg`. The second component, `sender`, is the process identifier of the sending process. The message can have several fields, depending on the requirements of the protocol, but it should have a field, `type`, indicating what message it is. The guard returns either zero or non-zero depending on whether the vendor has received a `MONEY` message and on whether the sender is the process `C`. If those conditions are fulfilled, the guard will return a non-zero value and the engine will call the corresponding body function which sets `readyv` to **true**. (For the moment, ignore the code scraps labelled “⟨Elaborate...⟩.” They do nothing to change the state of the protocol.)

In order to inform the protocol engine of the vendor’s action, the vendor will need to call `ADD_RECEIVE_ACTION` with the guard and body functions for the action.

⟨Vendor actions 17b⟩ ≡

```

int action1g( RCVD_MSG *rcvd )
{
    return( rcvd->msg->type == MONEY && rcvd->sender == C );
}

void action1b( RCVD_MSG *rcvd )
{
    readyv = TRUE;
    ⟨Elaborate on vendor action 1 23b⟩
}
◇

```

Macro defined by scraps 17b, 19a.
Macro referenced in scrap 20a.

⟨Add vendor actions 18⟩ ≡

```

ADD_RECEIVE_ACTION( action1g, action1b );
◇

```

Macro defined by scraps 18, 19b.
Macro referenced in scrap 20a.

The vending machine's second action is used to respond to selections from the customer. The guard function therefore returns **true** when the message received has a type of **SELECTION** and the sender is the customer, process id **C**.

In the AP notation, the second action is

```

rcv selection from c → if readyv → send item to c
    | ¬readyv → skip
fi;
    readyv := false

```

The vendor's response to the selection varies; if **MONEY** has been sent previously, then **readyv** will be **true** and the vendor will send a message **ITEM** to the customer. If **MONEY** has not been sent, **readyv** is **false** and the vendor will not send a message **ITEM**. In either case, **readyv** is set to **false**.

The body function contains the odd looking line,

```

MSG_BUFFER msg = {ITEM, sizeof( MSG_BUFFER )};

```

which defines and initializes a C variable, `msg`. The `MSG_BUFF` structure has two elements, the `type` mentioned above and the size of the structure in bytes, the `len`. For the message being sent by this function, the only `type` being used is `ITEM`. The size of the structure is generally constant and generally set with `sizeof`. The `len` is needed by `send_message`, which would not otherwise know how many bytes to send to the other process, in this case, `C`.

This example demonstrates that, while the syntax of the AP notation and the C implementation are different, the difference at this stage is largely a matter of format, terminology, and a little record keeping.

⟨Vendor actions 19a⟩ ≡

```
int action2g( RCVD_MSG *rcvd )
{
    return( rcvd->msg->type == SELECTION && rcvd->sender == C );
}
```

```
void action2b( RCVD_MSG *rcvd )
{
    MSG_BUFF msg = {ITEM, sizeof( MSG_BUFF )};
```

⟨Elaborate on vendor action 2, pt. 1 23c⟩

```
if (readyv)
{
    SEND_MESSAGE( &msg, C );
    ⟨Elaborate on vendor action 2, pt. 2 23d⟩
}
readyv = FALSE;
}
```

◇

Macro defined by scraps 17b, 19a.
Macro referenced in scrap 20a.

⟨Add vendor actions 19b⟩ ≡

```
ADD_RECEIVE_ACTION( action2g, action2b );
```

◇

Macro defined by scraps 18, 19b.
Macro referenced in scrap 20a.

The vending machine process defines all of the information above and in its function, `main`, initializes the engine with the process identifier `V` and the actions, and calls the engine.

"v.c" 20a ≡

```
#include <stdio.h>
#include "APC.h"

<Protocol constants 16>
<Vendor variables 17a>
<Vendor actions 17b, ... >

void main()
{
    INITIALIZE_ENGINE( V );
    <Add vendor actions 18, ... >
    ENGINE();
}
◇
```

3.2.3 The customer

Similar to the vending machine, the customer has one state variable, `readyc`, which becomes an integer in C.

<Customer variables 20b> ≡

```
int readyc = TRUE;
◇
```

Macro referenced in scrap 23a.

The customer's first action is of a different type than the vendor's two. Rather than receiving a message, its guard is based only on the process's local state variable.

When `readyc` is **true**, the customer sends money followed by a selection to the vending machine and sets `readyc` to **false**.

```

readyc → send money to v;
          send selection to v;
          readyc := false

```

Rather than using two variables to define the messages, the implementation uses one, `msg`. `msg`'s `len` is set to the size of the structure, and its `type` is first set to `MONEY` and then to `SELECTION`.

Notice that the customer process uses `ADD_LOCAL_ACTION` to notify the engine of the action, rather than `ADD_RECEIVE_ACTION`.

⟨Customer actions 21a⟩ ≡

```

int action1g( void )
{
    return( readyc );
}

void action1b( void )
{
    MSG_BUFF msg;

    msg.len = sizeof( MSG_BUFF );

    msg.type = MONEY;
    SEND_MESSAGE( &msg, V );
    ⟨Elaborate on customer action 1, pt. 1 24a⟩

    msg.type = SELECTION;
    SEND_MESSAGE( &msg, V );
    ⟨Elaborate on customer action 1, pt 2 24b⟩

    readyc = FALSE;
}
◇

```

Macro defined by scraps 21a, 22a.
 Macro referenced in scrap 23a.

⟨Add customer actions 21b⟩ ≡

```

ADD_LOCAL_ACTION( action1g, action1b );
◇

```

Macro defined by scraps 21b, 22b.

Macro referenced in scrap 23a.

The customer's second action is a receive action. The guard becomes **true** when the customer receives an **ITEM** message from the vending machine, process **V**.

When the customer process receives an item from the vending machine, the customer becomes ready again, continuing the great tradition of customers everywhere.

Compare the AP notation of this final action,

rcv item from v → readyc := true

with the implementation.

⟨Customer actions 22a⟩ ≡

```
int action2g( RCVD_MSG *rcvd )
{
    return( rcvd->msg->type == ITEM && rcvd->sender == V );
}

void action2b( RCVD_MSG *rcvd )
{
    readyc = TRUE;
    ⟨Elaborate on customer action 2 24c⟩
}
◇
```

Macro defined by scraps 21a, 22a.
Macro referenced in scrap 23a.

⟨Add customer actions 22b⟩ ≡

```
ADD_RECEIVE_ACTION( action2g, action2b );
◇
```

Macro defined by scraps 21b, 22b.
Macro referenced in scrap 23a.

Overall, the customer process follows basically the same steps as the vending machine process. The major difference is that the customer initializes its engine with the process id **C**.

```
"c.c" 23a ≡
```

```
#include <stdio.h>
#include "APC.h"

<Protocol constants 16>
<Customer variables 20b>
<Customer actions 21a, ... >

void main()
{
    INITIALIZE_ENGINE( C );
    <Add customer actions 21b, ... >
    ENGINE();
}
◇
```

3.2.4 Elaborations

The programs as written above are correct, but somewhat uninteresting. Neither of the programs produces any visible output despite the fact that they are exchanging messages at a furious rate. Therefore, the following elaborations have been added in order to show the activity of the processes. Refer back to the implementations above if the printing statements described here are not understandable.

```
<Elaborate on vendor action 1 23b> ≡
```

```
printf( "Received money\n" );
◇
```

Macro referenced in scrap 17b.

```
<Elaborate on vendor action 2, pt. 1 23c> ≡
```

```
printf( "Received selection\n" );
◇
```

Macro referenced in scrap 19a.

```
<Elaborate on vendor action 2, pt. 2 23d> ≡
```

```
printf( "Sent item\n" );  
◇
```

Macro referenced in scrap 19a.

(Elaborate on customer action 1, pt. 1 24a) ≡

```
printf( "Sending money\n" );  
◇
```

Macro referenced in scrap 21a.

(Elaborate on customer action 1, pt 2 24b) ≡

```
printf( "Sending selection\n" );  
◇
```

Macro referenced in scrap 21a.

(Elaborate on customer action 2 24c) ≡

```
printf( "Received item\n" );  
◇
```

Macro referenced in scrap 22a.



3.3 The request/reply protocol

The vending machine protocol is a particular instance of a very broad class of protocols—client/server protocols. Client/server protocols generally feature a specific, well-known process which provides services to other, client processes. These protocols are, in turn, a subset of request/reply protocols in which one or more processes make requests and receive replies from each other.

In order to provide a larger example of how to use the APC suite, let us take a simple request/reply protocol and, in subsequent sections, elaborate on it in order to show the usefulness of the APC suite in approaching more complex AP constructions.

For this simple protocol, define the process *p* as

```
process p  
var ready : boolean  
begin
```

```

    readyp → send request to q;
           readyp := false

  || rcv request from q → send reply to q
  || rcv reply from q → readyp := true
end

```

The process *q* is symmetrical.

3.3.1 Process identifiers and message types

The basic request/reply protocol needs two processes, *p* and *q*, and therefore the two process identifiers **P** and **Q**. It also has two different kinds of messages, **REQUEST**'s and **REPLY**'s. For variety's sake, the numerical constants associated with the message types are not 0 and 1.

