

Practical Considerations in Protocol Verification: The E-2C Case Study

Yifei Dong, Scott A. Smolka, Eugene W. Stark
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400, USA
{ydong, sas, stark}@cs.sunysb.edu

Stephanie M. White
Computer Science and Management Engineering Department
Long Island University
Brookville, NY 11548-1300, USA
Stephanie.White@liu.edu

Abstract

We report on our efforts to formally specify and verify a new protocol of the E-2C Hawkeye Early Warning Aircraft. The protocol, which is currently in test at Northrop Grumman, supports communication between a Mission Computer (MC) and three or more Tactical Workstations (TWSs), connected by a single-segment LAN. We modeled the protocol in the PROMELA specification language of the SPIN verification tool, and used SPIN to analyze a number of properties of the protocol. Our investigation revealed a race condition that can lead to a disconnect of an MC/TWS connection when there is one lost UDP datagram and significant timing delays. Such delays are virtually impossible under normal E-2C operating conditions, but could be due to noise on the MC/TWS LAN. A simple modification was proposed that avoids the disconnect in many situations. Practical considerations, however, mandated that the protocol be left as is: shutting down a noisy connection and reinitializing the TWS, with minimal delay and loss of information to the operator, was deemed preferable to operating in degraded mode.

1. Introduction

The E-2C Hawkeye is an all-weather, airborne, early warning/command and control aircraft developed by the Northrop Grumman Corporation [NGC98]. The Hawkeye can monitor 6 million cubic miles of air space and more than 150,000 square miles of ocean surface for the presence of aircraft, missiles, ships, and fixed targets. For example,

an E-2C over New York City could track all the traffic in the congested Boston-to-Washington air corridor.

The Hawkeye's Cooperative Engagement Capability (CEC), mission computer, and sensors enable the Hawkeye to serve as the U.S. fleet's information hub, fusing information from sources such as satellite and shipborne and airborne radar, and then distributing that information to those who need it. The system is configured as a client/server architecture, with the Mission Computer (MC) acting as the server and (currently) three Tactical Workstations (TWSs) as the clients.

The communications protocol underlying the system (the E-2C Display LAN protocol [NGC96]) runs on a single-segment local area network, and sits above TCP/IP and UDP/IP. All command and acknowledgement packets are transmitted by TCP/IP, which is reliable but has high overhead. Data packets are transmitted by UDP/IP, which is less reliable but has low overhead. The core of the protocol is its acknowledgement and retransmission scheme, and the protocol is designed to ensure that all functioning TWSs have the correct data. We shall henceforth refer to the protocol as the MC/TWS protocol.

In December 1996 the Northrop Grumman E-2C Project asked Stony Brook Professors Smolka and Stark to formally specify and verify the MC/TWS protocol. Since that time, Stony Brook personnel Dong, Smolka, and Stark met four times with Northrop Grumman personnel Newman, Sandler, and White, and also communicated via phone and email. It was agreed that Stony Brook would attempt to formally model the protocol and verify certain key properties using a computer-aided verification tool. To this end, Stony Brook wrote an abstract, finite-state specification of the protocol in PROMELA, the input language of the AT&T

Bell Labs SPIN verification tool [Hol97], and used SPIN to check several assertions and linear temporal logic (LTL) formulas against the specification. The verification was particularly challenging since the state space of the PROMELA model of the protocol contains approximately 289 million global states, despite the use of several simplifying assumptions and sophisticated state-space abstractions.

One of the main outcomes of our E-2C protocol analysis was the discovery of a race condition that can lead to a brief disconnect of a control connection between the MC and one of the TWSs, despite the fact that the TWS successfully receives all UDP data packets sent by the MC. The conditions under which this race condition can occur involve the loss of one UDP data packet, and significant timing delays. Such delays are virtually impossible under normal E-2C operating conditions, but might be caused by noise on the MC/TWS LAN.

