Efficient Automated Protocol Implementation Using RTAG

Diane Hernek
David P. Anderson

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

August 21, 1989

ABSTRACT

RTAG is a system for automated implementation of communication protocols from formal specifications. The RTAG specification language is based on attribute grammars, and allows complex protocols to be specified concisely and with minimal need for additional program code. This paper describes a set of techniques for efficient automated implementation of protocols from RTAG specifications, and compares the performance to that of hand-coded protocol implementations. We conclude that in many cases the performance of RTAG-based protocol implementations is acceptable for experimental or production uses.

This work supported by NSF grant CCR-8619302 and by the Defense Advanced Research Projects Agency (DoD), Arpa Order No. 4871, monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089.

1. INTRODUCTION

A distributed computer system is defined by its communication protocols, which typically are divided into several layers. At the bottom are link-level and media access protocols. At the next level up, transport protocols provide reliable flow-controlled byte streams [18,25], or request/reply communication [4,11,21]. With the increased interest in distributed and replicated data, reliable multicast [10], atomic multicast [3], and distributed transaction commit [13,14] protocols have been proposed and implemented.

Low-level protocols can sometimes be implemented in hardware, but higher-level protocols must often be implemented in software. There are many software-engineering difficulties in software protocol implementation [12]. Performance is critical. Protocols usually involve asynchrony, concurrency, real time, and interrupt programming, so debugging is difficult. For performance reasons, protocols often must run in the operating system kernel where debugging support is minimal. Finally, in large heterogeneous systems (such as internetworks) protocol implementations must conform to a ''standard'' that is often under-specified and may change over time.

One approach to dealing with these problems is to generate protocol implementations automatically from formal specifications. In this approach, a protocol is described using a machine-independent formalism. Software tools convert this description into a partial implementation. Hand-coded routines are still generally needed to interface the generated code to the local operating system and other layers.

Compared to hand-coded implementation, the automated approach has several potential benefits. First, formal specifications are easier to develop and reason about than programs. Second, programming costs can be reduced, especially in heterogeneous systems where protocols change over time. Experimentation with protocols can be done by modifying a single formal specification, instead of changing and debugging several hand-coded implementations. The task of integrating a new machine or operating system type into the system involves writing a single set of standard interface routines, rather than reimplementing several protocols.

Despite these benefits, the automated approach is not widely used in practice. Formalisms based on finite-state machines are amenable to efficient implementation, but can express complex protocols only with the addition of substantial high-level language code and data structures. Algebraic and logic-based languages offer better abstraction facilities, but are difficult to implement efficiently.

In this paper we attempt to show that abstraction can be achieved without sacrificing efficiency. RTAG (Real-Time Asynchronous Grammars) [2] is a system for automated implementation of communication protocols from formal specifications. It provides a specification language, based on attribute grammars, in which complex protocols can be specified concisely and with minimal need for additional code. We will summarize the syntax and semantics of the RTAG specification language, and describe the implementation of protocols using RTAG. We then give performance measurements of an RTAG implementation of a typical transport-level protocol, and compare it with a hand-coded implementation of a similar protocol. We argue that the performance of the RTAG implementation is acceptable for experimentation and potentially for production use.

2. THE RTAG SPECIFICATION LANGUAGE

An RTAG specification describes the behavior of a single protocol agent, such as a transport protocol module. The specification includes a context-free grammar in which terminal symbols represent 1) messages received and sent by the agent, and 2) the passage of real time. Any string of terminal symbols that can be derived from the grammar's goal symbol is a valid "conversation" in the specified protocol. RTAG does not specify the format of messages. In practice, messages are translated between external form (*e.g.*, network packets) and RTAG's internal form (attributed symbols) by hand-coded interface routines.

We will describe the basic syntax and semantics of RTAG using an example: the sending end of a simple alternating-bit protocol.

2.1. Symbols and Attributes

An RTAG specification begins with symbol declarations. The nonterminal symbols used in our example are declared as follows:

```
nonterminal <goal>

nonterminal <packet tail>
   int      seqno

nonterminal <packet>
   dataptr data

nonterminal <get-ack>
```

Symbols can have typed *attributes*. The type `dataptr` represents a reference to a block of dynamically-allocated message data. Standard types such as `int` and `boolean` are also available. The terminal symbols used in our example are declared as follows:

```
input   [U->DATA]                 // data from user
   dataptr data

output  [U<-ACK]                  // ack to user

output  [N<-DATA]                 // data to network
   int      seqno
   dataptr data

input   [N->ACK]                  // ack from network
   int      seqno
```

Two terminal symbols are pre-defined: `/timer/` has an integer attribute `interval`, and corresponds to the passage of `interval` units of real time; `/freedata/` has an attribute `data` of type `dataptr`, and frees the memory referenced by `data`.

2.2. Productions

An RTAG specification also includes a list of productions. Our alternating-bit example includes the following productions:

```
<goal>:          <packet tail> .
   $1.seqno = 0
;

<packet tail>: <packet> <packet tail> .
   $2.seqno = ($0.seqno + 1) % 2
;

<packet>:        [U->DATA] [N<-DATA] <get-ack> /freedata/ .
   $0.data = $1.data
   $2.data = $1.data
   $2.seqno = <packet tail>.seqno
   $4.data = $1.data
;
```

These productions generate a sequence of `<packet tail>` symbols, each of which accepts a data message from the user, assigns it a sequence number, sends it to the network layer, and waits for an acknowledgment using `<get ack>` (described below).

As shown above, a production can have *attribute assignments* that are performed when it is applied. The assignment syntax is similar to that of C [15], and expressions can include C's arithmetic, relational and logical operators. An expression associated with a production *P* may use *local symbol references* of the form `$n` to refer to a symbol within *P*. `$0` refers to the left-hand side-symbol and `$n` (n ≥ 1) refers to the nth symbol of the right-hand side of *P*. A *non-local symbol reference* is of the form `<X>` (referring to the nearest ancestor of that name) or `<X>/<Y>` (referring to the leftmost named child of that ancestor).

Concluding the alternating-bit protocol specification, the following productions wait for an acknowledgement of a data message, periodically retransmitting the message until it arrives. Acknowledgements with the wrong sequence number are ignored.

```
<get-ack>:  [N->ACK] [U<-ACK] .
   if $1.seqno == <packet tail>.seqno

| /timer/ [N<-DATA] <get-ack> .
   $1.interval = TIMEOUT_INTERVAL
   $2.data = <packet>.data
   $2.seqno = <packet tail>.seqno
;
```

The `|` denotes an *alternative production*, meaning that an instance of `<get-ack>` can be expanded in either of two ways:

(1) If an acknowledgement (`[N->ACK]`) with the correct sequence number arrives, the first production is applied. This production has an *enabling condition* (a Boolean-valued expression that must be true in order for the production to be applied).