Since this protocol will be expanded on in later sections, the string “pass *n*” is added to the end of the scrap names in order to prevent them from conflicting with different versions of the same scrap.

(Request/reply protocol constants, pass 1 25) ≡

```

#define P 0
#define Q 1

#define REQUEST 44
#define REPLY 54
◇

```

Macro referenced in scrap 29.

The first advanced feature that this protocol demonstrates is the use of specific message types, rather than the generic **MSG_BUFF**. Note that, however, the two fields of the types are the same as type fields of **MSG_BUFF**. In practice, it is a good idea to use the names **type** and **len** for the fields, but only the order and the size of the two fields matter; the type of the message is assumed to be in the first **sizeof(long)** bytes of a received message and the size of the whole message is assumed to be in the second **sizeof(long)** bytes. This technique is not useful in this particular case, but will become important later when a message type will need one or more extra fields.

⟨Message types, pass 1 26a⟩ ≡

```
typedef struct
{
    long type;
    long len;
} REQUEST_MSG;
```

```
typedef struct
{
    long type;
    long len;
} REPLY_MSG;
```

◇

Macro referenced in scrap 29.

3.3.2 Process *p*

When the process is ready, it will generate a request message to be sent to the other process, *q*.

```
readyp → send request to q;  
         readyp := false
```

⟨Process variables, pass 1 26b⟩ ≡

```
int readyp = TRUE;
```

◇

Macro referenced in scrap 29.

P's first action reflects the use of *ready_p* above; when *ready_p* is **true**, *P* sends a request to *Q*.

⟨*P*'s actions, pass 1 26c⟩ ≡

```
int action1g( void )
{
    if (readyp) return( TRUE );
```

```

    return( FALSE );
}

void action1b( void )
{
    REQUEST_MSG msg;

    msg.type = REQUEST;
    msg.len = sizeof( REQUEST_MSG );
    SEND_MESSAGE( (MSG_BUFFER *) &msg, Q );
    ready = FALSE;

    printf( "Sending request\n" );
}
◇

```

Macro defined by scraps 26c, 27b, 28b.
 Macro referenced in scrap 29.

⟨Add action 1, pass 1 27a⟩ ≡

```

    ADD_LOCAL_ACTION( action1g, action1b );
◇

```

Macro referenced in scrap 29.

In order to uphold P's end of the protocol, when the process receives a request from Q it needs to send a reply. This is a very generic reply, however, not involving any actual information transfer.

rcv request from q → *send reply to q*

⟨P's actions, pass 1 27b⟩ ≡

```

int action2g( RCVD_MSG *rcvd )
{
    return( rcvd->msg->type == REQUEST && rcvd->sender == Q );
}

void action2b( RCVD_MSG *rcvd )
{
    REPLY_MSG msg;
}

```

```

msg.type = REPLY;
msg.len = sizeof( REPLY_MSG );
SEND_MESSAGE( (MSG_BUFF *) &msg, Q );

printf( "Sending reply: %d\n", msg.slot );
}

```

◇

Macro defined by scraps 26c, 27b, 28b.
Macro referenced in scrap 29.

⟨Add action 2, pass 1 28a⟩ ≡

```

ADD_RECEIVE_ACTION( action2g, action2b );

```

◇

Macro referenced in scrap 29.

Finally, when P receives a reply to its request from Q, it can become ready again.

rcv reply from q → *ready_p* := true

⟨P's actions, pass 1 28b⟩ ≡

```

int action3g( RCVD_MSG *rcvd )
{
if (rcvd->msg->type == REPLY && rcvd->sender == Q) return( TRUE );
return( FALSE );
}

```

```

void action3b( RCVD_MSG *rcvd )
{
readyp = TRUE;

printf( "Receiving reply: %d\n", j );
}

```

◇

Macro defined by scraps 26c, 27b, 28b.
Macro referenced in scrap 29.

⟨Add action 3, pass 1 28c⟩ ≡

```
ADD_RECEIVE_ACTION( action3g, action3b );
```

◇

Macro referenced in scrap 29.

The implementation of P needs to

- include the APC suite header file and the `stdio.h` header,
 - define its constants, types, variables, and actions,
 - initialize the engine,
 - inform the engine of the actions of the process,
 - and call the engine itself.
-

"p1.c" 29 ≡

```
#include "APC.h"
#include <stdio.h>

<Request/reply protocol constants, pass 1 25>
<Message types, pass 1 26a>
<Process variables, pass 1 26b>

<P's actions, pass 1 26c, ... >

int main()
{
    INITIALIZE_ENGINE( P );
    <Add action 1, pass 1 27a>
    <Add action 2, pass 1 28a>
    <Add action 3, pass 1 28c>
    ENGINE();
    return( 0 );
}
◇
```

Like the specification of *p*, the implementation of *Q* is symmetrical to that of *P*. (This is a very useful phrase. In this instance, it means that *Q* can be constructed from *P* through the simple expedient of swapping the characters ‘p’ and ‘q’ in appropriate places.)

Chapter 4

Parameters, Timeouts, and Process Arrays

There are three major AP implementation issues which have not been discussed thus far. These are parameters, timeouts, and process arrays.

Parameters are actually rather simple to implement, although the idea behind them may not be obvious. In most cases parameters are used to allow the enabling of an action which picks some value nondeterministically from some range. Therefore, it is possible to implement them by defining a *C* variable for them and *setting* their value in the *guard* function of their action. This is a violation of the rules presented above, but only technically—the parameters are not used to determine the state of the protocol. Therefore, the action presented above,

```
readyp[j] → ⟨do something⟩
```

where *j* is a parameter ranging over $0 \dots N$, can become

```
int guard(void)
{
  for (j = 0; j <= N; j++) if (readyp[j]) return( TRUE );
  return( FALSE );
}
```

which leaves *j* set to the appropriate value for the processing of the body function.

Process arrays are similarly easy to handle. Since the process identifier used by each process is already implemented as an integer beginning at 0 for some process and ranging upwards for each of the rest, it is reasonable to skip the array part entirely and use the process array index from the specification as the process

identifier. This will generally require reading the identifier for each process from the command line, or some similar method which assigns it at run time. The suite makes the variable `proc_count` available to the actions—`proc_count` contains the number of processes participating in the protocol.

Timeouts, however, are somewhat more complex. Since the global information available to their predicates is not available to the implemented program, it is necessary to replace that global information with locally available information. The only truly global information that is available to a process is the current time. The global predicate of the specification should be replaced in the implementation with a delay in the execution of the timeout action which will hopefully have the same effect as the global predicate. For example, this often means that the implementation, rather than resending a message when that message has been lost, must send a replacement message after enough time has passed to indicate that the message has probably been lost.

The mechanics of timeout actions are rather simple. They are implemented as two functions in the same way as local and receive actions. Since the global information used by timeout implementations is a delay value, this delay is specified as one of the arguments to `add_timeout_action`, the function that notifies the suite of the presence of a timeout. Since the delay of the timeout execution must begin at a reasonable point, `add_timeout_action` should only be called from the body of another action; for example, right after the sending of a message which may need to be resent. Also, since the continued re-execution of a given timeout would not be useful in most cases, the timeout action set by `add_timeout_action` is a one shot affair; after the delay, after the guard of the action is attempted, no matter whether the body of the action was executed or not, the record of the pointers to the guard and body functions is deleted from the engine.

As a slight service to the timeouts, an extra parameter, `data`, can be associated with the action functions when they are added to the engine. This parameter is a `long` value which is passed on to the timeout functions when they are invoked.

4.1 Request/reply protocol revisited

The request/reply protocol in section 3.3 is certainly interesting, but it is not very general. An immediate idea that springs to mind is to allow each process to have more than one outstanding request. For example, process p might need to send N requests to process q without waiting; likewise, q could send N request to p before becoming unable to send further requests.

A process p with this ability might look like

```

process p
inp N : integer
var readyp : array [0...N - 1] of boolean
par j : 0...N - 1
begin
    readyp[j] → send request (j) to q;
                    readyp[j] := false

    || rcv request (j) from q → send reply (j) to q

    || rcv reply (j) from q → readyp[j] := true
end

```

Remember that a protocol parameter, such as j above, turns any action it is used in into an “action template,” which is replaced, in this case, with N nearly identical actions. The only difference between the new actions is that j is replaced by one (and only one, for each instance of j in a particular new action) of the values from its range.

4.1.1 Process identifiers, message types, and new constants

This version of p has the constants and message types of the previous one, but it adds a constant \mathbf{N} , which represents the number of outstanding requests that \mathbf{P} can have.

In the current version of the protocol, each message has a field into which a value is placed and from which it is read. This value corresponds with one of the elements of the array *ready_p*. The message types for the implementation, therefore, need an additional field which can be used to index into the array *ready_p*. This field is known as `slot`.

⟨Protocol constants, pass 2 33⟩ ≡

```

#define P 0
#define Q 1

#define REQUEST 44
#define REPLY 45

#define N 5
◇

```

Macro referenced in scrap 38b.

