Experience with RAPID Prototypes

Danny Dolev*

Institute of CS
Hebrew University
Jerusalem, Israel
dolev@cs.huji.ac.il

Ray Strong

IBM Research Division
Almaden Research Center
San Jose, CA 95120-6099
strong@almaden.ibm.com

Ed Wimmers

IBM Research Division
Almaden Research Center
San Jose, CA 95120-6099
wimmers@almaden.ibm.com

Abstract

The goals of the RAPID environment are (1) to make the programming of distributed protocols simple without restricting the protocol relevant choices of the programmer, (2) to provide encapsulation and reusability that are at least as powerful as those offered by object oriented programming, and (3) to provide for different styles of programming that make RAPID an easy transitional programming environment between older and lower level languages and C.

The environment provides and is programmed in the RAPID-FL subset of the functional language FL. Although the full power of FL is available to the programmer, a very small number of concepts need to be learned to program in RAPID-FL. Moreover, restriction to RAPID-FL means that one can have the safety of a functional language combined with reasonable uses of assignment.

RAPID makes storage management trivial and reduces the complexity of communication management to handling a few simple commands. The programmer can arrange for true broadcast or point-to-point communication via either UDP or TCP, without having to master the details of these communication protocols.

A protocol written in RAPID-FL is compiled to C. It can run anywhere C can run and interoperate with other programs written either in C or in RAPID-FL. So the programmer can build a RAPID prototype and replace pieces of it with C to provide a test bed for the development of a C program or for better performance.

In this paper we describe our experience using RAPID to perform clock synchronization experiments and to serve as scaffolding for high performance C code that implements a collective communication protocol for parallel machines.

*This research was supported in part by Yeshaya Horowitz Association.

1 Introduction

RAPID is a programming environment. It is designed with highest priority given to simplicity and ease of use. Its programming language is a small, easy to learn subset of the general purpose functional programming language FL [FL 89]. This subset is called RAPID–FL. While RAPID is not a general purpose programming language, it has been designed to facilitate the rapid prototyping of distributed protocols. A *protocol* consists of an initial state together with a state transition and with an input/output function that defines how the protocol responds to various input events with changes to its state and the production of various outputs. For a *distributed* protocol, the events of primary (but not exclusive) focus are the receipt and generation of messages and the passage of time. We further distinguish messages generated at other sites from those originating either at an operator console or from the local environment (which is provided by the operating system together with functions defined in the set of libraries comprising the RAPID environment). In RAPID–FL a programmer can specify a distributed protocol by specifying (defining) the initial state, and handling functions for each of the distinguished types of input events.

On the occasion of each input event to a RAPID protocol, the RAPID driver (an element of the RAPID local environment) provides, as input to the protocol defining function, the protocol state and the specific

1074-6005/94 $3.00 © 1994 IEEE

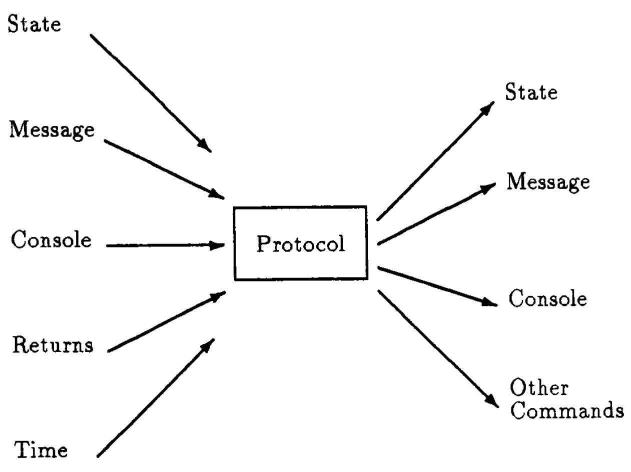

Figure 1: A RAPID protocol

set of messages or other input events that have occurred. The protocol defining function provides, as output, a set of commands to the driver, including commands to alter the current state. The cycle of information flow between the driver and the protocol defining function is the execution of the protocol. The driver is programmed in FL and in C. All FL programs are compiled to C. The driver part of the RAPID environment captures the platform and operating system dependencies of a RAPID protocol.