(2) If `TIMEOUT_INTERVAL` units of time elapse without an acknowledgement, the data packet is retransmitted.

The above protocol specification might generate a parse tree as shown in Figure 1.

2.3. Additional Features of RTAG

For more complex protocols, RTAG makes it possible to specify dynamically-created parallel *subprotocols*. This is done using *concurrent productions* which represent sets of parallel activities. The right-hand side of a concurrent production is enclosed in curly brackets:

```
<X>: {<Y> <Z>}
```

When this production is applied, the subprotocols represented by `<Y>` and `<Z>` proceed in

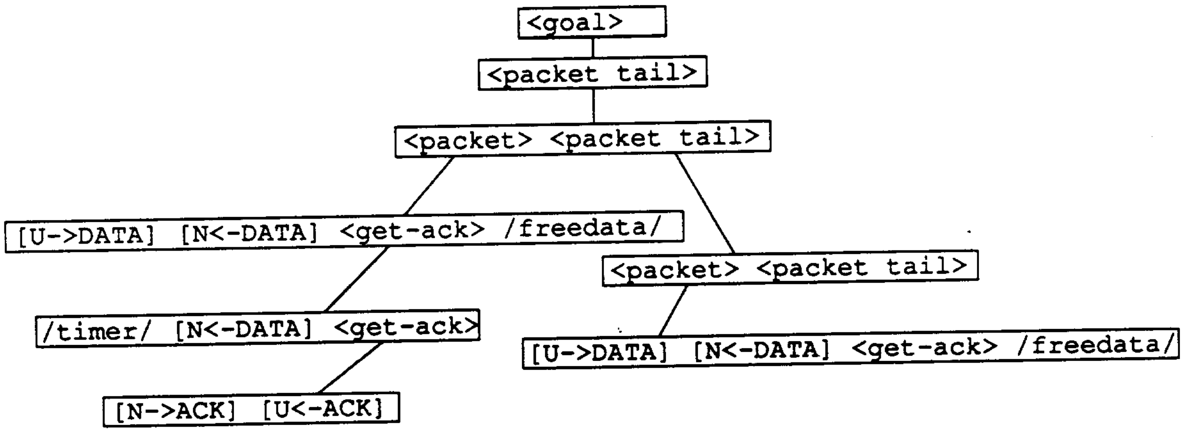
<goal>
<packet tail>
<packet> <packet tail>
[U->DATA] [N<-DATA] <get-ack> /freedata/
<packet> <packet tail>
/timer/ [N<-DATA] <get-ack>
[U->DATA] [N<-DATA] <get-ack> /freedata/
[N->ACK] [U<-ACK]

Figure 1: An example parse tree for the alternating-bit protocol described in Section 2.2.

parallel. This is useful in specifying protocols that have multiple sessions (*e.g.*, connections or transactions) or concurrent activities within a session (*e.g.*, the sending and receiving tasks of a full-duplex protocol, or different messages in a sliding window protocol).

When concurrent productions are used, several parts of and RTAG parse tree can be "active" simultaneously. When an input message arrives, there are potentially several nonterminals from which it can be derived. RTAG's semantics, in this case, are that the input message is derived from as many nonterminals as possible. Hence a single acknowledgment message, which might serve to acknowledge several outstanding data messages, can be handled separately in the subtrees corresponding to the data messages. Inputs that cannot be derived are ignored.

To allow messages to be matched up efficiently with sessions, an attribute name may be declared as the *key* attribute. At most one nonterminal symbol in a specification can have the key attribute. An input symbol having the key attribute can be derived only within a subtree rooted at a nonterminal symbol with the same key attribute value.

3. THE OPERATIONAL SEMANTICS OF RTAG

This section gives an informal description of RTAG semantics. To do this, we describe the behavior of an *RTAG parser* that interprets RTAG specifications. (the implementation of this abstract model will be discussed in Section 4). An RTAG parser implements a single protocol entity (such as a transport protocol module) by interpreting an RTAG specification of the protocol. The parser is part of a system (*e.g.*, an operating system kernel) in which messages can arrive asynchronously. Each output symbol is associated with an interface routine whose arguments correspond to the symbol's attributes. The parser maintains queues of incoming messages, timeouts, and internal events; this organization is shown in Figure 2.

The parser maintains a *parse tree* of attributed symbol instances. Initially, this tree consists of a single instance of the `<goal>` symbol. Productions are applied in response to 1) input events and 2) the passage of real time. A symbol instance *X* goes through the following sequence

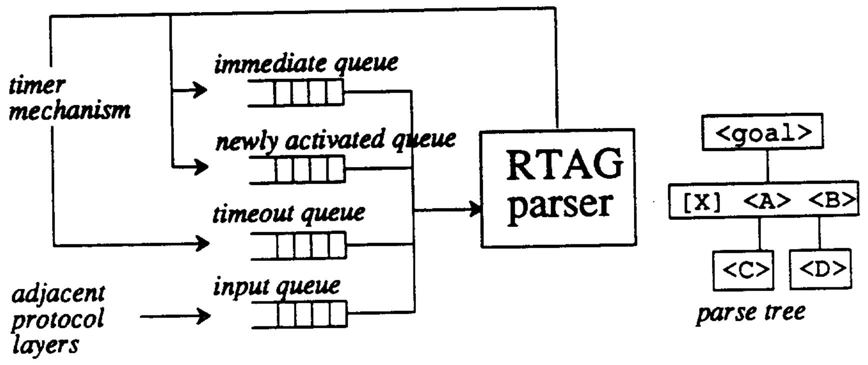

Figure 2: Components of an Abstract RTAG Parser.

of *states* during its lifetime:

- *Inactive*: Inactive nonterminals are not eligible for expansion. A symbol is initially inactive.
- *Active*: In a sequential production, symbol instances are activated when all of their left siblings are fully expanded (see below). All symbol instances in a concurrent production are activated immediately when the production is applied. When an output symbol is activated, the corresponding interface routine is called. When a nonterminal symbol is activated, it becomes eligible for expansion.
- *Expanded*: If X is a nonterminal, it becomes expanded when a production is applied to it.
- *Fully expanded*: An input or `/timer/` symbol is fully expanded when it is created. An output symbol becomes fully expanded when the corresponding action has completed. If X is a nonterminal, it is fully expanded when all its descendents are fully expanded.