⟨Message types, pass 2 34a⟩ ≡

```
typedef struct
{
    int type;
    int len;
    int slot;
} REQUEST_MSG;
```

```
typedef struct
{
    int type;
    int len;
    int slot;
} REPLY_MSG;
```

◇

Macro referenced in scrap 38b.

4.1.2 Process *p*

In this version of the request/reply protocol process *P*, the `readyp` state variable has been expanded to be an array. Also, a variable which is used to represent the parameter *j* is added.

Because `readyp` is now an array, it requires some explicit initialization code.

⟨Process variables, pass 2 34b⟩ ≡

```
int readyp[N];
int j;
```

◇

Macro referenced in scrap 38b.

⟨Initialize variables, pass 2 34c⟩ ≡

```
for (i = 0; i < N; i++)
{
    readyp[i] = TRUE;
}
```

◇

Macro referenced in scrap 38b.

p 's first action represents a serious departure from the previous protocol. The AP specification of it is

```
ready $p$ [ $j$ ] → send request( $j$ ) to  $q$ ;  
                  ready $p$ [ $j$ ] = false
```

remembering that j is a parameter. There would be two ways of implementing this action. The first would be to make N copies of it, with each value from $0 \dots N$ replacing j in some fashion. This is clearly the wrong way to go; it would require multiple copies of the code with only minor changes and it would break if the value of N was changed.

The alternative is to implement j as a variable, as was done with j . Then, when j was effectively set (actually, the action was chosen with j having an appropriate value), the implementation sets j to the correct value. In the case of this action, j is used to test the values of **ready p** until one is found which is **true** or until all have been checked. The function implementing the body of the action also uses the value of j set in the guard. This use of variables to substitute for parameters is one exception to the rule that the functions implementing action guards should not have any side effects.

The body, by the way, sets the fields of **msg** correctly and then passes the thing to **send_message**. The cast of the address of **msg** to a pointer to a **MSG_BUFF** is required since that is what **send_message** expects, and it is safe since the resulting address actually does point to a **MSG_BUFF** (plus an extra field tacked on the end).

$\langle P$'s actions, pass 2 35 $\rangle \equiv$

```
int action1g( void )  
{  
  for (j = 0; j < N; j++) if (ready $p$ [j]) return( TRUE );  
  return( FALSE );  
}  
  
void action1b( void )  
{  
  REQUEST_MSG msg;
```

```

msg.type = REQUEST;
msg.len = sizeof( REQUEST_MSG );
msg.slot = j;
SEND_MESSAGE( (MSG_BUFF *) &msg, Q );
readyp[j] = FALSE;

printf( "Sending request: %d\n", j );
}
◇

```

Macro defined by scraps 35, 36b, 37b.
Macro referenced in scrap 38b.

⟨Add action 1, pass 2 36a⟩ ≡

```

ADD_LOCAL_ACTION( action1g, action1b );
◇

```

Macro referenced in scrap 38b.

The second action,

rcv request (j) from q → send reply (j) to q

is implemented in a way very similar to the first, except that the value of *j* is simply set from the field of the received message. A loop such as that above is not required here to give *j* its parameter nature.

⟨*P*'s actions, pass 2 36b⟩ ≡

```

int action2g( RCVD_MSG *rcvd )
{
    REQUEST_MSG *req;

    if ( rcvd->msg->type == REQUEST && rcvd->sender == Q )
    {
        req = (REQUEST_MSG *) rcvd->msg;
        j = req->slot;
        return( TRUE );
    }
    return( FALSE );
}

```

```

void action2b( RCVD_MSG *rcvd )
{
    REPLY_MSG    msg;

    msg.type = REPLY;
    msg.len = sizeof( REPLY_MSG );
    msg.slot = j;
    SEND_MESSAGE( (MSG_BUFF *) &msg, Q );

    printf( "Sending reply: %d\n", msg.slot );
}

```

◇

Macro defined by scraps 35, 36b, 37b.
 Macro referenced in scrap 38b.

⟨Add action 2, pass 2 37a⟩ ≡

```

    ADD_RECEIVE_ACTION( action2g, action2b );

```

◇

Macro referenced in scrap 38b.

Finally, p 's third action is

rcv reply (j) from $q \rightarrow ready_p[j] = \text{true}$

and it is implemented much like the second action.

⟨ P 's actions, pass 2 37b⟩ ≡

```

int action3g( RCVD_MSG *rcvd )
{
    REPLY_MSG *reply;

    if (rcvd->msg->type == REPLY && rcvd->sender == Q)
    {
        reply = (REPLY_MSG *) rcvd->msg;
        j = reply->slot;
        return( TRUE );
    }
}

```

```

    return( FALSE );
}

void action3b( RCVD_MSG *rcvd )
{
    readyp[j] = TRUE;

    printf( "Receiving reply: %d\n", j );
}

```

◇

Macro defined by scraps 35, 36b, 37b.
 Macro referenced in scrap 38b.

⟨Add action 3, pass 2 38a⟩ ≡

```

    ADD_RECEIVE_ACTION( action3g, action3b );

```

◇

Macro referenced in scrap 38b.

The major change to the remaining parts of the implementation is the scrap to initialize the values of `readyp`.

"p2.c" 38b ≡

```

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

⟨Protocol constants, pass 2 33⟩
⟨Message types, pass 2 34a⟩
⟨Process variables, pass 2 34b⟩

⟨P's actions, pass 2 35, ... ⟩

void main()
{
    int i;

    ⟨Initialize variables, pass 2 34c⟩

    INITIALIZE_ENGINE( P );
    ⟨Add action 1, pass 2 36a⟩

```

```

    <Add action 2, pass 2 37a>
    <Add action 3, pass 2 38a>
    ENGINE();
  }
◇

```

As before, the implementation of q is symmetrical and only requires some substitutions.

4.2 Reliable request/reply

The request/reply protocol described in section 4.1 suffers from one fatal flaw; it will not resend lost messages, and therefore is unreliable.

The timeout mechanism can be used to correct this flaw. This requires considerable changes in both the protocol specification and in the implementation. The protocol now looks like:

```

process  $p$ 
inp  $N$  : integer
var  $ready_p$  : array  $[0 \dots N - 1]$  of boolean
par  $j$  :  $0 \dots N - 1$ 
begin
     $ready_p[j] \rightarrow$  send  $request(j)$  to  $q$ ;
     $ready_p[j] :=$  false

    || rcv  $request(j)$  from  $q \rightarrow$  send  $reply(j)$  to  $q$ 

    || rcv  $reply(j)$  from  $q \rightarrow$   $ready_p[j] :=$  true

    || timeout  $\neg ready_p[j] \wedge request(j) \#C.p.q = 0 \wedge reply(j) \#C.q.p = 0 \rightarrow$ 
    send  $request(j)$  to  $q$ 
end

```

The last action (whose guard is true when a slot is marked as in use and there is no corresponding message, either the *request* or a *reply* in the channel between p and q) is used to resend a request after either the request or the reply has been lost.

The change to the process has several far-reaching consequences to the program.

4.2.1 Constants and message types

In this case, the constants and message types needed by the implementation have not changed from the last version.

⟨Protocol constants, pass 3 40a⟩ ≡

```
#define P 0
#define Q 1

#define REQUEST 44
#define REPLY 45

#define N 5
◇
```

Macro referenced in scrap 47c.

⟨Message types, pass 3 40b⟩ ≡

```
typedef struct
{
    int type;
    int len;
    int slot;
} REQUEST_MSG;

typedef struct
{
    int type;
    int len;
    int slot;
} REPLY_MSG;
◇
```

Macro referenced in scrap 47c.

4.2.2 Process *p*

The variables, however, have changed considerably. Rather than a single boolean value or an array of boolean values, the *ready_p* state variable becomes an array (with one element for each possible message `slot` value) of structures—the first

element of the structure is the `ready` flag, which does actually serve the purpose of the previous `ready` array. The other two slots are used in the processing of the timeout for resending requests.

The initialization of the `ready` variable is as complex as the variable itself. `j`, however, is still used the same as it was, and seems rather innocuous now.

⟨Process variables, pass 3 41a⟩ ≡

```
struct
{
    int ready;
    int sent;
    int timed;
} ready[N];
int j;
◇
```

Macro referenced in scrap 47c.

⟨Initialize variables, pass 3 41b⟩ ≡

```
for (i = 0; i < N; i++)
{
    ready[i].ready = TRUE;
    ready[i].sent = 0;
    ready[i].timed = 0;
}
◇
```

Macro referenced in scrap 47c.

The first complex change required for the timeout implementation is in the very first action. Recall that the AP timeouts have access to global information about the state of the protocol. The global information required in this protocol is the state of the channel between p and q . The implementation clearly cannot easily gain access to that global information, and therefore must use with the information that it has available, the time.

In the APC suite, the timeout handlers use this timing information to allow the use of timeout actions by specifying a delay after a certain time during a protocol's execution. After the delay expires, the engine will attempt to execute the timeout action by testing the guard function. This use of a delay, however, requires a starting time—the time after which the delay will expire.

