Correct Implementation of Network Protocols from Abstract Specifications

Tommy M. McGuire

March 22, 2000

Abstract

Abstract specifications for network protocols are usually based on several assumptions that make verification easier but which make strictly faithful implementations of the specified protocols impractical, expensive, or impossible. (These assumptions usually deal with timeouts, action atomicity, action fairness, message propagation, and fault atomicity.) An implementation, therefore, is an approximation of the given abstract specification. In this dissertation, we develop an execution model that can be used in translating abstract protocol specifications written in the AP notation into C programs. Although this execution model does not strictly abide by the formal semantics of the AP notation, we show that it does maintain some important safety and progress properties of the original specifications.

1 Abstract Specifications of Protocols

A protocol specification has two complementary objectives. On one hand, the specification should be demonstrably correct. To support this, the notation used by the specification should hide unnecessary details to simplify the task of verifying the correctness of protocol specifications. On the other hand, the specification should be implementable. A good implementation of the specification should maintain the important properties of the specification.

Experience has shown that informal language is inadequate to specify network protocols: verification of correctness is impossible and good implementation is difficult to achieve. Unfortunately, the ambiguity and lack of good notation inherent in natural language conceal these problems, with the result that an apparently clear and thorough description of a network protocol may in fact conceal any number of pitfalls.

For example, the Transmission Control Protocol is specified in English[16, 4] and has been independently implemented many times using these informal specifications. This has led to many inconsistencies and significant problems in independent implementations of TCP which degrade the behavior of the protocol and endanger the state of the network[15].

A formal notation provides structure for describing protocols and supporting verification and implementation. A formal notation can either be strict or permissive.

- A permissive notation, such as C[18], Prolac[13], Promela++[2], Teapot[5], and Morpheus[1], is designed to make implementation easy. Such a notation allows protocol actions which are easy to implement and often allows greater flexibility of the specification than an abstract notation. Flexibility, however, obscures the properties of the protocols, and permissive notations are not intended to address verification.
- A strict, or abstract, notation, such as Estelle, LOTOS, SDL (all described in [17]), Esterel[3], and Promela[12], is designed to make verification easy. For example, such a notation abstracts away any details which interfere with the verification of the properties of the specification while placing strict limits on the actions of the specification.

In this dissertation, we choose to begin with an abstract notation for specifying network protocols and use that notation as the basis for implementing protocols.

2 Protocol Implementations

An abstract notation for specifying network protocols provides three models:

1. *A model of time*, which defines timeouts, the passing of time during the execution of processes in the network, and the execution of time-critical events.
2. *A model of concurrency*, which defines the atomicity, parallelism, and fairness of events executed by different processes in the network.
3. *A model of failure*, which defines what types of failures can occur and when those failures may occur.

To achieve abstraction, however, these notations often need to make unrealistic assumptions about these three models:

1. It is impractical to implement a conceptually manageable model of concurrency such as strict interleaving of actions, which requires all processes to agree on the action to execute before any action can be executed.
2. It is expensive to implement any abstract model of time, since closely synchronizing processes requires many messages to be exchanged, using resources that are then not available to the network.
3. It is impossible to ensure that failures are restricted to those allowed by an abstract model of failure.

As a result, a faithful implementation of a specification in an abstract notation is either impractical, expensive, or impossible.

In this dissertation, we take the view that the implementation should not be strictly faithful to the abstract specification, therefore. Rather, it should be an *approximation* of that specification. The goal of this approximation is to come as close as possible to the properties of the protocol, although they may not be achievable in a strict sense. The key to this approximation is to violate the assumptions upon which the abstract notation is based in a manner that verifiably preserves the properties of the abstract specification in the implementation.

In the next section, we present AP, the abstract notation used as a basis for this work, as well as the specific assumptions that are violated while implementing it. Subsequent sections introduce timed AP and APC, models of execution which are successively easier to implement, and demonstrate that the APC model preserves safety, progress, and fairness properties of timed AP for a restricted class of specifications. Finally, we discusses experiments and extensions.

3 AP

A network protocol specified in AP consists of a number of processes connected by channels. The processes in a network communicate with each other by sending and receiving messages over the channels.

A process in AP has the following form:

process {process name}
const {constant name} :{type} , ...
var {variable name} :{type} , ...
begin {action} ⫿ ... ⫿ {action} **end**

Constants are global—two constants in different processes with the same name have the same value. Variables are local to each process. Constants and variables are of the following types: boolean, integer, range of integers, and array.