Based on our finite-state analysis, we proposed a simple protocol modification that avoids the disconnect in many situations. The modification involves directing the MC to examine a retransmission field in the acknowledgement packet, so that it can distinguish old acknowledgements from new ones. It was ultimately decided, however, to leave the protocol as is. This is because the original protocol would shut down a noisy connection and the TWS would be reinitialized with minimal delay and loss of information to the operator. This is preferable to operating in degraded mode as it is critical that E-2C crew see accurate, up-to-date data. We view our experience with the MC/TWS protocol as one instance where, even though it was interesting and informative to learn about the race condition, practical considerations take precedence over what is suggested by findings due to formal verification.

In terms of related work, the recent literature is replete with verification efforts targeting communication protocols and other similar industrial applications. See [CW96] for a survey and [Bae90, GR97] for collections of case studies. The MC/TWS protocol differs from these primarily in the role it plays in an airborne early warning/command and control aircraft, and in the collateral requirements such as application demands.

The structure of the rest of this paper is as follows. Section 2 describes the SPIN verification tool. Section 3 gives a detailed description of the MC/TWS protocol while Section 4 explains how we modeled the protocol in PROMELA. Section 5 summarizes our verification results and Section 6 concludes.

2. SPIN

SPIN [Hol97] is a model checker for asynchronous systems specified in the language PROMELA. Safety and liveness properties are formulated using linear-time temporal

logic (LTL), and model checking is carried out using the automata-theoretic approach; i.e., by checking the non-emptiness of the intersection of Büchi automata (automata over infinite words), one for the system model and one for the formula.

Model checking in SPIN is performed on-the-fly, meaning that the state space of the model intersected with the formula automaton need only be built up to the point where the non-emptiness of the resulting automaton can be proven. SPIN also uses partial-order reduction to avoid exploring redundant computation paths in the model due to interleaved executions of independent events. The computation of the intersection can either be done in a conventional exhaustive manner, or, when this proves to be impossible due to state explosion, with an efficient approximation method based on bitstate hashing. With a careful choice of hashing functions, the probability of an exhaustive proof remains very high.

Besides being able to specify correctness properties in LTL, the PROMELA specification language includes two types of labels that can be used to define two complementary types of liveness properties: acceptance and progress. In the syntax of LTL, an acceptance property corresponds to a formula of the type $\Box\Diamond p$ (infinitely often p), where p is a user-defined accepting state. The violation of a progress property corresponds to a formula of the form $\Diamond\Box\neg p$ (almost always p), where p is a user-defined progress state. SPIN also offers an assertion-checking facility, which we used to determine if there were any unwanted terminations of an MC/TWS connection (see Section 5).

PROMELA is a nondeterministic guarded-command language with influences from Hoare's CSP and the language C. PROMELA includes support for data structures, interrupts, the dynamic creation of concurrent processes, and a variety of synchronous and asynchronous message passing primitives. Message passing is via channels with arbitrary numbers of message parameters.

3. Description of the MC/TWS Protocol

The E-2C MC/TWS protocol is a client/server data distribution protocol. The mission computer (MC) acts as the server, and the tactical workstations (TWSs) as the clients. The MC and the TWSs are connected by two types of channels: (1) TCP channels, which are used to transmit command and acknowledgement packets, are point-to-point and bidirectional; (2) UDP channels, which are used to broadcast data packets (called UDP transfer buffers, or UTBs) from the MC to the TWSs, are multicast and unidirectional. All channels can sustain delay. TCP channels are reliable in that they guarantee that data is received, but have higher overhead. UDP channels have lower overhead but do not guarantee receipt of data. There is no communication between TWSs within the protocol.