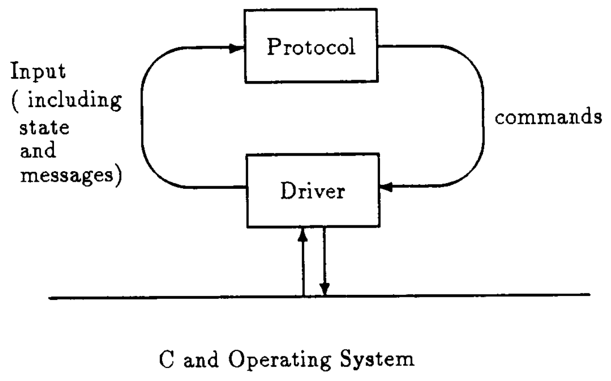

Figure 2: Relationship between protocol and driver

The remainder of this paper is organized into sections covering the style of programming in RAPID-FL, an example clock synchronization program written in RAPID-FL, a summary of RAPID features, a discussion of interoperation between C and RAPID, a discussion of our experience running clock synchronization experiments in RAPID, and a discussion of our experience using RAPID as scaffolding for the development of high performance C code. The complete RAPID-FL syntax and set of commands is presented in [DSW 93] along with a discussion of its relationship to the general purpose functional language FL.

2 RAPID-FL Style

The RAPID environment includes many useful features that are automatically provided to facilitate programming. In addition, templates are included for the functions and definitions that are protocol specific and must be provided by the RAPID-FL programmer. These are summarized in the following table.

RAPID features
Automatic storage management
Simple network communication
Easy to learn
Flexible structure of programs
Interoperates with C
Snapshot debugging
Templates Completed by User
Input Specification
Initial Protocol State
Statistics Specification
Message Handler
Console Handler
Driver Return Handler
Timeout Handler

The Input Specification serves as a repository for declarations of RAPID-FL identifiers. In RAPID-FL the scope of these defined identifiers is global and any temporary variables that will be used anywhere in the protocol must be declared here with an accompanying *type*. However, in addition to simple type declarations like *isint* (integer), *isreal* (real), and *isseq* (sequence), RAPID-FL allows a declaration of *isval* that allows identifier so declared to refer to any FL value. Thus a single temporary variable will suffice for multiple uses. An example temporary variable A is provided with the template. The part of the template to be filled in by the programmer is the part that specifies the protocol state. The other parts of the protocol input are already supplied.

In addition to specifying the identifiers that refer to parts of the protocol state, the programmer must build an initial state that satisfies these specifying declarations out of constants. A RAPID–FL character string or numerical *constant* is preceded by the symbol ~, as in ~"abc"or ~ 0.

The Statistics Specification can be used to collect measurements, statistics, or debugging information.

The heart of the protocol consists of the four handler templates for remote messages, messages from the local console, general messages from the local driver, and timeouts. Each handler is designed to be programmed in the style represented by the following schema:

$$
\begin{aligned}
&def\ Handler ==\\
&\qquad Event_1 \rightarrow Response_1;\\
&\qquad Event_2 \rightarrow Response_2;\\
&\qquad Event_3 \rightarrow Response_3;\\
&\qquad \vdots\\
&\qquad Event_n \rightarrow Response_n;\\
&\qquad Else : Response
\end{aligned}
$$

The $Event_i$ parts of the schema are predicates specifying input events. $Event_1$ is checked first. The right arrow represents *if ... then.* while the semicolon represents *else.*

3 Example Clock Synchronization Program

The following example is typical of the simple prototypes with which we have recently experimented. It is based on a clock synchronization protocol of Halpern, et. al. ([DHSS 84]). The program is broken into eight small files:

input a set of type declarations and an initial state for the protocol,

message the external message handler,

console the handler for input from the console,

return the handler for returns from driver commands issued by the protocol,

timeout the timeout handler,

stats a declaration for each statistic gathered by the protocol,

protocol RAPID–FL code that controls the use of the handlers (this code is supplied as a template but modifiable by the protocol developer),

driver RAPID–FL code that calls the RAPID library functions to support the protocol (this file is given type .fl and the other files are given type .inc and included in the driver file for compiling).

Generic code from the templates is displayed in normal type, while protocol specific code is displayed in bold type.