We say that a nonterminal symbol <X> *yields* a terminal symbol `[T]` if `[T]` can be left-derived from <X> by applying a sequence of enabled productions. For the purposes of this definition, all symbols in a concurrent production RHS are considered leftmost.

An *immediate production* is one that can be applied as soon as it becomes enabled, without waiting for the arrival of an input event or the passage of any time. More precisely, P is an immediate production of `<X>` if *P* is part of a production sequence that left-derives epsilon (the empty string), an output symbol, or `/freedata/`.

A production *P* can be applied to an active nonterminal `<X>` if *P* is enabled and either 1) an input event `[A]` occurs, and *P* is the first production in a sequence that yields `[A]` from `<X>`; 2) the first symbol of *P* is `/timer/`, and the specified amount of time has elapsed since `<X>` became active, or 3) *P* is an immediate production of `<X>`.

In operational terms, the RTAG parser works as follows. The four queues shown in Figure 2 are maintained. While any of the queues is non-empty, an entry is removed from one of them and processed. Otherwise the parser waits for a queue to become nonempty.

If a nonterminal `<X>` is activated, and `<X>` has an enabled timed production *P*, a timer is started. When the timer expires, a record containing a reference to `<X>` and to *P* is placed on the timeout queue. When the entry is processed, *P* is applied to `<X>` if it is still active.

When an attribute value is changed by an attribute assignment, an immediate production of an active nonterminal may become enabled as a result. The parser handles this by placing symbols with an immediate production whose enabling condition depends on the changed attribute on the *immediate queue*. Entries in this queue are handled as follows: if the symbol is still active, the enabling condition is re-evaluated, and if true the immediate production is applied.

When a symbol becomes active, the necessary processing (*i.e.*, checking immediate productions and starting timers) is not necessarily done immediately. Instead, the symbol is put on the *newly-active queue*, and this processing is done when it is removed from the queue.

An input symbol instance *s* is processed as follows: First, its *candidate set* is computed. This is the set of active nonterminal instances *X* from which *s* can potentially be derived, based on CFG information and key attribute values. Then the parser attempts to derive the input symbol from each of the candidates in turn.

4. THE DESIGN OF AN EFFICIENT RTAG PARSER

The previous section suggests an approach to RTAG-based automated protocol implementation. A prototype RTAG system [1], implemented the model in a straightforward way. This "old parser" interpreted a tokenized form of the RTAG specification, and its parse-tree data structure was very complex. As a result, its performance was poor: it used up to 100 times the CPU of hand-coded implementations.

We have developed a new implementation of RTAG with the goal of improving its performance. This system consists primarily of an *RTAG compiler* and an *RTAG parser*. The RTAG compiler, given an RTAG specification of a protocol, generates static data structure definitions and function definitions in the C language. An RTAG parser, combined with these data structures, this code, and system-dependent interface routines, implements the specified protocol.

Implementing several protocols on a number of different target machines with this RTAG-based system requires the following components (see Figure 3):

- The RTAG compiler running on a single host machine.
- An RTAG parser running on each target machine.
- A machine-independent RTAG specification of each protocol.
- A set of machine-dependent interface routines and externally-defined functions for each (*protocol, target machine type*) pair.

In the remainder of this section we discuss the techniques used in the new RTAG parser to increase performance. Some of the techniques reduce the overall CPU usage of the parser; others take CPU time out of the "critical path" between input events and their response, transferring it to "background" activities. In Section 5 we discuss the overall performance of the parser.

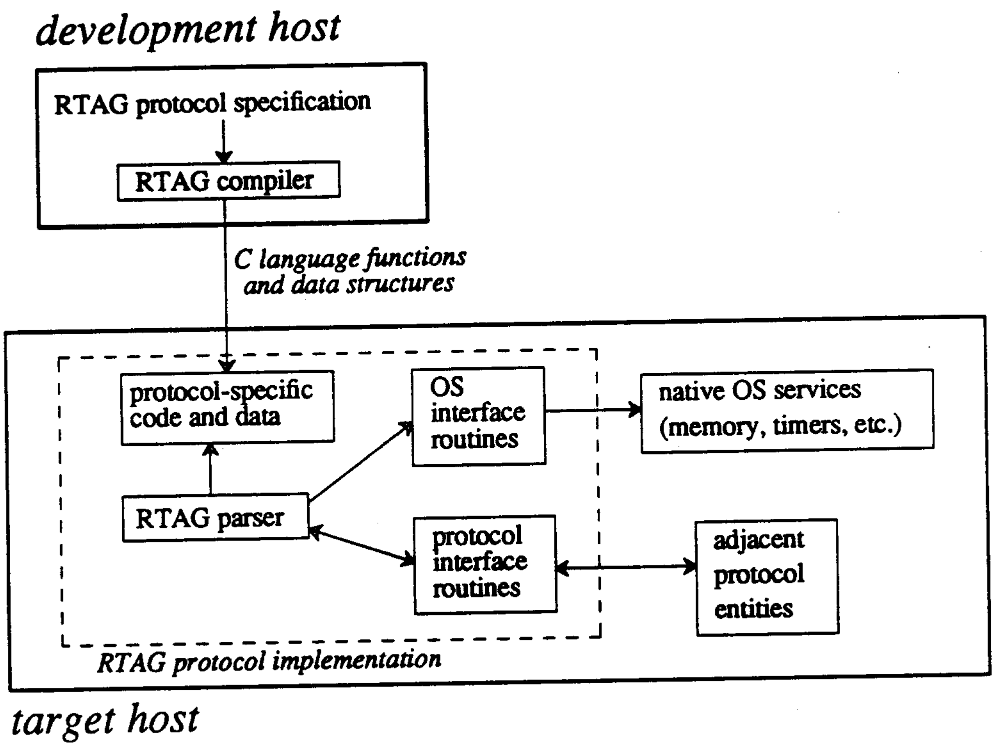
development host
RTAG protocol specification
RTAG compiler
C language functions and data structures
protocol-specific code and data
OS interface routines
native OS services (memory, timers, etc.)
RTAG parser
protocol interface routines
adjacent protocol entities
RTAG protocol implementation
target host

Figure 3: The software structure of an RTAG-based protocol implementation. The protocol developer supplies 1) the RTAG protocol specification and 2) routines that interface to the local operating system and to other protocol layers.