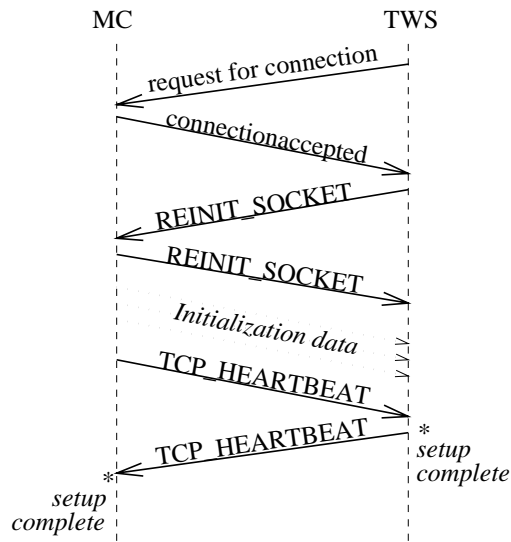


Figure 1. Setup TCP connection.

The MC/TWS protocol is structured in terms of three kinds of transactions:

Setup connection Each MC/TWS TCP connection is initiated by the TWS. Setup is realized using what essentially amounts to a three-phase handshake protocol, and is illustrated in Figure 1: (a) The TWS sends a request for connection to the MC, and waits for a positive response; (b) The TWS sends a REINIT_SOCKET packet to the MC, and the MC replies with REINIT_SOCKET; (c) The MC sends initialization data to the TWS with TCP_HEARTBEAT as the end-of-data mark; the TWS reads the data, after which it sends back TCP_HEARTBEAT.

No setup handshaking is needed for the UDP connections. The MC broadcasts blindly but keeps track of the IP address of each TWS. After an MC/TWS TCP connection has been setup, the TWS will listen to the line and begin to process incoming UTB data.

Heartbeat The MC periodically transmits a TCP_HEARTBEAT buffer to all connected TWSs every HEARTBEAT_INTERVAL seconds. Upon receipt of a TCP_HEARTBEAT buffer, an operational TWS will respond with a TCP_HEARTBEAT buffer within 20 ms. The function of the heartbeats is to monitor the TCP connection. If the server or client does not receive this periodic TCP_HEARTBEAT within a reasonable timeout period ($2 \cdot \text{HEARTBEAT_INTERVAL}$ seconds), it will assume that the connection has failed.

Data transmission This is the core of the protocol. The MC broadcasts UTB data via UDP. In each UTB, there

is a sequence number that is consecutive to the sequence number of the previous UTB. The MC will request an acknowledgment response from each TWS after MAX_SEQ UTBs have been transmitted, where MAX_SEQ is a parameter of the protocol. The request is actually a REQUEST_FOR_ACKNOWLEDGE_RESPONSE flag in the last UTB.

Upon receipt of a UTB, a TWS will check the UTB's sequence number and compare it to the sequence number of the previously received UTB. If it is consecutive, the TWS will record the last successfully received sequence number as the current one; otherwise it stops processing incoming UTBs. If the UTB-resident REQUEST_FOR_ACKNOWLEDGE_RESPONSE flag is set to OK, the TWS will send an acknowledgment with the last successfully received sequence number to the MC through TCP connection.

If the MC does not receive an acknowledgment from every TWS after 30 ms, a timeout will occur and the MC will resend the acknowledgment request and restart the timer. After the second timeout, the MC will disconnect the non-responsive TWSs.

Having collected acknowledgement from the TWSs, the MC calculates the minimum sequence number received over all the TWSs. If all TWSs have received all UTBs, the MC will go on to the next round of data transmission; otherwise, it will retransmit the UTBs from the minimum sequence number plus 1 to MAX_SEQ. After retransmission, the MC will again collect acknowledgments and disconnect those TWSs that do not respond or still cannot receive all the data.

The MC disconnects a TWS by sending it CLOSE_SOCKET. The TWS will try to reestablish the connection with the MC after disconnection.