All of our clock synchronization prototypes make use of an object called a *logical clock*. A logical clock is a way of maintaining a synchronized time service in software without touching the underlying hardware clock on which it depends. The details of logical clock functioning are beyond the scope of this paper. In this example, we will illustrate our interoperation with C by calling a logical clock implemented in C (see section 4). The methods for operating on a logical clock are represented by the functions **FlTime** (which reads the logical time), **FlSet** (which sets the logical time), and **FlLCInit** (which creates a logical clock). The clock synchronization protocol is invoked every period (P), and is controllable by the prototype user.

Figure 3 presents the first (input) part of the program. The flexibility of RAPID allows the programmer to specify the structure of the state the protocol maintains. In this example, the protocol state consists of three numbers (lines 7-9) but arbitrarily complex states (with named substates) can be expressed. The code beginning with line 1 defines the input to the protocol, which includes the State (beginning on line 4), the current hardware clock time in seconds (line 11), a sequence of external Messages (line 12), a sequence of messages from the Console (line 13), and a sequence Ret of returns from driver commands (line 14). Note that most of the code is supplied from the template. There are only three places where the protocol developer supplies code: (1) the specification of the state from line 7 to line 9, (2) matching initial values from line 19 to line 21, and (3) the initial value for the sequence of driver commands on line 23. Line 25 includes code that links the program to the libraries of the RAPID environment.

```
{ { type Input ==
[|
IS.isval ,
State.[|
      A.isval ,
      CommandSeq.isseq ,
      ET.isreal ,
      P.isreal ,
      L.isint
      |] ,
HardwareTime.isreal ,
Messages.isseq ,
Console.isseq ,
Ret.isseq
|]

def InitProtocolState == [
      [] ,
      InitCommands ,
      ~0.0 ,
      ~300.0 ,
      ~0 ,
      ]

def InitCommands ==
           [ [ ~Open , [ ] ] ] }
where
# include "/afs/alm/cs/rapid/grapid.inc" }
```

Figure 3: INPUT.INC

```
{ export ( HandleMessage ) {

def Link == s1 @ s1 @ Messages
def Message == s2 @ s1 @ Messages

def Tag == s1 @ Message
def T == s2 @ Message

def HandleMessage ==
      Not @ isseq @ Message
            → Discard ;
      Not @ ( (len @ Message) = ~2 )
            → Discard ;
      ( ( Tag = ~"Time") And
            ( T ge ET ) )
        → Execute:<
            A /gets [ ~UpdateStats ,
                [ ~"diff",
                T - (FlTime @ [ ]) ] ] ,
            CommandA ,
            A /gets ( FlSet @ [ T ] ) ,
            ET /gets ( T + P ) ,
            A /gets [ ~Send , [ L , [
                ~"Time", T ] ] ] ,
            CommandA ,
            A /gets [ ~Wakeup ,
                HardwareTime + P ] ,
            CommandA > ;
       Else: Discard

def Discard == id }
where ... }
```

Figure 4: MESSAGE.INC

Figure 4 contains the code that describes how to handle a single external message (first in the sequence Messages). Note the ease with which commands can be interspersed with assignments in lines 13-20. Line 1 says that the only function visible outside this file is HandleMessage, so we don't need to worry about defining functions already defined elsewhere in the program. Operator s1 selects the first element in a sequence; operator s2 selects the second element; etc. The sequence Messages always consists of pairs, the first of which is a link number indicating the origin of the message, and the second of which is the message body. Lines 2 through 5 define some handy abbreviations in terms of these operators. However, lines 4 and 5 assume that the body of the message is a sequence with two elements (a tag Tag and a time T); so lines 7 and 8 are tests for these assumptions. If the message passes the tests of line 9, then the block of code from line 10 through line 18 is executed.

The driver command UpdateStats in line 11 causes the value of T - (FlTime @ [ ]), which is the difference between the time on the message and the current reading of the logical clock, to be added to the statistics being collected by the program under the tag "diff" (see figure 8).

The function CommandA in lines 12, 16, and 18 has the effect of appending the command in A to the sequence CommandSeq of commands to be passed to the driver. CommandA is defined in RAPID–FL by

```
def CommandA == CommandSeq /gets (ar @
               [ CommandSeq , A ]).
```

where the operator ar means "append right."