4.1. Automatically Generated Code

The RTAG compiler generates C functions that perform 1) enabling condition evaluation, 2) attribute assignments, and 3) initialization of RHS block structures (see §4.5). These functions are compiled into the RTAG parser. In contrast, the old RTAG parser evaluated expressions interpretively.

4.2. Precomputed Candidate Sets

The parser maintains a data structure mapping each possible (*key attribute value*, *input symbol type*) pair to the corresponding candidate set. A nonterminal symbol instance *X* is added to all appropriate candidate lists when it is activated and no immediate production is enabled.

A hash table is used so that finding the candidate set for an input symbol takes constant time. (In contrast, the old parser recursively traversed the parse tree). The data structure uses *hints* (see §4.6) so that it need not be updated when active symbols are expanded. The extra work of adding symbols to candidate sets is small if nonterminals yield a small number of input symbols in the CFG, as is usually the case. This work can be done in the background, as can the task of "garbage-collecting" outdated table entries.

4.3. Non-Local Symbol References

The enabling conditions and attribute assignments of a production may refer to attributes of *non-local* symbols (ancestors of the symbol being expanded). The old parser resolved non-local references by traversing upwards in the parse tree. The new parser uses *downward propagation* of symbol pointers. In this technique, the RHS block (see §4.5) for a production *P* contains pointers to all of its ancestor symbols that are referenced in a production that is a potential descendant of *P*. These pointers are initialized (directly, or by copying from the parent RHS block) by the initialization function for *P*.

The following example (the sending end of a sliding-window protocol) illustrates downward propagation of symbol pointers. In this specification, the `window_end` attribute of the `<goal>` symbol stores the current end of the send window. The `<packet>` subprotocol receives data from the user, waits until it is within the send window, sends it, waits for an acknowledgement, updates the send window, and frees the data.

```
<goal>:         <packet tail> .                                  (1)
   $1.seqno = 0
;

<packet tail>: { <packet> <packet tail> } .                      (2)
   $2.seqno =  $0.seqno + 1
;

<packet>:       [U->DATA] <send packet> .                        (3)

<send packet>: [N<-DATA] <get ack> /freedata/ .                  (4)
   if <goal>.window_end >= <packet tail>.seqno
;

<get ack>:   [N->ACK] .                                          (5)
   if $1.seqno == <packet tail>.seqno
   <goal>.window_end = $1.credit
;
```

Since productions (4) and (5) reference `<goal>.window_end`, a pointer to `<goal>` is stored in the RHS blocks of all occurrences of productions (2) through (5) when they are applied. Likewise, a pointer to the `<packet tail>` ancestor is stored in all RHS blocks containing `<send packet>`.

Using the downwards propagation scheme, expressions involving non-local attribute references can be evaluated efficiently; a single indirect memory reference is used. In contrast, the old parser could have to traverse a branch of the parse tree for each reference. The cost of the new scheme is the actual propagation of pointers, not all of which may be used. However, in practice only one or two pointers are copied for each production.

4.4. Immediate Links

A change in the value of an attribute can potentially enable immediate productions of symbols elsewhere in the tree. An attribute *a* of a nonterminal *X* is called a *linkage attribute* if

(1) *a* is referenced in the enabling condition of an immediate production of *X*, and

(2) *a* is referenced non-locally as the target of an attribute assignment.

Whenever *a* is modified, the parser must locate all symbols of which it is a linkage attribute, and re-evaluate the enabling conditions of their immediate productions. (For example, in the

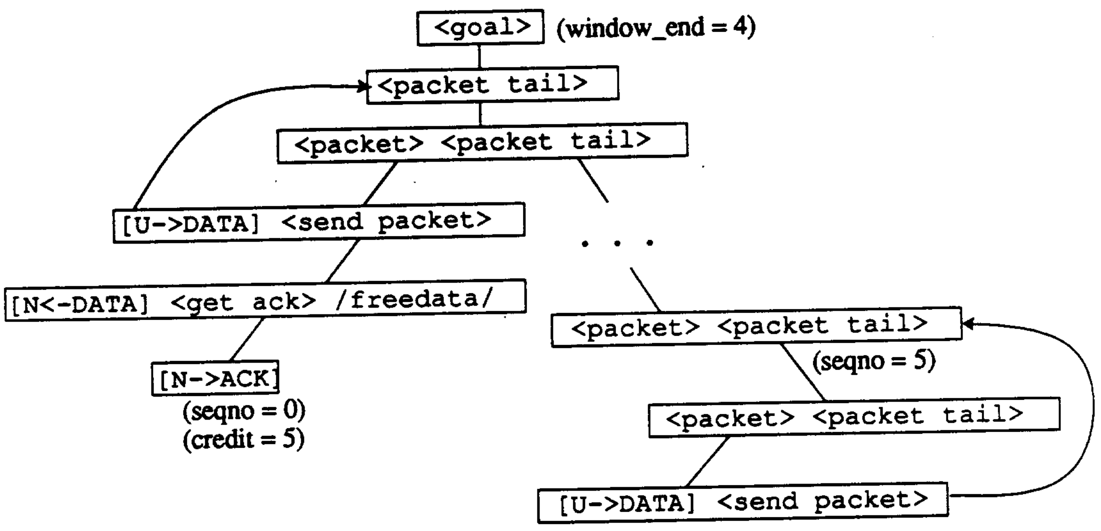

Figure 4: An example of symbol pointers and linkage attributes. In addition to the pointers shown, all RHS blocks have a pointer to the `<goal>` symbol. These pointers allow non-local attribute references to be evaluated efficiently.

example above the `window_end` attribute of `<goal>` is a linkage attribute.) Locating the symbols is perhaps the most difficult (and potentially slow) part of an RTAG parser algorithm. The general approach is to maintain *immediate links* from linkage attributes to the symbols whose productions they may enable.

The old parser maintained immediate links as a list, attached to each attribute, of pointers to symbols. Reverse pointers were also maintained so that, when a symbol was expanded, it could be removed from these lists. This "brute force" approach had high CPU overhead. The new parser stores all immediate links in a single hash table. Each entry contains pointers to 1) a linkage attribute and 2) a symbol it may enable. When a linkage attribute is modified, the parser checks the table and adds the necessary symbols to the immediate queue. When a nonterminal *X* is activated and no immediate production is enabled, an automatically-generated function establishes immediate links between *X* and each of its linkage attributes. The entries in the immediate-link table are "hints" (see §4.6) so that no clean-up need be done when X is expanded. Garbage collection is done in the background.