Each action consists of a guard and a sequence of one or more statements. There are three types of guards:

- *Local guards* are predicates which refer only to local variables and constants.
- *Receive guards* are of the form

  rcv {msg} **from** {process name}

  and are enabled only when a message matching that specified in the guard is at the head of the channel between the process named in the guard and this process. When executed, the receive guard removes the message from the channel.

- *Timeout guards* are of the form:

  timeout {global predicate}

  A global predicate can refer to local variables and constants, the variables and constants in other processes, and the contents of the channels between any two processes.

Each statement in the sequence of statements in an action is one of the following:

- **skip**, which does nothing.
- Assignment.
- Sending a message, which inserts the message into the channel between this process and the one specified in the statement.
- Selection, which is of the form:

  if {predicate} $\rightarrow$ {sequence of statements}
  $[\!]$ ... $[\!]$ {predicate} $\rightarrow$ {sequence of statements}
  fi

- Iteration, which is of the form:

  do {predicate} $\rightarrow$ {sequence of statements} **od**

In AP, the following three types of faults are allowed:

- Message loss.
- Message reordering.
- Message corruption, which replaces the message in the channel with the special message, *error.*

The execution of a network of processes is based on the following five assumptions (that make protocol implementation difficult and expensive):

1. *Abstract timeout:* The predicate of a timeout guard can include the variables and constants from any process and the contents of any channel.
2. *Action atomicity:* Enabled actions in a network are executed one at a time.
3. *Action fairness:* If any action is continuously enabled, then that action will eventually be executed.
4. *Immediate message propagation:* When a message is sent, the action to receive it is immediately enabled.

5. *Fault atomicity:* Faults occur one at a time, and are not simultaneous with any action.

An example of a protocol specified in AP involving two processes, p and q, is as follows:

```
**process** $p$
**var** $\mathit{readyp} : \textbf{boolean}$
**begin**
   $\mathit{readyp} \rightarrow \textbf{send } \mathit{rqst} \textbf{ to } q;\ \mathit{readyp} := \textbf{false}$
 $[\!]$ $\textbf{rcv } \mathit{rqst} \textbf{ from } q \rightarrow \textbf{send } \mathit{rply} \textbf{ to } q$
 $[\!]$ $\textbf{rcv } \mathit{rply} \textbf{ from } q \rightarrow \mathit{readyp} := \textbf{true}$
 $[\!]$ $\textbf{timeout } \neg \mathit{readyp} \wedge (\mathit{rqst}\#ch.p.q + \mathit{rply}\#ch.q.p = 0) \rightarrow \textbf{send } \mathit{rqst} \textbf{ to } q$
 $[\!]$ $\textbf{rcv } \mathit{error} \textbf{ from } q \rightarrow \textbf{skip}$
**end**
```

Process q is the same, except that p is replaced by q and q by p.

In this protocol, whenever process p is ready, it sends a request to q and becomes unready. When p receives a reply from q, it becomes ready again. If p receives a request from q, it sends a reply. If p has sent a request and is waiting for the reply, and either the request or the reply is lost, the request is resent. Finally, if p receives a corrupted message from q, it removes it from the channel.

4 Timed AP

Because abstract timeouts are very difficult to implement, we introduce a new model called *timed AP*. Timed AP is exactly the same as AP except that the global predicates in AP are replaced by timed predicates. (In the dissertation, we plan on developing an algorithm for replacing global predicates by timed predicates while maintaining all the safety and progress properties of the specified protocol.)

In timed AP, each process has a real-time clock. The clock in each process may not be synchronized with the clock in any other process, but every clock advances at the same rate.

The execution of an action or fault in timed AP takes no time. The clocks advance by the execution of an implicit action. Messages cannot be received in the same instant that they are sent, and the time between sending a message and receiving that message is the *propagation delay* of the message.

In timed AP, a timeout guard is of the form:

timeout $G \wedge L$ **since** $\{c_0, ..., c_n\}$

where G is a local predicate of the process, L is a time delay which is strictly larger than zero, and $c_0, ..., c_n$ identify other actions in the same process. We require the guard of each timeout action satisfy the condition that no action in the process, other than those identified in the **since** predicate, may change G from false to true.

Each timeout action introduces a time variable which is zero in the network's initial state and which increases with the process's real-time clock. The variable is reset to zero whenever the timeout action or one of the actions $c_0, ..., c_n$ is executed.

While executing a network specified in timed AP, if at some state we discover that more time than L has passed since the initial state or the last execution of the timeout action or one of the actions specified in the timeout action's guard, and the local predicate G is true, then the guard is true and the timeout action can be executed.