The operator **/gets** is a simple and safe assignment. Only variables defined in the input file (figure 3) are allowed on its left hand side. The value of the expression on its right hand side is computed and used to replace the current value of the variable on the left. The function Discard is defined in line 20 as the identity function (id) which makes no change to the state of the protocol. It could instead be defined to display the message at the console or count the number of such messages. We have deleted the code from the template that includes the appropriate other files in line 21.

Figure 5 displays the code for handling a single console message from the sequence Console. Console messages are character strings. The program terminates according to line 4 if "quit" is entered at the console. If either "?" or "help" is entered, then the program displays a menu according to line 5 and lines 14 through 19. Note that most of the menu is supplied by the template. Included in the template is the function

```
1.  { export ( HandleConsole ) {
2.  def C == s1 @ Console
3.  def HandleConsole ==
4.      (C = ~"quit") → Stop @ ~"Thanks
             for using RAPID."
5.      ((C = ~"?") Or (C = ~"help")) →
          Execute:< A /gets [ ~Display ,
          MainMenu ] , CommandA > ;
6.      (C = ~"reset") → Execute:< A /gets
            [ ~ResetStats , [ ] ] ,
            CommandA > ;
7.      (C = ~"report") → Report ;
8.      ((( len @ C ) ge ~8 ) And
            ( ~"period "= [s1,s2,s3,
            s4,s5,s6,s7]@ C )) →
9.          P /gets ( /p @ (/d:~"New
            Period") @ string2real
            @ tln @ [ ~7 , C ] ) ;
10.     Else:id
11. def Report == Execute:<
12.     A /gets [ ~ReportStats , [ ] ] ,
            CommandA ,
13.     A /gets [ ~Record, State] ,
            CommandA >
14. def MainMenu == [
15.     ~"Welcome to RAPID" ,
16.     ~"At any time enter one of the
            following commands:" , ~"" ,
17.     ~"help or ? for this message" ,
18.     ~"quit to terminate protocol" ,
19.     ~"period #        # must be real
        to set period in sec", ~""]}
20. where ... }
```

Figure 5: CONSOLE.INC

```
1.  { export ( HandleRet ) {
2   def TagRet == s1 @ s1 @ Ret
3.  def ArgRet == s1 @ s2 @ s1 @ Ret
4.  def ValRet == s2 @ s2 @ s1 @ Ret

5.  def HandleRet ==
6.        (TagRet = ~"Wakeup") →
                    HandleTimeout ;
7.        (TagRet = ~"Open") →
                    Execute:<
8.             L /gets ValRet ,
9.             A /gets FlLCInit @ [ ~0, ...
                    ] > ;
10.       Else:id }
11. where ... }
```

Figure 6: RETURN.INC

Report, defined from line 11 to line 13. This function arranges to display the statistics that have been gathered by the program (see figure 8) and to display the current state of the protocol (State).

The program allows a user to enter a new period (P) from the console. The value for the new period in seconds is converted from a character string to a real by the FL library function string2real in line 9. The /d operator has the effect of displaying its immediate argument at the console while it passes the new value of P to the /p operator. The /p operator displays its argument (of any type) at the console and passes it on. We refer to /p as the snapshot operator because it allows the programmer to take a snapshot of any data structure without interrupting the computation, and also without having to compute, declare, or specify the type of that data structure. Snapshots are very useful in debugging, as well as in communicating with the console operator.

Figure 6 displays code for processing driver returns. Notice the event-response style of coding allows the protocol to be expressed at a high level in lines 6-10. In this program the only return of interest (besides timeout returns from driver command Wakeup) is the return from the intial command Open (see figure 3). The integer returned from Open is put into L according to line 8 of figure 6 and the logical clock is initialized by the call to function FlLCInit in line 9 there. We have suppressed the details of the parameters passed to FlLCInit since they are outside the scope of the paper. This code illustrates a typical programming practice for RAPID prototypes: performing various initializing operations in response to a return from an initial driver command.

```
1.  { export ( HandleTimeout ) {
2.  def ArgRet == s1 @ s2 @ s1 @ Ret
3.  def ValRet == s2 @ s2 @ s1 @ Ret
4.  def T == ET - (FlTime @ [ ])

5.  def HandleTimeout ==
6.        (T le ~0) → Execute:<
7.             A /gets [ ~Send , [ L , [
                    ~"Time", ET ] ] ] ,
8.             CommandA ,
9.             A /gets [ ~Wakeup ,
                    HardwareTime + P + T ],

10.            CommandA > ;
11.       Else: Execute:<
12.            A /gets [ ~Wakeup ,
                    HardwareTime + T ] ,
13.            CommandA > }
14. where ... }
```

Figure 7: TIMEOUT.INC

Figure 7 presents the code for handling timeouts. Since a timeout is a special case of a return, this code is called from the code in figure 6 when the tag on the return is "Wakeup."