The parse tree shown in Figure 4 illustrates the use of linkage attributes. The scenario is as follows. Packets 0 through 4 have been sent. The `<send packet>` symbol for packet 5 is waiting for the send window end to be increased. An acknowledgement arrives for the packet 0, and it shifts the send window to include packet 5. Using the symbol pointer stored in the RHS block of `<get ack>`, `<goal>.window_end` is updated. Because this is a linkage attribute, the table is checked, and as a result the `<send packet>` symbol for packet 5 is placed on the immediate queue. When the symbol is later processed, its enabling condition will be re-evaluated and the production will be applied.

4.5. Memory Allocation

The old RTAG parser allocated separate blocks of memory for each symbol instance and attribute instance. The new parser allocates memory in *RHS blocks*, each of which represents the RHS of a production P. A RHS block has a header followed by a sequence of symbol instances (one for each RHS symbol of P). Each symbol instance is followed by a sequence of attribute instances (one for each of its attributes). Non-local pointers are stored at the end of the RHS block. For efficiency, RHS block allocation is done using fixed-size blocks (the size of the largest RHS block).

The use of RHS blocks simplifies symbol and attribute addressing, and reduces the overhead of assigning UIDs (see §4.6) Allocating fixed-size blocks results in internal fragmentation, but it reduces the amount of time needed for allocation.

4.6. Hint-Based Data Structures

The purpose of several of the parser's data structures (timer records, candidate lists and immediate links) is to locate a nonterminal instance X when an event (a timeout, the arrival of an input message, or a change in an attribute value) occurs. When X is expanded by a production, all references to it in these structures become outdated. Rather than update the data structures each time a symbol is expanded, we represent the structures using *hints*[1]. Outdated entries remain in the data structures and all entries are checked for validity during normal processing. Any entries found to be invalid are removed at that time.

Validity is determined by checking first that the symbol has not been removed, that is, that the RHS block has not been freed. This is done using unique identifiers, or *UID*s. When a RHS block is allocated, it is assigned a UID (a 32-bit sequence number). When the RHS block is deallocated, it is given a special invalid UID. When a reference to a symbol instance is stored in a timer record, candidate record or immediate link, the UID of its RHS block is stored with it. A symbol is checked for existence by comparing the UID in the record with the UID in the RHS block. If the UIDs match (*i.e.*, the symbol still exists), the parser determines whether it is still eligible for expansion by checking a state field in the symbol instance.

This technique simplifies the parse tree data structure and reduces the cost of deleting symbols from the tree. It may increase the number of entries processed, since some invalid pointers may have to be traversed. However, it is possible to garbage-collect data structures such as the candidate set and the immediate link tables in the background.

5. THE PERFORMANCE OF RTAG PROTOCOL IMPLEMENTATIONS

In this section we estimate the performance of RTAG protocols relative to that of hand-coded implementations of the same protocols. We give CPU time measurements for typical protocol operations, and predict system-wide performance measures such as throughput, CPU load, and RPC delay. Our conclusion is that, for a range of hardware parameters, the RTAG protocols perform well enough for experimental and, in some cases, production use.

[1] Hints have been used in other types of software systems, including file system directory structures [19], monitors [16], and distributed naming systems [23]

As a benchmark, we use a general-purpose stream transport protocol. This type of protocol uses several common protocol mechanisms: timer-based retransmission, sliding transmission window, and multiple concurrent sessions. (Similar mechanisms are used for request/reply communication as well, so our conclusions are likely to hold for a range of protocols.) The sending and receiving parts of the protocol are roughly equal in CPU requirements, so we analyze only the sending end.

We developed an RTAG specification of the Xerox Sequenced Packet Protocol (SPP) [25], a protocol of the type described above. As a representative hand-coded implementation, we use the Delta-t protocol [24]. Except for connection establishment (which we do not discuss) Delta-t is basically similar to SPP.

We analyze the case where a user program makes a system call to send a large block of data on an existing connection. The resulting CPU activity is shown in Figure 5. Following the initial processing of the system call, the protocol enters into a steady state in which it repeatedly receives an acknowledgement and transmits another data packet. Each of these steps includes processing in the session layer (buffer management and copying data from user space to packet buffers), within the transport layer, and at the network layer and below (data checksumming, routing, and device driver).

Watson and Mamrak implemented the Delta-t protocol in the VMS operating system running on a DEC VAX 11/750. They performed detailed microsecond-level measurements of the CPU time used by the various components of protocol processing. We generalize their measurements to include the following parameters:

C = CPU speed in MIPS
P = amount of user data per data packet (bytes)

We assign the VAX 11/750 a MIPS rating of 0.75. We assume that the protocol CPU times will vary inversely with C, and that the portions of time involving user data operations (copying and checksumming) will increase linearly with P. All times are in microseconds.

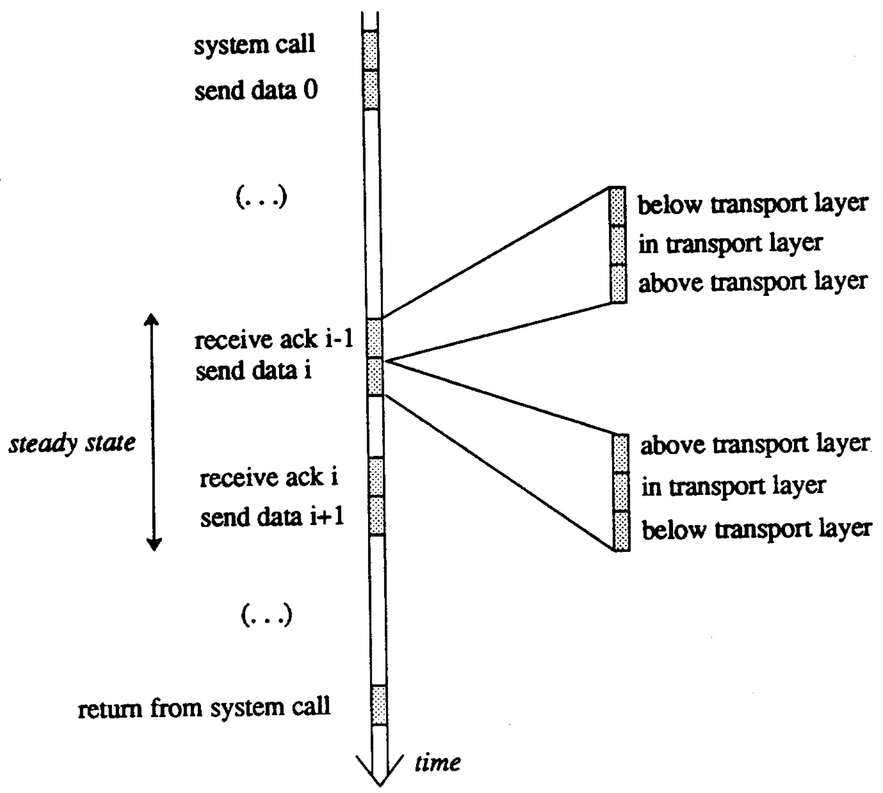

Figure 5: The CPU activity (shaded areas) involved in sending a large block of data.