The choice of this time requires in-depth knowledge of the protocol. In the case of this protocol, the timeout action is being used to resend a presumably missing message. Therefore, the start time of the delay should be the time the original message is sent. In this way, the `add_timeout_action` function differs from the other `add_..._action` functions. The others are called from the `main` function before invoking the engine and are set up permanently. `add_timeout_action` is called from the place in one of the protocol actions which represents its start time. Also, after a call to `add_local_action`, the guard and body functions of the local action remain in effect for the duration of the process; a particular invocation of `add_timeout_action` results in one specific invocation of the timeout guard and possibly body actions after a given delay.

One difficulty arises when using the timeout action implementation system presented here—the timeout action functions are too limited to know when not to work. This, then, is the reason for the additional fields (after `ready`) in the `readyp` array. The protocol needs some way of telling a timeout action that it is not actually necessary to resend a message. In the action below, after setting `ready` to `false` for a slot, the process increments a counter of the number of times a message bearing that slot number has been sent. For more information on this topic, see the discussion of action 4 below.

For comparison, the AP specification for this action is

```
readyp[j] → send request(j) to q;
           ready[j] := false
```

⟨*P*'s actions, pass 3 42⟩ ≡

```
int action1g( void )
{
    for (j = 0; j < N; j++) if (readyp[j].ready) return( TRUE );
    return( FALSE );
}

void action1b( void )
{
    REQUEST_MSG msg;

    msg.type = REQUEST;
    msg.len = sizeof( REQUEST_MSG );
    msg.slot = j;
```

```

SEND_MESSAGE( (MSG_BUFF *) &msg, Q );
⟨Add action 4, pass 3 47a⟩
readyp[j].ready = FALSE;
readyp[j].sent++;

printf( "Sending request: %d\n", j );
}

```

◇

Macro defined by scraps 42, 43b, 44b, 46.
 Macro referenced in scrap 47c.

⟨Add action 1, pass 3 43a⟩ ≡

```

ADD_LOCAL_ACTION( action1g, action1b );

```

◇

Macro referenced in scrap 47c.

Action 2 is specified as

rcv request (j) from q → *send reply (j) to q*

and the implementation here does not differ from it more than the implementation in the previous version did. One characteristic of this protocol that is worth noting is that loss of reply messages is handled in the same way as the loss of request messages. The sender of the request notices that no reply is forthcoming and resends the request. No additional processing is required on the replying end.

⟨P's actions, pass 3 43b⟩ ≡

```

int action2g( RCVD_MSG *rcvd )
{
    REQUEST_MSG *req;

    if ( rcvd->msg->type == REQUEST && rcvd->sender == Q )
    {
        req = (REQUEST_MSG *) rcvd->msg;
        j = req->slot;
        return( TRUE );
    }
    return( FALSE );
}

```

```

    }

void action2b( RCVD_MSG *rcvd )
{
    REPLY_MSG    msg;

    msg.type = REPLY;
    msg.len = sizeof( REPLY_MSG );
    msg.slot = j;
    SEND_MESSAGE( (MSG_BUFF *) &msg, Q );

    printf( "Sending reply: %d\n", msg.slot );
}

```

◇

Macro defined by scraps 42, 43b, 44b, 46.
 Macro referenced in scrap 47c.

⟨Add action 2, pass 3 44a⟩ ≡

```

    ADD_RECEIVE_ACTION( action2g, action2b );

```

◇

Macro referenced in scrap 47c.

Action 3 does not materially differ from the third action of the previous version of the request/reply protocol. The actual difference is the use of the **ready** field of the **readyp** array rather than a whole element of the array.

This action does not do anything with the other two slots of the array. By setting the **ready** slot to **false** this action temporarily removes the problem (unneeded resends of request messages will not be done if there are no outstanding requests). It is handled more permanently by action 1 incrementing the **sent** count when that action brings the problem up again.

⟨P's actions, pass 3 44b⟩ ≡

```

int action3g( RCVD_MSG *rcvd )
{
    REPLY_MSG *reply;

    if (rcvd->msg->type == REPLY)
    {
        reply = (REPLY_MSG *) rcvd->msg;
    }
}

```

```

        j = reply->slot;
        return( TRUE );
    }
    return( FALSE );
}

void action3b( RCVD_MSG *rcvd )
{
    readyp[j].ready = TRUE;

    printf( "Receiving reply: %d\n", j );
}

```

◇

Macro defined by scraps 42, 43b, 44b, 46.
 Macro referenced in scrap 47c.

⟨Add action 3, pass 3 45⟩ ≡

```

    ADD_RECEIVE_ACTION( action3g, action3b );

```

◇

Macro referenced in scrap 47c.

The timeout action specified in the protocol above is

timeout $\neg readyp[j] \wedge request(j) \#C.p.q = 0 \wedge reply(j) \#C.q.p = 0 \rightarrow$
send *request(j)* to *q*

The implementation, or at least the guard of it, is radically different.

First, the check of the **ready** element is similar to the AP action. However, the other two conjuncts of the AP guard are handled by the delay associated with the timeout implementation. For the following discussion, assume that **ready** is **true**, i.e., that there is an outstanding request.

The implementation imposes some other processing on the guard function and adds a new conjunct. The increment of the **timed** element is used to insure that unneeded message resends do not occur. In its simplest form,

- the **sent** element counts the number of request messages sent by this process for each slot of **readyp**,
- the **timed** element counts the number of timeout action delay expirations that have occurred for each slot of **readyp**;

- and if `sent` equals `timed` then the currently expiring delay (associated with this invocation of the guard function) refers to the currently outstanding request,
- in which case the request or reply message is missing (inferred from the delay) and needs to be resent.

The `d` used by the guard is the general data item recorded by `add_timeout_action`. In this process it is used to let this timeout guard know which `readyp` element is expiring.

The body function of this action simply resends the message; and because the resent message counts as a request, it increments the `sent` counter and resets a timeout.

The call to `add_timeout_action` defined here is used in the timeout body function as well as the body of action 1. It records the value of `j` in both cases, remembering the slot number from which this request is being sent. The final parameter to the call is the delay associated with this timeout. The value of 50000 microseconds is an arbitrary choice—it needs to be large enough to actually indicate that a message has been lost rather than delayed but small enough to insure that the protocol does not suffer too greatly in the event that a resend is needed.

Finally, action 4’s guard and body functions are declared early in the program file to satisfy the C compiler as to how they are called, since they are used before they are defined.

$\langle P$'s actions, pass 3 46 $\rangle \equiv$

```
int action4g( long d )
{
    readyp[d].timed++;
    if (!readyp[d].ready && readyp[d].timed == readyp[d].sent)
    {
        j = d;
        return( TRUE );
    }
    return( FALSE );
}

void action4b( long d )
{
    REQUEST_MSG msg;

    msg.type = REQUEST;
```

```

    msg.len = sizeof( REQUEST_MSG );
    msg.slot = j;
    SEND_MESSAGE( (MSG_BUFF *) &msg, Q );
    ⟨Add action 4, pass 3 47a⟩
    readyp[j].sent++;

    printf( "Resending request: %d\n", j );
    }
◇

```

Macro defined by scraps 42, 43b, 44b, 46.
 Macro referenced in scrap 47c.

⟨Add action 4, pass 3 47a⟩ ≡

```

    ADD_TIMEOUT_ACTION( action4g, action4b, j, 50000 );
◇

```

Macro referenced in scraps 42, 46.

⟨Declare action 4, pass 3 47b⟩ ≡

```

    int action4g( long d );
    void action4b( long d );
◇

```

Macro referenced in scrap 47c.

The program file does not require many changes to use the timeout handlers of the APC suite.

"p3.c" 47c ≡

```

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

⟨Protocol constants, pass 3 40a⟩
⟨Message types, pass 3 40b⟩
⟨Process variables, pass 3 41a⟩

⟨Declare action 4, pass 3 47b⟩

⟨P's actions, pass 3 42, ... ⟩

```

```

void main()
{
  int i;

  ⟨Initialize variables, pass 3 41b⟩

  INITIALIZE_ENGINE( P );
  ⟨Add action 1, pass 3 43a⟩
  ⟨Add action 2, pass 3 44a⟩
  ⟨Add action 3, pass 3 45⟩
  ENGINE();
}
◇

```

As always, the process q is symmetrical to this process.

4.3 Request/reply using multiple processes

Another extension to the previous request/reply protocol of section 4.2 that comes to mind is to use more than two processes; suppose M processes are created which spread requests among themselves in order to distribute some processing load. Each of the M processes could have N requests outstanding among the collection.

This extension introduces process arrays, which require only a small change in the implementation strategy. On the other hand, in order to improve the flexibility of the implementation, it reads the values for M and N from the command line—therefore, the number of processes and the number of outstanding messages per process may be set at runtime. Also, because this protocol uses a process array, all of the processes are identical. The only difference is in the process identifiers used by each, and this information can be read from the command line as well.