Figure 8 presents the code that defines the data structure used for collecting statistics. When the driver command UpdateStats is called in figure 4, the value has tag "diff" so it is added to the appropriate bucket of statistics: one bucket takes values greater

```
{ export ( NStats ) {
def NStats == [ [ ~"diff", [
                    ~0.0 , ~0.05 ] ] } }
```

Figure 8: STATS.INC

```
{ export ( Protocol ) {
def Protocol ← isInput ==
          ExtractCommands @ Execute:<
     DoWhile:(Not @ isnull @ Messages):<
          HandleMessage ,
          Messages /gets (tl @ Messages) >,

     DoWhile:(Not @ isnull @ Console):<
          HandleConsole ,
          Console /gets (tl @ Console) >,
     DoWhile:(Not @ isnull @ Ret):<
          HandleRet ,
          Ret /gets (tl @ Ret) > > }
where ... }
```

Figure 9: PROTOCOL.INC

than or equal to 0.0 but less than 0.05, the other bucket takes values greater than or equal to 0.05 (50 milliseconds). These statistics are the adjustments made to the logical clock by the program in response to external messages. When the command "report" is issued at the console (see figure 5), the count, mean, standard deviation, three largest, and three smallest values from each bucket are displayed in response. When "reset" is issued at the console, all the buckets are emptied. The protocol developer may define different numbers of buckets for an arbitrary number of tags.

Figure 9 consists of three loops that exhaust the input to the protocol of external messages, console messages, and returns from the driver. The function ExtractCommands takes the commands that have been placed in CommandSeq by the operation of the protocol and passes them to the driver as the output of the function Protocol. Note that figure 9 and figure 10 have no protocol specific code, though they are written in RAPID–FL and available for modification.

```
Driver where {
def Driver == DoForever @
       (/d: ~"Welcome...Enter ? for menu.")
            @ InitExtState @
  [ unInput @ InitInput , ExtractCommands
         @ InitInput , NStats , InitIOStatus ]


def DoForever ← ext.isExtState ==
          DoForever @ Execute:<
     DoCommands ,
     WaitForNext ,
     Commands /gets (Protocol @ mkInput
          @ Inner) , > }
where ...
```

Figure 10: DRIVER.FL

4 Interoperation with C

Interoperation between FL and C is greatly facilitated by the fact that FL is translated into C. FL operates by maintaining a large C data structure that is used to implement a heap. When programming strictly in FL, this heap is entirely hidden from the programmer and garbage collection is performed when needed. Furthermore, the internal C data structures that FL uses to represent its entities are also hidden from the FL programmer. However, when calling C from FL, the programmer needs to convert FL representations of entities into normal C representations. In addition, the programmer needs to be concerned about garbage collection issues since if new data is added to the heap, a garbage collection might be initated.

Usually it is the case that the C programmer is not interested in manipulating FL data structures and simply wants to write a C program with the FL data converted into usual C format. Fortunately, RAPID provides an easy and uniform mechanism for converting FL data to C format and vice versa. Essentially the C programmer can use a C macro that defines a C function that is callable directly from FL. This new function automatically converts the FL data structures into standard C format and then calls the C function specified in the macro call.

For example, suppose the programmer wished to call the trigonometric C function **sin**. The programmer would write the following piece of code:

```
CALL_C1(FlSine,real,sin,real)
```

This invocation of the CALL_C1 macro defines a new C function FlSine. The function FlSine can be called directly from FL with an argument of type real (FL format). FlSine converts this argument which is a real in FL format into a real (i.e. a double) in C format. Next FlSine calls the C library function **sin**. Finally, FlSine converts the answer of type double returned by **sin** into the FL format for reals and returns this value to FL. FlSine also manages garbage collection issues so the C programmer need not worry about them.