Extrapolating from the measurements of Watson and Mamrak, the CPU time of a system call (trapping to the kernel, doing other system call processing, and later returning) is

$$S = 875/C$$

The CPU time D for transmitting a data packet (in the hand-coded implementation) has the following components.

$$D_{session} = (388 + 0.450P)/C$$
$$D_{transport} = 424/C$$
$$D_{network} (631 + 0.354P)/C$$

$$D = D_{session} + D_{transport} + D_{network} = (1443 + .804P)/C$$

The CPU time A for processing an ack has the following components:

$$A_{session} = 351/C$$
$$A_{transport} = 407/C$$
$$A_{network} = 835/C$$

$$A = A_{session} + A_{transport} + A_{network} = 1593/C$$

The total CPU time for processing an ack and sending the next data packet is therefore

$$T_{hand-coded} = A + D = (3036 + 0.804P)/C$$

We measured the CPU time used by the RTAG implementation of SPP, also on a VAX 11/750. The transport protocol CPU time needed to process an ack and send the next data packet was found to be:

$$A_{transport} + D_{transport} = 8909/C$$

In the following performance comparison, we assume that the RTAG protocol uses the same session and network layers as the hand-coded protocol, so the total CPU time to handle an ack and send the next data packet is

$$T_{RTAG} = (11114 + 0.804P)/C$$

5.1. Performance Comparison for Stream Communication

We analyze the steady-state operation of a stream transport protocol under the assumptions that 1) the protocol sends a large enough amount of data so that it reaches steady state between system calls; 2) no packets are retransmitted; 3) a sufficiently large window size is used, and 4) every data packet is acknowledged. In the steady state, the system repeatedly receives an ack, processes it (at the network, transport, and session levels), and transmits the next data packet.

Let B denote the throughput (assumed to be constant) of the network itself, measured in bytes per second. In steady state, a new data packet can be sent at most every P/B seconds. If $P/B \le T$ then the CPU in the sending host is the system bottleneck. An ack is present immediately after each data packet is sent, the CPU is busy 100% of the time doing protocol processing, and the network is not fully loaded. One packet of size P is sent every $1/T$ seconds, so end-to-end throughput is P/T bytes/second (packet headers are assumed to have negligible length).

If, on the other hand, $P/B > T$ then the system is I/O-bound, *i.e.*, the network is the bottleneck. The network is capable of absorbing B/P packets per second, so the CPU load due to protocol processing is TB/P.

Figure 6 illustrates CPU load as a function of CPU speed for several values of packet size and network bandwidth. It can be seen from these graphs that, for a range of combinations of

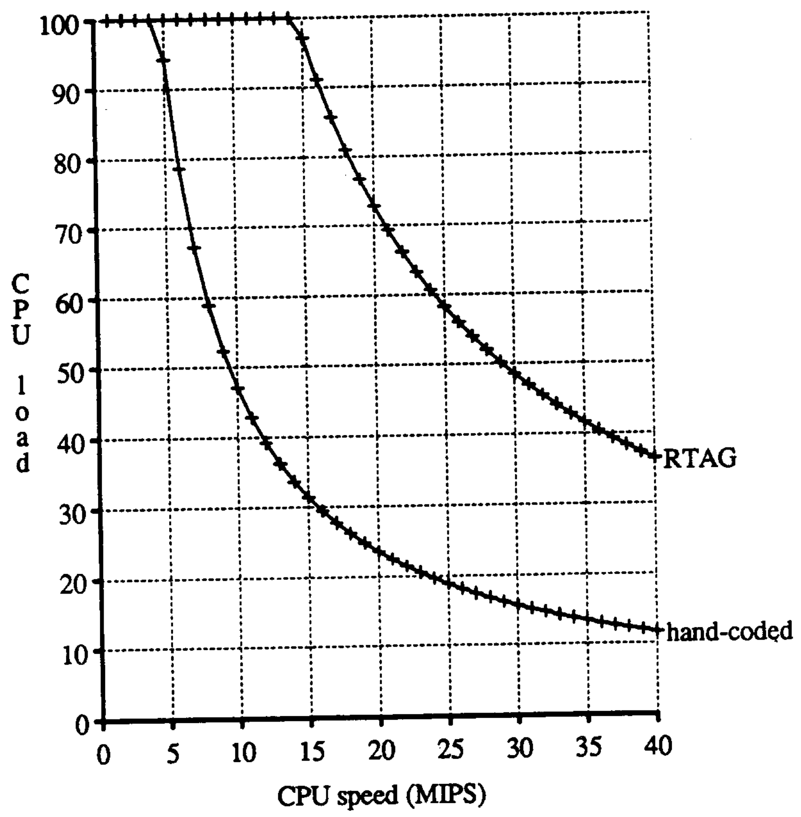

Figure 6a: Packet size = 1024 bytes, network bandwidth = 10 Mbps.