The protocol is

```

process  $p$  [ $i : 0 \dots M - 1$ ]
inp  $N$  : integer {# of outstanding requests}
var  $ready$  : array [ $0 \dots N - 1$ ] of  $0 \dots M$ ;
     $k$  :  $0 \dots M - 1$ ; {Next process for requests}
     $l$  :  $0 \dots M - 1$ 
par  $j$  :  $0 \dots N - 1$ 
begin

```

```

readyyp[j] = M → k := (k + 1) mod M;
                if k = i → k := (k + 1) mod M
                || k ≠ i → skip
                fi;
                readyyp[j] := k;
                send request(j) to p[k]

|| rcv request(j) from p[l] → send reply(j) to p[l]

|| rcv reply(j) from p[l] → readyyp[j] := M

|| timeout readyyp[j] ≠ M
    ∧ request(j) #C.p[i].p[readyyp[j]] = 0
    ∧ reply(j) #C.p[readyyp[j]].p[i] = 0 →
    send request(j) to p[readyyp[j]]

```

end

In this instance, there is no process q to be symmetrical to p .

One change to the protocol, from the previous versions, is evident. The elements of *readyyp* are not now boolean; they are used to store the process identifier to which the corresponding message has been sent. A special value, M , is used to indicate that the slot is available; it is not a valid process array index.

4.3.1 Constants and message types

The constants and message types remain almost unchanged from the last version of the request/reply protocol. Notably, \mathbb{N} is absent. By reading it from the command line, the implementation requires \mathbb{N} to be a variable although after it is set from the command line it is not changed.

(Protocol constants, pass 4 49) ≡

```

#define REQUEST 44
#define REPLY 45
◇

```

Macro referenced in scrap 56c.

⟨Message types, pass 4 50a⟩ ≡

```
typedef struct
{
    int type;
    int len;
    int slot;
} REQUEST_MSG;
```

```
typedef struct
{
    int type;
    int len;
    int slot;
} REPLY_MSG;
```

◇

Macro referenced in scrap 56c.

4.3.2 Process p

The variables and structure of the process are also unchanged—almost. Additional variables are i , used to hold the current processes' process identifier; M and N , used to hold the constants used by the process; and k , which is used to spread the requests around the process array fairly.

The initialization is slightly more complex. i , N , and M are read from the command line arguments and converted to integers. The `ready` array is now allocated dynamically from the known size of the structures making up its elements and the number of elements required, N .

⟨Process variables, pass 4 50b⟩ ≡

```
struct ready_struct
{
    int ready;
    int sent;
    int timed;
};
```

```
int          i;
int          M;
```

```

int          N;
int          k = 0;
int          j;
struct ready_struct *readyp;
◇

```

Macro referenced in scrap 56c.

(Initialize process variables, pass 4 51a) \equiv

```

i = atoi( argv[1] );
M = atoi( argv[2] );
N = atoi( argv[3] );

readyp = malloc( N * sizeof( struct ready_struct ) );

for (l = 0; l < N; l++)
{
    readyp[l].ready = M;
    readyp[l].sent = 0;
    readyp[l].timed = 0;
}
◇

```

Macro referenced in scrap 56c.

The first action of the process is

```

readyp[j] = M  $\rightarrow$  k := (k + 1) mod M;
    if k = i  $\rightarrow$  k := (k + 1) mod M
    || k  $\neq$  i  $\rightarrow$  skip
    fi;
    readyp[j] := k;
    send request(j) to p[k]

```

This action sets **k** to a good, next process identifier; sets *readyp*[j].ready (the structure element used to hold the actual state of the slot) to **k**; and sends the request.

The implementation, however, is required to do the bookkeeping required for message resends using the timeout action, action 4.

(P's actions, pass 4 51b) \equiv

```

int action1g( void )
{
    for (j = 0; j < N; j++) if (readyp[j].ready == M) return( TRUE );
    return( FALSE );
}

void action1b( void )
{
    REQUEST_MSG msg;

    k = (k + 1) % M;
    if (k = i)
    {
        k = (k + 1) % M;
    }
    readyp[j].ready = k;

    msg.type = REQUEST;
    msg.len = sizeof( REQUEST_MSG );
    msg.slot = j;
    SEND_MESSAGE( (MSG_BUFF *) &msg, k );

    <Add action 4, pass 4 56a>
    readyp[j].sent++;

    printf( "Sending request %d to %d\n", j, k );
}

```

◇

Macro defined by scraps 51b, 53, 54b, 55b.
 Macro referenced in scrap 56c.

<Add action 1, pass 4 52> ≡

```

    ADD_LOCAL_ACTION( action1g, action1b );

```

◇

Macro referenced in scrap 56c.

The second action of the process is

rcv request(j) from p[l] → send reply(j) to p[l]

The implementation of this action is relatively straightforward. Two differences need examination:

- The variable l from the protocol specification is not present. It is actually not needed; the information it carries is held in the `sender` element of the `RCVD_MSG` structure. Contrast l , which is not needed, with j in this action, which is also not needed but is present because it actively simplifies the programming here. If j was not used here, the information it carries is available from the `slot` element of the `REQUEST_MSG` structure. However, accessing that information requires a cast (and a temporary, simplifying variable) since the function only has access to a pointer to a `MSG_BUFF`. Thus, j is a useful value whereas l is not.
- The guard function does not check who the sending process is. This check, which appears in the previous protocols, is not necessary—it is unlikely that an unknown process will be sending messages to this one. If it were to be here and if this protocol did differentiate between between some processes of the array, it would most likely take the form of a range check in the guard `if` statement.

$\langle P$'s actions, pass 4 53 $\rangle \equiv$

```
int action2g( RCVD_MSG *rcvd )
{
    REQUEST_MSG *req;

    if ( rcvd->msg->type == REQUEST )
    {
        req = (REQUEST_MSG *) rcvd->msg;
        j = req->slot;
        return( TRUE );
    }
    return( FALSE );
}

void action2b( RCVD_MSG *rcvd )
{
    REPLY_MSG    msg;

    msg.type = REPLY;
    msg.len = sizeof( REPLY_MSG );
}
```

```

    msg.slot = j;
    SEND_MESSAGE( (MSG_BUFF *) &msg, rcvd->sender );

    printf( "Sending reply %d to %d\n", j, rcvd->sender );
}

```

◇

Macro defined by scraps 51b, 53, 54b, 55b.
 Macro referenced in scrap 56c.

⟨Add action 2, pass 4 54a⟩ ≡

```

    ADD_RECEIVE_ACTION( action2g, action2b );

```

◇

Macro referenced in scrap 56c.

$p[l]$'s third action is

rcv reply (j) **from** $p[l] \rightarrow ready_p[j] := M$

The implementation is straightforward given an understanding of the points discussed by the previous action.

⟨ P 's actions, pass 4 54b⟩ ≡

```

int action3g( RCVD_MSG *rcvd )
{
    REPLY_MSG *reply;

    if (rcvd->msg->type == REPLY)
    {
        reply = (REPLY_MSG *) rcvd->msg;
        j = reply->slot;
        return( TRUE );
    }
    return( FALSE );
}

void action3b( RCVD_MSG *rcvd )
{
    ready_p[j].ready = M;
}

```

```

    printf( "Receiving reply: %d from %d\n", j, rcvd->sender );
}

```

◇

Macro defined by scraps 51b, 53, 54b, 55b.
 Macro referenced in scrap 56c.

⟨Add action 3, pass 4 55a⟩ ≡

```

    ADD_RECEIVE_ACTION( action3g, action3b );

```

◇

Macro referenced in scrap 56c.

The timeout action for the process is

```

timeout  ready $p[j] \neq M$ 
            $\wedge$  request(  $j$  ) #C.p[ $i$ ].p[ready $p[j]$ ] = 0
            $\wedge$  reply(  $j$  ) #C.p[ready $p[j]$ ].p[ $i$ ] = 0  $\rightarrow$ 
send request(  $j$  ) to p[ready $p[j]$ ]

```

Fortunately, this action has also not changed materially from the previous version.

⟨P's actions, pass 4 55b⟩ ≡

```

int action4g( long d )
{
    ready $p[d]$ .timed++;
    if ( ready $p[d]$ .ready != M && ready $p[d]$ .timed == ready $p[d]$ .sent )
    {
        j = d;
        return( TRUE );
    }
    return( FALSE );
}

```

```

void action4b( long d )
{
    REQUEST_MSG msg;

    msg.type = REQUEST;
}

```

```

    msg.len = sizeof( REQUEST_MSG );
    msg.slot = j;
    SEND_MESSAGE( (MSG_BUFF *) &msg, readyp[j].ready );
    <Add action 4, pass 4 56a>
    readyp[j].sent++;

    printf( "Resending request: %d to %d\n", j, readyp[d].ready );
}
◇

```

Macro defined by scraps 51b, 53, 54b, 55b.
 Macro referenced in scrap 56c.

Nor has the steps needed to use the action.

<Add action 4, pass 4 56a> ≡

```

    ADD_TIMEOUT_ACTION( action4g, action4b, j, 50000 );
◇

```

Macro referenced in scraps 51b, 55b.

<Declare action 4, pass 4 56b> ≡

```

    int action4g( long d );
    void action4b( long d );
◇

```

Macro referenced in scrap 56c.