Figures 2 and 3 illustrate the data transmission protocol in the case of successful UTB transmission, without and with retransmission, respectively. In both figures, $N = 10$ and there are three fields in a UTB header: *sequence number*, *request-for-acknowledgement flag*, and *retransmission flag*. The acknowledgement contains two fields: *last consecutive received UTB sequence number* and *retransmission flag in last-received UTB buffer*. The scenario depicted in Figure 3 involves TWS₂ dropping UTB number 8, and TWS₃ dropping UTB number 9, so their last consecutive message sequence numbers are 7 and 8, respectively. The MC thus rebroadcasts UTB data from UTB number 8.

The three types of transactions described above are inter-related in various ways. For example, TCP_HEARTBEAT is only sent to connected TWSs, and the MC only expects acknowledgments from connected TWSs during UTB transmission. Also, the MC only sends TCP_HEARTBEAT after the current UTB sequence is completed, and a TWS attempts a connection reinitialization only after the MC de-

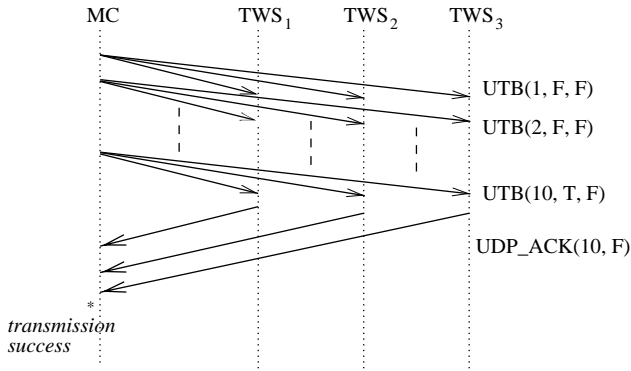


Figure 2. A successful UTB transmission sequence without retransmission.

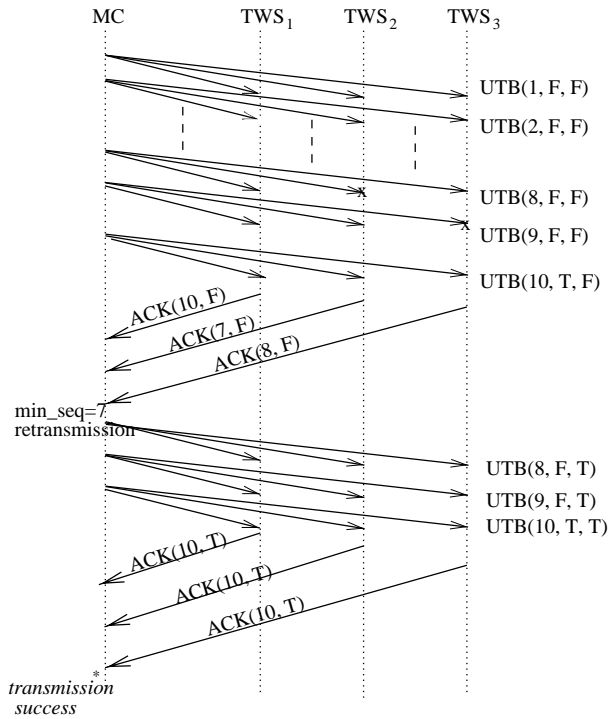


Figure 3. A successful UTB transmission sequence with retransmission.

tests a transmission error and sends a DISCONNECT to the TWS.

4. Specifying the Protocol in PROMELA

PROMELA, SPIN’s input language, is a fairly low-level specification language. As such, a rendering of the E-2C protocol in PROMELA required some indirect modeling, which we now describe.

4.1. TCP and UDP Channels

As discussed in Section 3, the MC/TWS protocol uses two types of channels. TCP channels are point-to-point and bidirectional, and are modeled by two arrays of unidirectional channels, one from the MC to the TWSs, the other from the TWSs to the MC. The MC can communicate on all of these channels, but each TWS can only send messages on one outgoing channel and receive messages from one incoming channel.