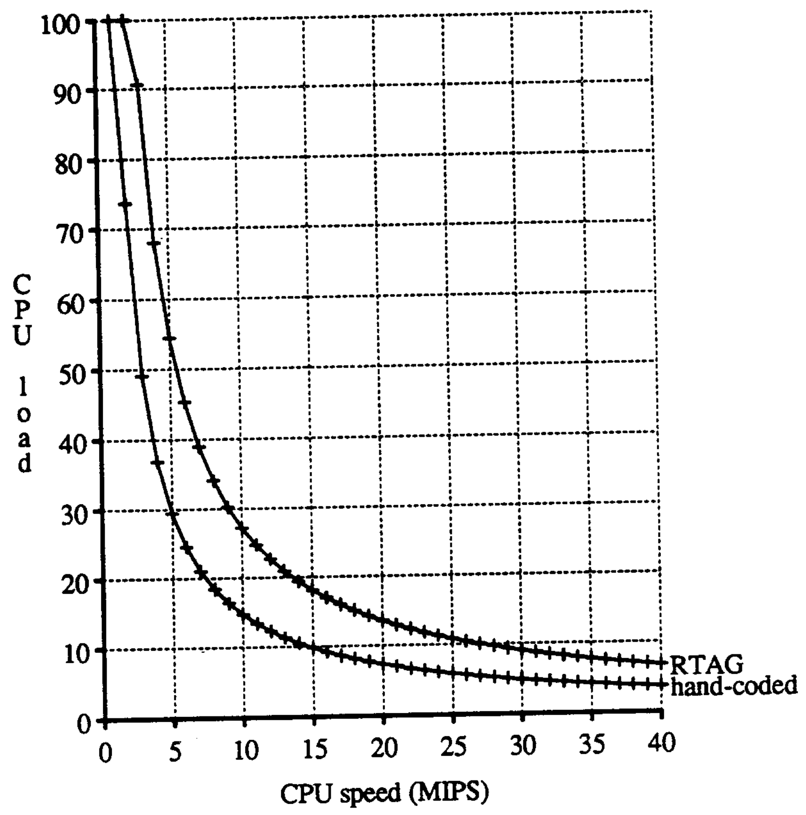

Figure 6b: Packet size = 8096 bytes, network bandwidth = 10 Mbps.

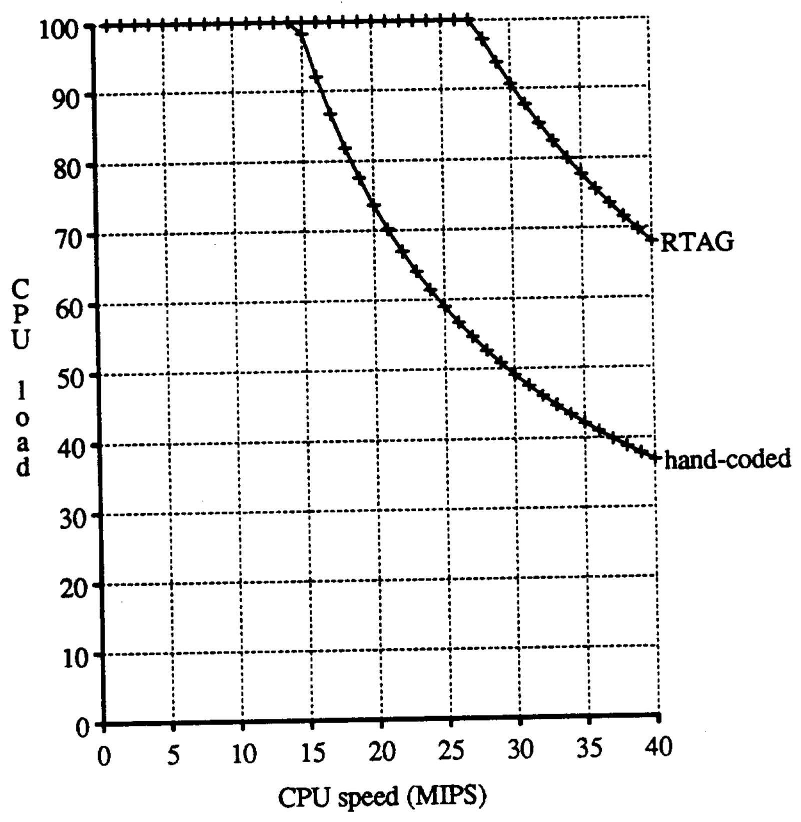

Figure 6c: Packet size = 8096 bytes, network bandwidth = 100 Mbps.

CPU speed, network bandwidth, and packet size, the performance of the RTAG protocol is not significantly worse than that of the hand-coded protocol. In the flat (CPU-bound) part of the curves, RTAG gives a lower throughput than hand-coded. The throughput ratio depends on the packet size, and approaches one as the packet size increases. In the tail (I/O-bound) part of the curves, the RTAG and hand-coded protocols achieve the same throughput, but the RTAG protocol imposes a larger CPU load. Again, the difference depends on packet size. The absolute size of the load diminishes as CPU speed increases.

5.2. Performance Comparison for Request/Reply Communication

We now estimate the difference between RTAG and hand-coded protocols for request/reply communication (*e.g.*, remote procedure call). In this case, a user-level "client" process issues a system call to send a request message over an existing connection. The message is received and delivered to a waiting "server process" which prepares and sends a reply message. The reply message is then delivered to the waiting client.

The delay seen by the client is the sum of many terms: 1) argument and result marshalling and demarshalling; 2) client process system call and return; 3) protocol processing for the request and reply messages, in the session, transport, and network layers, in both the client and server hosts; 4) network queuing delay, transmission, and propagation times; 5) scheduling of the server process, and 6) service time. Of these, the use of an RTAG transport protocol affects only the CPU time of transport protocol processing. Assume that the request and reply messages each occupy one packet. As can be seen from the measurements given earlier, the difference between RTAG and hand-coded is on the order of

$$11114/C - 3036/C = 8078/C$$

at the sender, and about the same at the receiver. In other words, the absolute difference in delay is on the order of 1.6 milliseconds for 10 MIPS machines. The relative difference, of course, depends on the other delay components. For operations such as file access, the service time alone might be tens of milliseconds, so the RTAG protocol would not make a significant difference.

Furthermore, the RTAG CPU times can be divided into high- and low-priority components. The high-priority component is the time required for RTAG to receive an input event, locate its candidates and derive it from each of them, generating any output events that may be triggered. The low-priority component is the time spent removing fully-expanded symbols from the parse tree, processing the immediate and newly active queues, and garbage-collecting linkage structures. Our measurements show that the high-priority activities use only 20 to 25% of the total

CPU time. Since the low-priority activities are not in the ''critical path'' of request/reply communication, the actual delay different will be smaller than that given above.

6. COMPARISON WITH RELATED WORK

The merits of specifying protocols using formal description techniques (FDTs) and generating implementations automatically from specifications are well known [6,17,22]. A number of protocol specification techniques have been developed. Most, for example, Estelle [7], FAPL [20] and Blumer and Tenney's system [5], have been based on finite-state machines (FSM). Others, such as LOTOS [8,9], are algebraic languages.