Finally, the entire file implementing the process (and indeed, the whole protocol) is `p4.c`. Since it does read some required information from the command line, it prints out a “Usage” message and exits when the information is not available.

"p4.c" 56c ≡

```

#include <malloc.h>
#include <stdio.h>
#include <stdlib.h>
#include "APC.h"

<Protocol constants, pass 4 49>
<Message types, pass 4 50a>

```

⟨Process variables, pass 4 50b⟩

⟨Declare action 4, pass 4 56b⟩

⟨P's actions, pass 4 51b, ... ⟩

```
void main( int argc, char *argv[] )
{
    int          1;

    if (argc < 4)
    {
        fprintf( stderr,
                 "Usage: %s <proc id> <proc count> <rqsts>\n", argv[0] );
        exit( 1 );
    }
}
```

⟨Initialize process variables, pass 4 51a⟩

```
INITIALIZE_ENGINE( i );
⟨Add action 1, pass 4 52⟩
⟨Add action 2, pass 4 54a⟩
⟨Add action 3, pass 4 55a⟩
ENGINE();
}
```

◇

Chapter 5

Reference Guide and Conclusion

5.1 The APC engine

This section presents a short guide to using the functions provided by the APC suite. It is divided into sections describing the two functions for initializing and executing the engine, the two functions for adding local and receive actions to the lists maintained by the engine, the function for setting up a timeout action, the function for sending a message, and finally the types and variables exported by the library for use by programs implementing protocols.

5.1.1 Initializing and executing the engine

Generally, the first function from the library to be called will be `initialize_engine`. This function assigns initial values to the variables internal to the library such as the lists of actions.

`proc` is the process identifier of the process invoking the engine. Depending on the form of process identifier configuration, this value may need to be coordinated with the process's configuration information. For more information, check the various forms of configuration, such as `APC-driver`. In any case, the process identifier will need to be unique among the processes making up the protocol. `initialize_engine` handles this configuration, including setting the calling process's identifier and locating the sending and receiving information for the other processes.

Like all of the functions here, `initialize_engine` returns a **false** value if no errors occurred. If an error did occur, it returns a **true** value and sets `prtcl_err` to a text string describing the error.

In order to simplify the use of the functions, C preprocessor macros are defined which test the value returned by the functions and print out the value of

`prtcl_err` on `stderr`. Furthermore, if the cpp identifier `CONTINUE_ON_ERROR` is not defined before the APC header file is included, the macro causes the program to halt with an error. The macros are capitalized versions of the functions which they call. For example, `INITIALIZE_ENGINE` invokes and tests the error condition of `initialize_engine`.

(Prototype `initialize_engine` 60a) ≡

```
int initialize_engine( int proc );
```

◇

Macro referenced in scrap 69.

(Macro definitions 60b) ≡

```
#ifndef CONTINUE_ON_ERROR
#define EXIT
#else
#define EXIT exit( 1 )
#endif

#define INITIALIZE_ENGINE( p ) \
    if ( initialize_engine( p ) ) \
    { \
        fprintf( stderr, "%s\n", prtcl_err ); \
        EXIT; \
    }
```

◇

Macro defined by scraps 60b, 61b, 62b, 63b, 65b, 66b.
Macro referenced in scrap 69.

`engine` executes the calling process's part of the protocol. The suite maintains lists of local, receive, and timeout actions and `engine` enters a loop in which the local actions' guards are called until all have returned **false**, then the process waits for either a message to arrive or the next timeout to happen. When one of those events takes place, the guards of either the receive actions (if a message has arrived) or the timeout actions (if a timeout has expired) are called and then `engine` loops back to try the guards of the local actions. In the cases of all three actions, if the guard function returns **true**, the body function is immediately called.

`engine` takes no arguments; any information required by it should have been supplied by `initialize_engine` above or `add_local_action`,

`add_receive_action`, and `add_timeout_action` below. It returns after an error has occurred or after the variable `prctl_dne` has been set to **true** by the processing of some action.

`engine` returns **false** if there has been no error or **true** if there has, in which case it sets `prctl_err` to a text string describing the problem.

The C preprocessor macro `ENGINE` can be used to simplify the error condition handling for calls to `engine`.

⟨Prototype `engine` 61a⟩ ≡

```
int engine( void );
```

◇

Macro referenced in scrap 69.

⟨Macro definitions 61b⟩ ≡

```
#define ENGINE() if (engine())          \
                {                       \
                fprintf( stderr, "%s\n", prctl_err ); \
                EXIT;                   \
                }
```

◇

Macro defined by scraps 60b, 61b, 62b, 63b, 65b, 66b.
Macro referenced in scrap 69.

5.1.2 Receive and local actions

When the engine discovers that a message has arrived, it calls the functions implementing the guards of the receive actions in order to determine if one of those actions is applicable to the new message. If a guard returns **true**, indicating that the action is applicable, the function implementing the corresponding action body is invoked. If the guard returns **false**, the engine continues with the next guard function until all have been exhausted; if none returns **true**, the message is discarded.

It is intended that the guard functions should not have an effect on the state of the process (except in the limited sense required to handle parameterized actions). The body functions will need to modify variables or send messages, however.

A structure is passed to the functions implementing receive actions, giving them

1. A copy of the message itself, including the default fields of the *type and* length of the message as well as any fields sent along with them, and
2. The process identifier of the process that sent the message.

The structure is defined by the `RCVD_MSG` type.

In order to initialize the list of receive actions, each pair of functions implementing a receive action should be passed to the engine by the function `add_receive_action`. The guard function takes a pointer to a `RCVD_MSG` as an argument and returns either `true` or `false`, and is the first argument to `add_receive_action`. The body function gets the same argument but does not return a value; it is the second argument to `add_receive_action`. The memory pointed to by `rcvd` is managed by the engine and should not be modified, allocated, or deallocated by either of the two user-specified functions. Once added to the list of receive actions by `add_receive_action`, a pair of functions remains in use until the process is terminated.

If `add_receive_action` detects an error while adding the pair of functions to the list, it will return `true`; otherwise it will return `false`. If there is an error, `prctl_err` will be set to a string describing the problem. `ADD_RECEIVE_ACTION` is the macro simplifying error handling for `add_receive_action`.

(Prototype `add_receive_action` 62a) ≡

```
int add_receive_action( int (*guard)( RCVD_MSG *rcvd ),
                      void (*body)( RCVD_MSG *rcvd ) );
```

◇

Macro referenced in scrap 69.

(Macro definitions 62b) ≡

```
#define ADD_RECEIVE_ACTION( g, b ) if (add_receive_action( g, b )) \
    { \
    fprintf( stderr, "%s\n", prctl_err ); \
    EXIT; \
    }
```

◇

Macro defined by scraps 60b, 61b, 62b, 63b, 65b, 66b.
 Macro referenced in scrap 69.

Local actions are very similar to receive actions; the only major difference between the functions implementing the two is that the local actions' functions do not have any parameters. They are assumed to be able to read the state from the variables of the process directly.

Most of the ideas required for the implementation of receive actions remain the same for local actions. The guard function should return either **true** or **false**, the body should not return any value, and they should be introduced in that order to the engine by the function `add_local_action`. Additionally, once added to the list of local actions maintained by the engine, the guard and body functions of a local action will continue to be used until the process is terminated.

`add_local_action` returns **true** and sets `prctl_err` in the case of an error. Otherwise it returns **false**.

(Prototype `add_local_action` 63a) \equiv

```
int add_local_action( int (*guard)( void ),
                    void (*body)( void ) );
```

◇

Macro referenced in scrap 69.

(Macro definitions 63b) \equiv

```
#define ADD_LOCAL_ACTION( g, b ) if (add_local_action( g, b )) \
    { \
        fprintf( stderr, "%s\n", prctl_err ); \
        EXIT; \
    }
```

◇

Macro defined by scraps 60b, 61b, 62b, 63b, 65b, 66b.
Macro referenced in scrap 69.

5.1.3 Timeout actions

Timeout actions are more difficult to implement. In addition to the guard and body functions which are otherwise similar to receive and local actions, timeouts require more information in order to perform their role. In particular, since the global information used by an Abstract Protocol timeout action is not available to the implementation, alternate information must be substituted. In this case, a delay replaces any non-local information needed by a timeout action.

The delay, `delay`, is passed along with the guard and body functions to the engine by `add_timeout_action`. `delay` is specified in microseconds from the time `add_timeout_action` is called. However, due to the nature of Unix process scheduling, it can only represent a minimum delay; the guard (and possibly the body) functions will be invoked sometime after the delay has expired.

An additional argument is required to `add_timeout_action`, the `data`. This is an integer value which is passed without modification or examination to the guard and body functions as their only argument when they are invoked. This `data` can be used to provide information to the guard and body functions such as to specifically identify the particular timeout among all similar expiring timeouts.