UDP channels, which are broadcast, unidirectional and admit the possibility of message loss, are modeled as an array of channels, each one from the MC to a TWS. Broadcast is captured by having the MC send the same message on all the channels. In the model, a TWS will receive all messages broadcast by the MC, but will nondeterministically decide to drop or process these messages. This achieves the desired effect of a broadcast channel that may potentially drop messages.

A TWS not possessing a connection with the MC ignores all UTB messages it receives from the MC. In the actual protocol, this is a consequence of the fact that an unconnected TWS does not listen to its UDP channel and the signal goes away. In PROMELA, we cannot set the channel to null but rather use a “dummy” receive statement that simply ignores the messages received.

4.2. Timeout

There is no facility in PROMELA for explicit real-time specifications. There is a `timeout` construct but this is actually a system-wide `else` statement that is executed when no other statement is executable. We therefore modeled timeout by placing an additional nondeterministic choice construct anywhere in the code where timeout is possible.

4.3. State Space Abstraction

State space abstractions are used to decrease the number of global states that need to be explored in order to verify the system under investigation. In the case of the MW/TWS protocol, the main state space abstraction we employed involved the modeling of a UTB transmission cycle between

the MC and the TWSs. The key observations here are: (i) the properties we are interested in verifying, such as eventual message delivery, absence of deadlock, and no unnecessary disconnects, are *data independent* [Wol86], in the sense that they do not depend on the actual data contained in the messages transmitted, but rather just on whether or not the messages are transmitted at all; (ii) when a UTB sent to a TWS is lost, the TWS ignores all subsequent messages in the current transmission cycle.

Given these characteristics of the protocol, instead of modeling a UTB transmission cycle as a sequence of MAX_SEQ UDP data transmissions, we model it as *one* UDP transmission with two fields: the sequence number of the first UTB buffer in the transmission cycle and the retransmission flag. Therefore a first attempt by the MC to broadcast a sequence of UTBs to the TWSs is captured by setting the first field to one and the second to zero. The same type of message is also broadcast by the MC to request acknowledgements from the TWSs. In this case, the first field is set to zero (not a valid UTB sequence number), and the value of the second field is unused.

Each TWS, upon receiving a transmission-sequence-modeling message, nondeterministically chooses an integer between one and MAX_SEQ, representing the sequence number of the last consecutively received UTB. If all UTBs are not received by all TWSs (i.e., some TWS selected an integer strictly less than MAX_SEQ), the MC will broadcast a message in which the first field is set to the minimum of the integers selected by the TWSs, plus one, and in which the second field (the retransmission flag) is set to one.

Upon receiving a request-for-acknowledgement message from the MC, a TWS nondeterministically chooses a number that is zero or one. This reflects whether or not the acknowledgement request was dropped by the TWS.

The use of sequence-modeling messages significantly reduces the size of the PROMELA model of the protocol. There are two reasons for this. First, fewer messages are exchanged between the MC and TWSs, and, secondly, it eliminates the need to explicitly model the behavior of a TWS in response to receiving a UTB subsequent to the drop of a UTB during the same transmission cycle.

Moreover, we were further able to reduce the size of the state space by using PROMELA's `atomic` construct to atomically execute the section of code in a TWS that is responsible for deciding which, if any, UTB is dropped in a transmission cycle, and if the request for acknowledgement is dropped. PROMELA's `atomic` construct can be used to bracket a section of code that is to be executed atomically, i.e., without interleaved execution with code from other processes in the system. Since the code sections we bracketed represent internal computation of the protocol, the `atomic` construct could be safely used in these instances.

5. Summary of Verification Results

We set out to verify the following properties about the MC/TWS protocol:

- Can a TCP connection between the MC and a TWS always be established?
(property: *successful connect*)
- Does the protocol contain a deadlock or livelock?
(properties: *no deadlock* and *no livelock*)
- Does a TWS eventually receive all UTB buffers sent by the MC?
(property: *eventual delivery*)
- Can an MC/TWS connection be terminated when a single UTB is lost?
(property: *disconnect one* and *modified protocol*)

The names associated with these properties correspond to the various rows of Table 1, where our model checking results are summarized. The results were obtained using version 2.9.7 of the SPIN model checker on an SGI IP25 Challenge machine with 16 MIPS R10000 processors and 3GB of main memory. Each execution of SPIN, however, was carried out on a single processor with 1.9GB of available main memory.