More formally, the CALL_Cn macro has the following format:

```
CALL_Cn(FlName,AnsType,CFname,T1,...,Tn)
```

where FlName is the name of the newly defined C function that is directly callable from FL, CFname is the name of the C function that is to be called, AnsType is the type returned by the function CFname, and T1,...,Tn are the types of the argument to CFname. The function FlName must be applied (in FL) to a sequence of length n whose FL types correspond to the C types T1,...,Tn. The function FlName converts each of the n arguments from FL format to C format and then calls the C function CFname with these arguments. Finally, the function FlName coverts the answer returned by CFname (which is of type AnsType) into FL format and returns this value.

5 Experience with Clock Synchronization Prototypes

We have recently completed a number of clock synchronization experiments in the RAPID environment. Since it only takes a few days to write a simple clock synchronization program like that of section 3, we were easily able to try many variants of protocols in the literature. The goal of this work was to compare the performance of various protocols in different environments, especially via broadcast communication media. However, much as the high level nature of RAPID–FL contributes to speed in coding and debugging, it also contributes to a very poor performance. Thus a RAPID prototype cannot hope to predict the actual performance attainable by a protocol. Instead it can try to provide relative performance measurements, comparing programs written in the same style of RAPID–FL and operating under the same load. Moreover, the slowdown due to RAPID (as compared, for example, with highly tuned C code) appears to be superlinear in the size and complexity of the program; so we are limited to comparing protocols that can be realized with about the same size and complexity of RAPID–FL code.

To make such comparisons as fair as possible, when the communication network is not dedicated to our experiments, we arranged for the RAPID environment to allow concurrent, non-interfering communication for several distributed systems consisting of sets of processes running RAPID-FL code. This means that we could run several prototypes independently and they would each experience an identical load on the network (because they were all running concurrently). To accomplish this concurrency, each RAPID prototype holds an initial dialogue with the console in which it establishes its host id and an offset to the default port addresses used for communication. Each independent system of processes uses the same offset. Then all processes in a system can communicate without interfering with the communication of processes in any other system, except in that each message in any system contributes to the total network load.

Here we report on a comparison of three well known clock synchronization protocols:

HAL, a slighlty more complex variant of the code presented in section 3 [DHSS 84],

MAR, a variant of the protocol that appears in K. Marzullo's dissertation [M 84] and is the basis for the peer synchronization protocol of the DCE time service [DCE 91], and

PCS, a variant of Probabilistic Clock Synchronization [C 89].

Fixing the network as a single broadcast medium, there are two important performance criteria for clock synchronization protocols:

precision, roughly speaking, the worst case difference in simultaneous clock readings, and

message efficiency, the average number of broadcast messages per second needed to attain a given precision.

In our comparison studies, we concentrated on finding the best precision attainable by each protocol.

The basic protocol presented in section 3 is a *peer* protocol, in the sense that no one clock is taken as the standard. This has certain fault tolerance advantages, although we have not implemented the fault tolerance parts of the protocol as given in [DHSS 84]. However, the protocol presents a problem for comparison purposes because it does not measure, estimate, or predict its own precision. It was easy to remedy this defect by adding a periodic round trip synchronization (the basis of PCS) to the protocol. This adds to the number of messages and the complexity of the code; but makes HAL more directly comparable with MAR and PCS, each of which estimates its own precision as part of the protocol.

MAR is a peer protocol that operates by making round trip synchronizations and using a complex interval intersection method for deriving a time adjustment and new precision estimate from the previous time and precision estimates of all the participants. We simplified the protocol, slightly weakening its fault tolerance properties, by arranging to do the interval intersections one at a time, obtaining a new time and precision estimate after each round trip, rather than waiting for round trip communication with each participant. If there are only two participants, this simplification makes no difference.

PCS is a *master-slave* protocol: one clock is deemed the master and the others synchronize to it. The protocol presented in [C 89] is presented as if it has already been tuned to its best operating characteristics: the distribution of round trip delays is assumed known and the protocol tries repeated round trip synchronizations until it satisfies a given precision constraint (with high probability) or fails after a fixed number of trials. Allowing the protocol to try forever to satisfy a given precision constraint does not work well when the constraint is close to the best possible precision attainable by the protocol. We tried several different modifications and settled on one that makes a small number of attempts to meet its given constraint and then relaxes the constraint by a small amount. RAPID gave us the luxury of experimenting with a number of different versions, rather than guessing which one to try.