For example, imagine a timeout used to resend a message after it is presumed lost. Because there is no way in the APC suite to remove a timeout action after it has been added to the list maintained by the engine, a timeout will expire for each message sent; yet most of these expiring timeouts are unnecessary because an acknowledgement of the message sent will be received before the delay has expired. Therefore, it is necessary to keep track of the messages sent versus the timeouts which have expired, in order to ignore those expiring timeouts which need to be ignored while responding to those which need further action. The `data` can be used to inform the function implementing the guard of a timeout action of the number of messages sent at the time the timeout is entered into the system; comparing this count with the number of timeouts which have expired at the time that the guard is invoked will separate the bogus timeouts from the needed ones—if the number of timeouts expired is less than the number of messages sent, then the currently expiring timeout delay does not refer to the currently outstanding message; if they are equal, then the current timeout does refer to the current message which therefore needs to be resent.¹

Alternatively, the `data` could be used as an index into some larger structure which would provide more information to the functions implementing the timeout action.

Another difference between timeout and receive/local actions is that the former are single-shot events; the `guard`, `body`, and `data` are added to the engine associated with a specific `delay` and when the delay expires, regardless of whether the guard returns `false` or if any action is taken by the body, the `guard`, `body`,

¹This idea does have certain problems, and it is related to the situation in which the number of expired timeouts is greater than the number of messages sent. In time, the two counts will wrap around since each is a finitely-represented value. If the delay is long enough and enough messages are sent and received the count could wrap around so that a current-message count laps the current-timeout count and provides falsely equal values. In practice, this is not expected to be a problem since there are a great many possible values in a long and even the longest timeout is short compared to the time required to send that many messages on current networks.

and **data** are removed from the system. Further calls to `add_timeout_action` are required each time a message is sent, for example.

In all other ways, `add_timeout_action` behaves just as `add_receive_action` and `add_local_action` do.

(Prototype `add_timeout_action` 65a) ≡

```
int add_timeout_action( int (*guard)( long ),
                      void (*body)( long ),
                      long data,
                      long delay );
```

◇

Macro referenced in scrap 69.

(Macro definitions 65b) ≡

```
#define ADD_TIMEOUT_ACTION( g, b, da, de ) \
    if (add_timeout_action( g, b, da, de )) \
    { \
        fprintf( stderr, "%s\n", prtcl_err ); \
        EXIT; \
    }
```

◇

Macro defined by scraps 60b, 61b, 62b, 63b, 65b, 66b.
Macro referenced in scrap 69.

5.1.4 Sending messages

When a process wishes to send a message to another process, `send_message` is invoked. The first argument, `buffer`, is a pointer to a buffer containing the message to be sent. The second argument, `receiver`, is the process identifier of the process which is to receive the message. A process can send a message to itself or any other process in the protocol whose identifier is known to the engine.

The message buffer should contain at least two fields: a type, `type`, and a length, `len`. The `type` is a protocol-specific value which is uninterpreted by the engine but is designed for user code to determine the kind of message being sent or received. On the other hand, the `len` is used by the engine to determine the number of bytes in the message; the size of the data which needs to be transmitted. Additional fields needed in the message can be tacked on after these two.

Differing machines can have differing representations for the fields in the data message, for example the conflict between little-endian and big-endian representations of integers. In order to prevent this conflict from causing problems in the execution of a protocol, the fields should be transformed to canonical representations before being sent and back to machine representations after being received. The standard socket programming systems provide four simple functions to handle these steps: `htonl`, `htons`, `ntohl`, and `ntohs`. The first four letters specify the action, *net to host* or *host to net*. The last letter indicates the size of the number being transformed, *long* or *short*. Using the functions is simple:

Sending Process	Receiving Process
<code>msg.field = htonl(4);</code>	<code>var = ntohl(msg.field);</code>

This code, assuming the `msg` was transferred in the mean time, puts 4 into `var`. In order to simplify the protocol writer's responsibilities, the `type` and `len` fields are transformed to and from the network representations by the engine.

In the case of a problem, `send_message` returns `true` and sets `prtcl_err`; otherwise it returns `false`.

<Prototype `send_message` 66a> ≡

```
int send_message( MSG_BUFF *buffer,
                 int receiver );
```

◇

Macro referenced in scrap 69.

<Macro definitions 66b> ≡

```
#define SEND_MESSAGE( b, r ) if (send_message( b, r )) \
                             { \
                             fprintf( stderr, "%s\n", prtcl_err ); \
                             EXIT; \
                             }
```

◇

Macro defined by scraps 60b, 61b, 62b, 63b, 65b, 66b.
Macro referenced in scrap 69.

5.1.5 Types and variables

A message is handled in a protocol implementation as an instance of the `MSG_BUFF` data type. It is the basic building block of the messages passed between the processes in a protocol. Simple messages can be created using the structure as it is, by assigning the `type` field a long integer indicating a particular message type and the `len` field the value `sizeof(MSG_BUFF)`.

More complex messages require additional fields, such as sequence numbers, data, and so forth. These more complex messages can be constructed by creating additional data types whose first two elements are the `type` and `len`. For example, a connection request with a sequence number might be:

```
typedef struct
{
    long type;
    long len;
    long seq;
} CRQST_TYPE;
#define CRQST 4
...
CRQST_TYPE crqst;
...
crqst.type = CRQST;
crqst.len = sizeof( CRQST_TYPE );
crqst.seq = i++;
...
SEND_MESSAGE( (MSG_BUFF *) &crqst, Q );
```

⟨Exported types 67⟩ ≡

```
typedef struct msg_buff_struct
{
    long      type;
    long      len;
}           MSG_BUFF;
◇
```

Macro defined by scraps 67, 68a.
Macro referenced in scrap 69.

When a message is received, it and the process identifier of the sender is supplied to the guards of the receive actions, and potentially to the bodies. This is

done by passing a pointer to a `RCVD_MSG` to the functions. The first field is a pointer to the received message structure and the second is the sender's process identifier.

(Exported types 68a) \equiv

```
typedef struct rcvd_msg_struct
{
    MSG_BUFF      *msg;
    int           sender;
}                RCVD_MSG;
◇
```

Macro defined by scraps 67, 68a.
Macro referenced in scrap 69.

The engine exports three variables. The first is `proc_count`, which is the number of processes involved in the protocol. Acceptable process identifiers range from `0 ... proc_count - 1`.

(Exported variables 68b) \equiv

```
extern int      proc_count;
◇
```

Macro defined by scraps 68bc.
Macro referenced in scrap 69.

The other two variables are `prtcl_err`, which is used to return a zero-terminated text string describing an error condition, and `prtcl_dne`, which halts the `engine` function when set to `true`.

(Exported variables 68c) \equiv

```
extern char     *prtcl_err;
extern int      prtcl_dne;
◇
```

Macro defined by scraps 68bc.
Macro referenced in scrap 69.

5.1.6 The APC engine header file

The file `APC.h` should be `#included` by any implementation file using the APC suite to create a protocol's process.

If they are not already defined, `APC.h` defines `TRUE` and `FALSE` for the use of the implementation. It also `#includes` `netinet/in.h`, which has the definitions of `htonl` and related functions. `netinet/in.h` requires `sys/types.h`

```
"APC.h" 69 ≡
```

```
#include <sys/types.h>
#include <netinet/in.h>

<Exported types 67, ... >
<Exported variables 68b, ... >
<Prototype initialize_engine 60a>
<Prototype engine 61a>
<Prototype add_receive_action 62a>
<Prototype add_local_action 63a>
<Prototype add_timeout_action 65a>
<Prototype send_message 66a>
<Macro definitions 60b, ... >

#ifndef TRUE
#define TRUE 1
#define FALSE 0
#endif
◇
```

5.2 Compiling and executing APC protocols

While the rest of this work deals mainly with writing the programs which implement protocols, this section discusses compiling and executing those programs. The APC suite is designed to be flexible and could be embedded within a larger framework to produce a more useful application; but, since that has not been a focus of this work up until this point, this section will not deal with the issues raised by such an effort. Specifically, this section will assume that the task at hand is to produce a “toy” implementation of a protocol, perhaps to get a feel for how the protocol behaves or to compare differing protocols.

Therefore, assume that you have several source files implementing the various processes of the protocol; i.e., `p.c` and `q.c`. These source files contain the definitions of the functions implementing actions as well as a `main` function and the variables required for each process. In actuality, you may need to break these up into separate files, but linking them together only modestly complicates the procedures that follow.

5.2.1 Compiling the programs

A modest feature of the APC suite is that compiling a program which uses functions from the APC suite is no more difficult than compiling any program written in C. Assuming that the APC library is in a directory *libdir*, the APC header file is in the directory *includedir*, the source program is `p.c`, and that the executable will be `p`, all that is required to compile the program is the command,

```
cc -Iincludedir -Llibdir -o p p.c -lAPC
```

if the C compiler is known as “`cc`” and no other libraries are needed. Generally, the `-Idirectory` argument to a C compiler tells the compiler to look into *directory* to find header files in addition to those in the standard places, such as `/usr/include`. Similarly, `-Ldirectory` tells the compiler to look for libraries in *directory*. Finally, `-llibrary` tells the compiler to look for a library called `liblibrary.a`. `libAPC.a` is the name of the library which contains the APC suite functions. Additional libraries, directories, or other arguments may be necessary, depending on the protocol and implementation.