To continue the example above, the protocol involving two processes, p and q, in timed AP is as follows:

```
process $p$
var $readyp$ : boolean
begin
  $A$ : $readyp \rightarrow$ send $rqst$ to $q$; $readyp :=$ false
  [] rcv $rqst$ from $q \rightarrow$ send $rply$ to $q$
  [] rcv $rply$ from $q \rightarrow readyp :=$ true
  [] timeout $\neg readyp \wedge 2D$ since $\{A\} \rightarrow$ send $rqst$ to $q$
  [] rcv $error$ from $q \rightarrow$ skip
end
```

where D is the upper bound on the propagation delay. Again, process q is the same, except that p is replaced by q and q by p.

These processes are identical to those in Section 3, except that the global predicate detecting that a message has been lost is replaced by a timed predicate which uses a delay to detect that a message has been lost. Since action execution takes no time, the maximum length of time between sending a request and receiving a reply without a message loss occurring is $2D$.

5 APC

To address the remaining four assumptions from Section 3, we introduce a model of execution of timed AP specifications called *APC.*

The syntax used to specify protocols in APC is the same as that of timed AP. However, the assumptions made by APC are significantly different.

1. *Local action atomicity:* In APC, the execution of an action cannot be interleaved with the execution of any action from the same process. However, the execution of an action from a process can be interleaved with the execution of an action from any other process.

   More specifically, the execution of an action in APC consists of a sequence of events. There are nine types of events:

(a) Detecting that a local or timed guard is true.
(b) Detecting a receive guard is true and receiving the specified message.
(c) Executing a **skip** statement.
(d) Executing an assignment statement.
(e) Sending a message.
(f) Testing that a predicate is true in an **if** statement.
(g) Testing that a predicate is true or false in a **do** statement.
(h) Transmitting a message (as discussed below in assumption 1i).
(i) Fault occurrence.

These nine types of events can be partitioned into three classes:

- Internal events are of types a, b, c, d, e, f, and g.
- External events are of types h and i.
- Message events are of types b, e, h, and i.

Events in APC can be executed simultaneously except for the following restrictions:

(a) The internal events of an action execution cannot be interleaved with the internal events of another action execution of the same process—therefore they cannot be simultaneous.
(b) No two message events on the same message can be simultaneous.
(c) No two transmission events (i.e. events of type h) can be simultaneous on messages in the same channel.

2. *Weak action fairness:* If a process has only one continuously enabled action, then that action will eventually be executed.

3. *Delayed message propagation:* Each channel consists of two stages. When a message is sent, it enters the first stage in which it is not visible to the receiving process. As a separate event, a message is transmitted from the first stage to the second stage, where it is visible to the receiving process when it advances to the head of the channel.

4. *Weak fault atomicity:* Faults involving a particular message occur one at a time and they do not occur before or while the message is being transmitted nor while it is being received.

The execution of the first action from the example of Section 4 is the following sequence of events:

1. Detecting that *readyp* is true.

2. Sending *rqst* to process q.

3. Assigning **false** to *readyp*.
4. Transmitting the *rqst* message.

The third and fourth events can occur in either order and all of these events can be interleaved or simultaneous with events from the first, third, fourth, and fifth actions of process q.

6 Correctness Preservation

In order to show that a protocol specified in timed AP is correctly implemented in APC, we present the following two theorems.

6.1 Theorem of safety and progress preservation

A *computation* of a protocol in APC is a sequence of events. A *complete* computation is a computation in which if any of the events in the execution of an action are part of the computation, then all of the events in the execution of that action are part of the computation.

Let P be a protocol specified in APC and M be an implementation of P using APC. Then, for every complete computation c_M of M, there is a computation c_P of P such that the following two conditions hold:

1. The state of P at the end of c_P is equivalent to the state of M at the end of c_M.
2. The sequence of actions and faults executed in c_P is equivalent to the sequence of actions and faults executed in c_M.

A proof of this theorem can be sketched as follows. Condition 1 can be seen by performing the following transformations on c_M:

1. Serialize the computation.

   Two events in a computation are *independent* iff the variables and messages involved in each event are disjoint. A step consisting of n simultaneous events can be serialized into n steps consisting of single events if the events are independent.

   In APC, simultaneous events are independent:

   (a) Simultaneous internal events belong to different processes and therefore do not involve the same set of variables.

   (b) Simultaneous external events involve separate messages and are therefore independent.

   (c) Simultaneous internal and external events are independent since external events do not involve variables and no two message events on the same message can be simultaneous.

The state of M at the end of c_M is unchanged by serialization.

2. Reorder the result of the first transformation so that if the first event of the execution of an action A occurs before the first event of an action B before reordering, then all events of the execution of action A occur before any of the events of the execution of action B after reordering, and any fault events are not interleaved with the events of the execution of any action.