Unlike HAL, which does periodic resynchronization, MAR and PCS are driven to resynchronize when their current precision estimates exceed given thresholds. Their precision estimates grow at a rate corresponding to a worst case drift rate between hardware clocks in the network. As a side effect of our work with HAL, we measured the actual drift rate between clocks in our network. The worst case drift rate used for estimates as part of all three protocols is significantly larger than the measured rate.

To summarize the results of our experiments, we found that HAL and MAR performed similarly: attaining best precisions near 30 milliseconds in our environment. With coaxing, PCS was able to attain a better precision by a factor of 2, at the cost of lower message efficiency. MAR can also be a master-slave protocol. Operating in this manner, its best precision was a few milliseconds worse than the best for PCS. It should be emphasized that only the relative performance results are meaningful and that small differences are not significant and may not be reproduced if the protocols are programmed in better performing code. The relative simplicity of PCS may give it a performance advantage over MAR (acting as a master-slave protocol).

Detailed comparison results are beyond the scope of this paper. Here we stress the flexibility and speed with which we can program and modify clock synchronization protocols in RAPID–FL in order to perform meaningful experiments. We used the results of these experiments to design and test new protocols and to discover weaknesses of the existing protocols. This work is ongoing, but has already resulted in several new inventions in this area. Without the luxury of being able to design, code, and debug a new protocol in a few weeks, we would not have attempted this investigation, and would not have made the discoveries that led to new invention. Of course much more than a few weeks work went into the development of RAPID; but our experience with clock synchronization protocols shows that this work has a significant payoff.

6 Experience with RAPID as Scaffolding

Another current application of the RAPID environment is its use as scaffolding for high performance C code. In ongoing work reported in [BDHOS 93], we wrote a driver in RAPID–FL for a new transport layer that is intended to provide reliable UDP broadcast communication for a standardized set of operations in parallel computing called *collective communication*. The transport layer provides initialization, send, and receive interfaces. Our RAPID scaffolding was designed to read a script (corresponding to the collective communication calls of a parallel program) from a file, make appropriate calls to the transport layer, and measure the real time elapsed per call.

The original scaffolding was written in one person week and debugged in a second week. It allows for numerous tuning parameter changes and even script changes on the fly. The console operator, sets up the experiment by executing the RAPID–FL code on each participating network node. After the experiment has reached a steady state, the operator resets the RAPID statistics using the command "reset" as in the example of figure 5. Then after a suitable duration, the operator requests output using the command "report". The experiment may be stopped at this point, continue, or temporarily halt for a switch in script.

Two methods are used to estimate the contribution of the transport layer to the measured elapsed time of the experiment. One method involves extrapolating this time by plotting the time per call as we increase the number of calls to the transport layer in an inner loop of RAPID–FL that does not incur most of the overhead of RAPID because it stays in the handler program and does not return to the driver to check for console messages. The other method involves operating the code in a *disabled* mode in which all of the RAPID-FL code executes but the calls to the transport layer are disabled and take no time. The difference between operating in enabled and disabled modes represents the time contribution of the transport layer. Each method requires that the experiments be performed in similarly loaded networks, so there is some possibility for error if the load changes with time. However, the early experimental results reported in [BDHOS 93] seem robust and reproducible. Here we have successfully used RAPID to obtain absolute performance measurements for code that has been written for high performance.

7 Features Summary

In this section, we summarize the features of RAPID that make it a good environment for rapid prototyping.

Foremost of these features is **trivial storage management**. The RAPID–FL programmer is not concerned with the management of storage. In our experience we have observed that storage management accounts for a very significant proportion (perhaps even a majority) of systems programming bugs. Moreover, concerns with allocating and managing complex data structures certainly cost in terms of programming and debugging time. RAPID–FL data structures are sequence based. The atomic data types are reals, integers, Booleans, and character strings. Data structures can be built from these types by sequence construction. Elements of sequences may have any type including sequence. The programmer need not be concerned with uniformity. For example, a queue can be implemented using a sequence and append and select operators. The programmer need not worry about the type of elements to be put into the queue or whether they have uniform type.