Assuming the compilation succeeds, the resulting `p` will be an executable program for the machine on which the compilation was done (and similar machines, of course, and assuming the compiler was not configured for cross-compilation). However, due to the fact that most interesting protocols involve several executing processes, running the protocol requires additional steps.

5.2.2 Executing the protocol

There are two possible difficulties in starting a protocol involving several processes.

1. It may be necessary to invoke the programs on several different machines.
2. It is necessary to provide each of the running processes with the “addresses” of all of the other processes. In TCP/IP terms, an address is a (*host name*, *port number*) pair. The *host name* is the name of the machine on which a given process is running; the *port number* is a serial number, unique on that host, to which messages to a process can be addressed.

For current purposes, the APC suite solves both of those problems with a driver program. The driver program reads a configuration file consisting of a host name, a process identifier, and a command line for each process. The command line specifies the program implementing the process along with any arguments that it may need. The program on the command line should be either in the execution path for the user on the remote machine, or should be specified by a path, either fully qualified from the root directory on the remote machine or relative to the home directory of the user. The host name identifies the host on which the process is to run and the identifier is the process identifier to be used by the program; it needs to match the identifier specified by the call to `initialize_engine` by the program. The format of the file is just that, a sequence of lines each containing the triple *host name, process id, command line*. Blank lines are ignored and anything that follows the character “#” on a line is assumed to be a comment and is also ignored.

An example configuration file would be:

```
bovina 0 p # The executable ~/p on bovina should be process 0

jeckle.cs.utexas.edu 1 testing/bin/q foo # q needs an argument, foo
```

By default, the configuration file should be called `APC-configuration`.

To start the driver program and begin the protocol, invoke

```
$ APC-driver
```

Because the driver program is a short script which may not satisfy all requirements, it is described in more detail in the next section.

As an example, the configuration file for the multi-process request/reply protocol is

```
"APC-configuration" 71 ≡
```

```
homsona 0 ./p 0 5 6 > p0.out 2>&1
homsona 1 ./p 1 5 6 > p1.out 2>&1
homsona 2 ./p 2 5 6 > p2.out 2>&1
homsona 3 ./p 3 5 6 > p3.out 2>&1
homsona 4 ./p 4 5 6 > p4.out 2>&1
◇
```

This configuration starts all of the processes on a machine `homsona`, with 5 processes allowed 6 outstanding messages. All of the output and errors from the programs are

logged in files called `pn.out`. Notice that the second and fourth columns match—the second tells the configuration system which process id to use for which process, and the fourth tells each running process which process id to assume.

5.2.3 APC driver internals

In order to understand the peculiarities of the APC driver program, more discussion is needed about how it works. The driver is a Perl script which calls many other programs to actually do the work. The three most important are:

- **rsh**—The TCP/IP remote program execution command. In order to start `p` on the remote machine, the driver uses `rsh` to start a program, `APC-remote`, executing on that machine. For more information on the use of `rsh`, see its documentation.
- **APC-remote**—The first part of the APC suite configuration system. `APC-remote` takes as arguments the address of the `APC-server` (described below), the process identifier, and the command line to be executed. `APC-remote` contacts the server, informs it of the address that the process will be using, gets from it the addresses of all of the processes, and executes the command line, *becoming* the protocol process. The configuration information, including all the required addresses for the other processes is passed to `initialize_engine` through environment variables by `APC-remote`.
- **APC-server**—The protocol configuration hub. `APC-server` provides the driver script with the port number which it will be using (the driver can find the host name on its own), and waits for the appropriate number of `APC-remote`'s to contact it. Then it distributes the protocol configuration and exits. Any errors that it discovers after giving the driver script its port number will be described in a text file `APC-server.err` that it creates before it dies.

If the protocol does not have well-defined termination conditions, in particular if all of the programs implementing the protocol do not set `prtcl_dne` and exit (this is the case with several of the example protocols presented here), it will be necessary to terminate the running protocol with whatever normal interrupt procedures are available. Usually, this will mean typing Cntrl-C one or more times.

5.3 Conclusion

5.3.1 Design and implementation

This paper deals covertly with many implementation issues. This section will look at some of those issues, but not to the depth of the complete implementation.

The basic design issue was whether to try to implement a compiler or to build a toolkit to which a compiler front end could be added later if deemed necessary. This work focused on a toolkit approach for two reasons:

- It would get to the interesting problems faster and easier. Building a compiler requires describing the grammar of AP in a formal way, working to implement things such as standard arrays and statements, and a variety of other things which are not directly related to the task of implementing a networking protocol.
- It was philosophically more interesting. This was a problem of AP being both too close and too far away from a standard programming language. Most of the work of a compiler, such as **if** statements, **do** loops, and so forth are implemented perfectly well in the compiler for any procedural language. Therefore, writing a compiler to transform an **if** statement to an **if** statement seemed relatively pointless. On the other hand, some features of AP, particularly **timeout** actions, do not easily lend themselves to machine translation. (The handling of **timeout** actions is described in chapter 4.) The guards of these actions in particular require access to global state information which is not available in a distributed system. It is therefore necessary to transform the global predicate into a predicate using only local information and the only kind of global information available to a process, the current time. This sort of transformation is difficult or impossible for a compiler; it requires human intervention anyway.

For both of the above reasons the current design was chosen. It is small, to the point, and it avoids difficult and possibly error-inducing machine transformations.

Most of the secondary design issues, such as which language to use, were arbitrary choices—no issues forced an particular choice.

The key to the design is the recognition that AP protocols are reactive. Each action examines the current state and potentially transforms it into another state. This recognition made it possible to create a generic model of local and receive actions, and to use that model to create an engine for executing those actions. This model was then extended to allow timeout actions.

5.3.2 Future work

Currently, the APC suite serves well as a testbed for protocol designs. Several avenues of future work are available:

- The use of pseudo-inheritance to implement different message types with different sizes and kinds of fields is particularly unpleasant. It should be possible to rid the suite of this particular blight by reimplementing the system in an object-oriented language such as C++. A smart message class should have a method of converting its own representation into a form suitable for transmission, and a way of recreating itself upon receipt from another machine. Such a class would also handle the processing required to convert message elements into network-oriented canonical formats.
- The suite as currently implemented is small and apparently flexible, and the programs created with it are also fairly small and quick. It would be interesting to compare the performance of this method of implementing a networking protocol with a traditionally implemented network, both in the realm of messages sent per time unit and load on the machines involved. If the protocol implemented with this suite is not terribly worse than the traditional protocol implementation, it might be worthwhile to look at improving the performance of protocols in general with this style of implementation and at improving the performance of this suite.
- The timeouts remain a thorny issue. While it is difficult to imagine a way of getting more information for their guards or of a different kind of information which could better be used to implement them, it seems that it should be possible to improve the suite's use of the timeout idea or, at least, the interfaces that the suite uses to access that idea.
- This suite has, until this point, been used only to implement “toy” protocols. It might be interesting to examine what kind of issues are raised by the use of this suite in a production protocol, such as tftp. Since actions invoked by the engine can easily call other functions, in order to get data, for example, it should be feasible to embed the engine in a larger framework to put together a robust application. However, significant changes to the APC driver would be required.

Bibliography

- [Barnes 82] Barnes, J.G.P. *Programming in ADA*. Wokingham, England: Addison-Wesley, 1982.
- [Brown 91] Brown, Geoffrey M., Mohamed G. Gouda, and Raymond E. Miller. “Block Acknowledgment: Redesigning the Window Protocol.” *IEEE Transactions on Communications*, Vol. 39, No. 4 (April 1991).
- [Burns 93] Burns, James E., Mohamed G. Gouda, and Raymond E. Miller. “Stabilization and pseudo-stabilization.” *Distributed Computing*, 7 (1993).
- [Gouda 91] Gouda, Mohamed G., and Nicholas J. Multari. “Stabilizing Communication Protocols.” *IEEE Transactions on Computers*, Vol. 40, No 4 (April 1991).
- [Gouda 93] Gouda, Mohamed G. “Protocol verification made simple: a tutorial.” *Computer Networks and ISDN Systems*, 25 (1993).
- [Knuth 86] Knuth, Donald E. *T_EX: The Program*. Reading, Massachusetts: Addison-Wesley, 1986.
- [Knuth 92] Knuth, Donald E. $\langle\langle$ *Literate Programming* $\rangle\rangle$. Center for the Study of Language and Information Lecture Notes Number 27 (1992).
- [Knuth 93] Knuth, Donald E. *The Stanford GraphBase*. New York: ACM Press, 1993.
- [Pleier 93] Pleier, Christoph. “The Distributed C Development Environment.” Institut für Informatik, Technische Universität München, 1993.
- [Sewell 89] Sewell, Wayne. *Weaving a Program: Literate Programming in WEB*. New York: Van Nostrand Reinhold, 1989.
- [Stevens 90] Stevens, W. Richard. *UNIX Network Programming*. Englewood Cliffs, New Jersey: PTR Prentice Hall, 1990.