The finite-state model has some weaknesses for specification of complex protocols. It is difficult to model protocols as a dynamic set of concurrent activities. Because information in an FSM can only be stored in the current state, the number of states necessary to express protocols (for example, a window protocol with sequence numbers) quickly becomes very large. Most FSM-based systems rely upon separate high-level language code to express important parts of protocols. Finally, FSM descriptions are difficult to read and understand because they do not make apparent the common event sequences.

Algebraic specifications solve the above problems of FSMs, but have the disadvantage of being difficult to translate into implementations. To date, algebraic techniques have only been used to specify and verify communication protocols; no automatic implementations generated from them have been reported.

RTAG shares many of the advantages of algebraic specifications. Complex protocols can be decomposed into a hierarchy of separately-specified subprotocols. Attributes associated with grammar symbols are used for data exchange and synchronization between subprotocols. Event sequences are apparent from the specification syntax. RTAG is more powerful than FSM-based techniques. While it is always possible to convert an FSM to an equivalent context-free grammar, the reverse is not true. At the same time, as we have attempted to show in this paper, RTAG is amenable to efficient automated implementation.

7. CONCLUSION

We have described the RTAG system for automated protocol implementation from formal specifications. The RTAG language, because of its support for parallelism and remote attribute references, allows complex protocols to be specified more completely and with better structure than FSM-based formalisms. We have also described a software system for automatic generation of protocol implementations from RTAG specifications. The system uses many techniques to improve performance. The most important of these techniques are:

- The use of automatically-generated native code for various parser functions (*e.g.*, to initialize a RHS block, and to evaluate enabling conditions), instead of interpretation of data structures.
- Downwards propagation of non-local symbol references.
- Precomputation of candidate sets.
- The use of hint-based data structures. All references to symbols (in queues, timer records, candidate sets, and immediate links) are stored as hints that include an expected UID of the symbols. This allows symbols to be deleted without cleaning up links to them.
- The distinction of high- and low-priority protocol processing; hint-based structures can be garbage-collected when no output actions are pending.

RTAG protocol implementations are slower than their hand-coded counterparts. However, as we have shown, the differences in transport-level CPU time may have little impact on system-level performance. When the system is I/O-bound, end-to-end throughput is the same for RTAG as for hand-coded, and in many cases CPU load is only slightly higher.

8. REFERENCES

1. D. P. Anderson, A Grammar-Based Methodology for Protocol Specification and Implementation, Ph.D. Thesis, University of Wisconsin - Madison, August 1985.
2. D. P. Anderson, "Automated Protocol Implementation with RTAG", *IEEE Transactions on Software Engineering*, March 1988.
3. K. P. Birman and T. A. Joseph, "Exploiting Virtual Asynchrony in Distributed Systems", *Proc. of the 11th ACM Symp. on Operating System Prin.*, Austin, Texas, Nov. 8-11, 1987, 123-137.
4. A. Birrell and B. Nelson, "Implementing Remote Procedure Calls", *ACM Trans. Computer Systems 2*, 1 (Feb. 1984), 39-59.
5. T. P. Blumer and R. L. Tenney, "A Formal Specification Technique and Implementation Method for Protocols", *Computer Networks 6*, 3 (July 1982), 201-217.
6. G. Bochmann and C. A. Sunshine, "Formal Methods in Communication Protocol Design", *IEEE Trans. on Commun. 28*, 4 (April 1980), 624-631.
7. G. Bochmann, G. W. Gerber and J. M. Serre, "Semiautomatic Implementation of Communication Protocols", *IEEE Transactions on Software Engineering 13*, 9 (September 1987).
8. J. P. Briand, M. C. Fehri, L. Logrippo and A. Obaid, "Structure of a LOTOS Interpreter", *SIGCOMM '86 Symposium, Stowe, Vermont*, August 1986.
9. E. Brinksma, "A Tutorial on LOTOS", *Proc. 5th IFIP Symposium on Protocol Specification, Testing, and Verification*, June 1985.
10. J. Chang and N. F. Maxemchuk, "Reliable Broadcast Protocols", *ACM Trans. Computer Systems 2*, 3 (Aug. 1984), 251-273.
11. D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations", *Proc. of the 9th ACM Symp. on Operating System Prin.*, Bretton Woods, New Hampshire, Oct. 10-13, 1983, 128-140.
12. D. Clark, "Modularity and Efficiency in Protocol Implementation.", *RFC 817, SRI Network Information Center*, July 1982. MIT Laboratory for Computer Science, Computer Systems and Communications Group.
13. J. Gray, "An Approach to Decentralized Computer Systems", *IEEE Trans. on Software Eng. 12*, 6 (June 1986), 684-689.
14. R. Haskin, Y. Malachi, W. Sawdon and G. Chan, "Recovery Management in QuickSilver", *ACM Trans. Computer Systems 6*, 1 (Feb. 1988), 82-108.
15. B. W. Kernighan and D. M. Ritchie, "The C Programming Language", *Prentice-Hall, New Jersey*, 1978.
16. B. W. Lampson and D. D. Redell, "Experience with processes and monitors in Mesa ", *Communications of the ACM 23* , 2 (Feb. 1980), 105-117 .
17. N. Nounou and Y. Yemini, "Development Tools for Communication Protocols: An Overview", *IEEE Global Telecommunications Conference*, November 1984.
18. J. Postel, "Transmission Control Protocol", *DARPA Internet RFC 793*, Sep. 1981.
19. D. D. Redell, "Pilot: An Operating System for a Personal Computer", *Communications of the ACM 23*, 2 (February 1980), 81-92.
20. G. D. Schultz, D. B. Rose, C. H. West and J. P. Gray, "Executable Description and Validation of SNA", *IEEE Trans. Commun. COM-28* (1980), 661-677.

21. A. Spector, ''Implementing Remote Operations Efficiently on a Local Network'', *Comm. of the ACM 25*, 4 (Apr. 1982), 246-260.

22. C. A. Sunshine, ''Formal Techniques for Protocol Specification and Verification'', *Computer 12* (1979), 20-27.

23. D. B. Terry, ''Caching Hints in Distributed Systems '', *IEEE Transactions on Software Engineering* , January 1987 P 48-54 .

24. R. W. Watson and S. A. Mamrak, ''Gaining Efficiency in Transport Services '', *ACM Transactions on Computer Systems 5* , 2 (May 1987), 97-120 .

25. ''Internet Transport Protocols'', *Xerox Corporation, Xerox System Integration Standard 028112*, December 1981.