A second major feature is **simple communication**. The programmer need not learn the intricacies of the underlying UDP and TCP protocols. Instead, there are simple driver commands for opening communication and sending messages. RAPID provides point-to-point communication via TCP/IP. If the command [~Open , [~"aa@bb.cc"]] is issued, then RAPID attempts to open a TCP/IP connection with a corresponding RAPID process on the host with internet name aa@bb.cc. This communication is not limited to the local network. If the command [~Open , [ ]] is issued, RAPID initializes UDP broadcast communication on the local network. In each case the driver returns an integer token that is used for sending messages and appears on each message that arrives via this logical communication link. Note that any RAPID data structure can be sent on a message and received at the target. The programmer need not worry about formatting or encoding data structures that are normally implemented as linked lists and arrays. RAPID handles all data conversions transparently.

A third feature of RAPID is that RAPID–FL is **easy to learn** because there is a relatively small number of language elements. The general purpose language, FL, contains much more than what the programmer needs to program in RAPID–FL.

An important feature that FL provides for RAPID is the ease with which we can adapt RAPID to **alternative programming styles**. The RAPID–FL features of assignment (/gets) and loops (DoWhile and DoUntil) are not natural constructs in FL; but are supported by higher order FL functions. We added these features to RAPID as needed by users of RAPID. The additions were easy to make because they simply involved additional FL programs in the RAPID libraries. No change to the FL compiler was required. We found it convenient to use a style of programming that required temporary global variables. A more functional style is also available and even more natural in FL.

The feature emphasized in section 3 and section 6 is that RAPID–FL **compiles to and interoperates**

with C. This feature enables the use of RAPID as scaffolding in the development of performance critical systems. It even allows for iterative development in which a series of prototypes is produced. The functions performed by the prototypes do not change, except as required by the designers. But the first prototype is coded entirely in RAPID–FL and the final prototype is coded in C, hopefully with ever better performance. Once a RAPID–FL program is compiled to C, it can be linked with other programs and compiled to produce an executable object. This object is just as portable as the C and does not require any further presence of the RAPID libraries in order to function. We have experimented with RAPID client-server prototypes operating via the internet from coast to coast of the U.S. The client prototypes were simply transferred to hosts at other locations where they could be executed immediately.

A final feature for this summary is what we call **snapshot debugging** using the /p operator. This operator allows the programmer to take snapshots of intermediate data structures and display them at the console during debugging runs of the prototype. There is no need for the programmer to know the type of the data structure being displayed.

Our experience suggests that programming and debugging RAPID–FL is an order of magnitude faster than programming and debugging in a lower level language like PL/I or C. The language seems to be ideal for the rapid prototyping of distributed protocols, assuming either that performance is not an issue, or that the prototype will eventually be replaced by one with better performance after issues of feasibility and completeness of design have been resolved.

8 Acknowledgements

The authors would like to thank Alex Aiken and John Williams for their response to requests for enhancements and bug fixes to the FL compiler. Thanks also to the RAPID users for their suggestions.

References

[FL 89] J. Backus, J. Williams, E. Wimmers, P. Lucas, and A. Aiken, FL Language Manual, Parts 1 and 2, IBM Research Report RJ7100, 1989.

[BDHOS 93] J. Bruck, D. Dolev, C. Ho, R. Orni, and R. Strong, PCODE: An Efficient and Reliable Collective Communication Protocol for Unreliable Broadcast Domains IBM Research Report RJ9631, 1993.

[C 89] F. Cristian, Probabilistic Clock Synchronization, *Distributed Computing* **3**, pp. 146-158, 1989.

[DHSS 84] D. Dolev, J. Halpern, B. Simons, and R. Strong, Fault-Tolerant Clock Synchronization, *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing*, 89-102, 1984.

[DSW 93] D. Dolev, R. Strong, and E. Wimmers, RAPID: An Environment for Rapid Prototyping of Distributed Protocols, IBM Research Report (in progress).

[M 84] K. A. Marzullo, Maintaining time in a distributed system: An example of a loosely coupled distributed service, Ph.D. dissertation, Stanford University, Stanford, CA, February, 1984.

[DCE 91] Distributed Computing Environment, Time Service Specification, Version T1.1.0, June 11, 1991.