   To ensure these conditions, it is necessary to move any event a (which is not the first event of an action execution) from the execution of action A which is interleaved with events from the execution of action B or with fault events to position before any of these events. This can be done by swapping event a and each event b, which can be done if a and b are independent.

   - If a is an internal event,
     - and b is an internal event, then they are from different processes and are therefore independent.
     - and b is a transmission event, then a cannot involve the message being transmitted and therefore a and b are independent.
     - and b is a fault event, then a cannot involve the message involved in the fault and therefore a and b are independent.
   - If a is a transmission event,
     - and b is an internal event, then b cannot be from a process that has received the message transmitted by a and a and b are independent.
     - and b is an transmission event, then b must be on a different channel and a and b are independent.
     - and b is a fault event, then a and b must be independent because faults do not occur before transmission.

   Again, the state of M at the end of c_M is unchanged.

3. Collect each sequence of action events into an action execution and remove the unnecessary transmission events.

The resulting computation is equivalent to a computation c_P.

Condition 2 is true after c_M is serialized and reordered as described above. However, the reordering transformation described above does not change the relative order of the first event of each action execution.

6.2 Theorem of fairness preservation

In order for a protocol executing under the weaker fairness assumption of APC to implement the stronger fairness assumption of timed AP, there must be an upper bound on the number of actions that can be executed before any action that is continuously enabled is executed.

In order to satisfy this requirement, we make the assumption that there is a bound on the number of messages in any channel at every state of the execution.

Additionally, all processes are *timid:*

1. The execution of every local action disables that action.

2. No local action execution enables any other local action.

Under these conditions, any action that is continuously enabled will eventually be executed.

7 Experiments

We have implemented a library of C functions based on APC. This library enables the hand-implementation of timed AP processes that communicate using UDP messages.

To use the library, each action is divided into two functions; one implementing the guard and one implementing the statements of the action. The key to the library is the observation that, for most protocols, a process spends most of its time either waiting for a message to be received or for a timeout to expire. When a message arrives, the guards of the receive actions can be tested against it and the first guard returning true can cause the associated statements to be executed. Similarly, when a timeout expires, the function implementing the guard can be tested and, if it returns true, the function implementing the statements of the timeout action can be executed. At this point, the local state of the process may have changed, and the guard functions for the local actions can be tested and the statements of any local actions which are enabled can be executed until all of the local guards have returned false. In this state, no further actions will be enabled until another message is received or until a timeout expires.

The APC library also provides facilities to execute the processes on remote machines and to allow the processes to locate each other in order to exchange messages.

Using APC requires that each AP specification be translated into C by hand, using the functions in the APC library:

- `initialize_engine`

- `add_receive_action`

- `add_local_action`

- `add_timeout_action`

- `send_message`

- `engine`

The resulting C code is compiled, linked with the library, and executed.

There are a number of issues involved in correctly implementing network protocols which are not addressed by the formal framework presented earlier. These issues are:

1. Since it is very difficult to verify the correctness of the APC library against its model discussed in Section 5, correctness of the APC library needs to be validated experimentally, by implementing a number of protocols and checking the behavior of those protocols against the expected behavior of the protocol in timed AP.

2. As described above, there are a number of restrictions on protocols to be implemented correctly. In order to describe the practical boundaries defined by these restrictions, it is necessary to implement protocols which test those boundaries and examine any difficulties found.

3. In order to compare the relative merits of this approach, it is necessary to implement well-known protocols in to compare these implementations with previous implementations of those protocols, according to the following factors:

    - The time and effort required to implement the protocol.
    - The size and complexity of the implementation.

Since the current implementation of the APC library is based on UDP, the protocols chosen for the experimental work are limited to those based on UDP. These protocols are:

- The heartbeat protocol, which is discussed in more detail below, exercises all of the facilities of APC while displaying simple behavior, and is therefore ideal for validating the correctness of the APC library.

- The secret exchange protocol[8] is a secure protocol, and needs to be verified as well as implemented correctly.

- The real-time transport protocol[11] is a well-known protocol which has been implemented independently, and can therefore be used for comparison.

- The network time protocol[14] uses message timings to synchronize clocks between processes, which may be difficult to implement in timed AP as described in Section 4.

8 Possible Extensions

In this section, we present two extensions which address some of the other issues involved in correctly implementing network protocols.

8.1 Protocol termination

For abstract protocols, it is customary to assume that each protocol execution is infinite. On the other hand, running protocols have finite lifetimes and need to be terminated in an orderly way. The processes of the protocol must be able to agree that the protocol is terminated, in the presence of message errors.

We identified a class of heartbeat protocols that efficiently ensure that if a process terminates or fails then the remaining processes also terminate, while tolerating message loss, as described in [10]. These protocols provide a compromise between the delay of detecting termination, the frequency of beat messages sent to report the status of processes, and the probability of a false detection of termination due to message loss. This class of protocols allow those parameters to be adjusted to application requirements. Another protocol can be integrated with the accelerated heartbeat, providing the resulting protocol with a well-defined termination scheme.

There are two ways to integrate the accelerated heartbeat with another protocol:

1. Compose the accelerated heartbeat with the other protocol, modifying the actions of both.

2. Redefine the send and receive primitives to become alert to termination by using the accelerated heartbeat internally. For a send primitive, either the message is sent to the other process or an indication is returned that the other process has terminated. For a receive primitive, either the process receives a message, the process receives an indication that the other process has terminated, or the process continues to wait.

   The send primitive of TCP is alert in the above sense, but the receive primitive is not. [9] describes a number of functions which can be used in place of the TCP receive primitive but which are alert.

The models described in Sections 4 and 5 need to be extended in order to specify protocols with finite lifetimes and to take advantage of the heartbeat protocols.

8.2 Protocol modularity

When specifying a protocol, it is useful to separate it from other protocols which it relies on, or which rely on it. This allows the protocol to be understood in isolation and verified based on assumptions about the correctness of the other protocols.

One example of this modularity is the accelerated heartbeat protocol described above, independently from any specific protocol which will apply it.

When implementing network protocols, however, they must be joined together. There are two techniques to do this:

1. To connect the processes of a protocol, so that a message sent by one process using the facilities of another protocol is first sent to a process

implementing the second protocol. This technique joins the protocols into hierarchical layers.

2. To compose the protocols together, modifying them as necessary. This approach has an advantage in terms of efficiency[6, 7]. On the other hand, it rapidly increases the complexity of the composite protocol.

Since the APC library is based on the architecture of a single process communicating by means of UDP messages, it does not currently support the first technique. On the other hand, one possibility for supporting the second technique is to automatically compose AP processes, which is also not supported by the APC library. In either case, care must be taken to preserve the properties of the individual protocols and to verify that these properties have been preserved.

References

[1] Mark B. Abbot and Larry L. Peterson. A language-based approach to protocol implementation. *IEEE/ACM Transactions on Networking*, 1(1), September 1993.

[2] A. Basu, G. Morrisett, and T. von Eicken. Promela++: A language for constructing correct and efficient protocols. In *Proceedings of the Conference on Computer Communications (IEEE Infocom)*, San Francisco, CA, March/April 1998.

[3] Gérard Berry. The foundations of Esterel. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 1998.

[4] R. T. Braden. RFC 1122: Requirements for internet hosts - communication layers, October 1989.

[5] Satish Chandra, James R. Larus, Michael Dahlin, Bradley Richards, Randolph Y. Wang, and Thomas E. Anderson. Experience with a language for writing coherence protocols. In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*, Berkeley, CA, October 1997. USENIX Association.

[6] D. D. Clark. RFC 817: Modularity and efficiency in protocol implementation, July 1982.

[7] David D. Clark and David L. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *SIGCOMM Symposium on Communications Architectures and Protocols*, Philadelphia, PA, September 1990. ACM.

[8] Mohamed G. Gouda, M. El-Nozahy, C.-T. Huang, and Tommy M. McGuire. Hop integrity in computer networks. In preparation.

[9] Mohamed G. Gouda and Tommy M. McGuire. Alert communication primitives in TCP. Submitted to *Software: Practice and Experience*.

[10] Mohamed G. Gouda and Tommy M. McGuire. The accelerated heartbeat protocols. In *18th IEEE International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998. IEEE Computer Society.

[11] Audio-Video Transport Working Group, H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RFC 1889: RTP: A transport protocol for real-time applications, January 1996.

[12] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[13] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A readable TCP in the Prolac protocol language. In *Proceedings of SIGCOMM '99*, Cambridge, MA, August 1999. Association for Computing Machinery.

[14] David L. Mills. RFC 1305: Network time protocol (version 3), March 1992.

[15] Vern Paxson. Automated packet trace analysis of TCP implementations. In *Proceedings of SIGCOMM '97*, Cannes, France, June 1997. Association for Computing Machinery.

[16] J. Postel. RFC 793: Transmission control protocol, September 1981.

[17] Kenneth J. Turner. *Using Formal Description Techniques: An Introduction to Estelle, LOTOS, and SDL*. John Wiley & Sons, 1993.

[18] Gary R. Wright and W. Richard Stevens. *TCP/IP Illustrated*, volume 2. Addison Wesley, 1995.