Moreover, the following parameter settings were used on all runs: NTWS, the number of TWSs, was set to two, and MAX_SEQ, the number of UTBs broadcast by the MC before requesting acknowledgement, was set to three. Finally, to further reduce the state space size, it was assumed that the MC does not transmit heartbeats during a UTB transmission phase, and vice versa.

Besides LTL model checking, three other verification techniques were used: non-progress loop checking (for livelock), invalid endstate checking (for deadlock), and explicit assertion checking (for disconnect on loss of a single message). Moreover, the partial-order reduction, bitstate hashing, and weak fairness options of SPIN (see [Hol97]) were employed in all cases.

For successful connection establishment, we model checked the following LTL formula:

```
<>(connected(0) && connected(1))
```

where `connected(i)` is true when TWS_{*i*} has a TCP connection to the MC, `<>` is the LTL operator meaning “eventually”, and `&&` is SPIN syntax for conjunction. SPIN reported that this formula is true of our model of the protocol and thus a TCP connection between a TWS and the MC can always be eventually established. (Actually, the LTL formula states something stronger than this, namely: It is always the case that a state is eventually reached in which

Property	states	transitions	memory (MB)	time (hrs:mins:secs)
successful connect	289M	848M	708	7:37:14
no deadlock	14M	42.7M	300	0:15:33
no livelock	186M	746M	709	6:51:01
eventual delivery	13.4M	36.3M	695	0:21:23
disconnect one	140	190	1.25	0:0:06
modified protocol	17.6M	57.6M	300	0:21:24

Table 1. Summary of verification results.

all TWSs are connected.) This was expected as the TWS attempts to establish a connection with the MC once it begins execution and whenever its connection is closed.

To check for the possibility of deadlock in the protocol, we used SPIN’s facility for checking for the occurrence of invalid endstates (a terminal state representing abnormal termination). Deadlock was detected at first, but after closer examination, we found it was due to use of a too small buffer size in the modeling of TCP channels. After increasing the buffer size to 2 for each TCP channel (still a relatively small buffer size for actual TCP channels), the protocol was found to be deadlock-free. The “no deadlock” entry in Table 1 is for the deadlock-free version of the protocol.

Livelock was checked using the non-progress cycle detection algorithm of SPIN. Each transmission of a UTB message was designated a progress state. The verifier detected the existence of a non-progress cycle, which means that the protocol can livelock. However, using SPIN’s trace mechanism, we determined that this cycle involved the circular exchange of TCP_HEARTBEAT messages between the MC and a TWS. After disabling the heartbeat transitions, no other livelock was found; this is the run reported as entry “no livelock” in Table 1.

LTL model checking was also employed to determine whether or not a TWS eventually receives all UTBs. The following LTL formula was used for this purpose:

```
<>(tws_recv_all(0) &&
tws_recv_all(1))
```

where `tws_recv_all(i)` means that TWS_i has received all UTBs broadcast by the MC. SPIN reported this formula to be false, and this is because the protocol only attempts retransmission once in the presence of transmission errors. Therefore, the protocol cannot tolerate the loss of messages in both the first and second phases of a UTB transmission cycle (with the second phase being the retransmission phase). The E-2 Program designed the protocol in this way so that E-2 operators are assured that data on their displays is always up-to-date and reliable. The entry entitled “eventual delivery” in Table 1 gives the relevant performance data for this run of the verifier.

When a TWS goes off line and is reconnected, the MC retransmits all necessary data to re-populate the database on the reconnecting TWS. Therefore, the TWS does eventually receive all messages necessary to bring it back into synch with the others, but does not get outdated data.

To detect whether the MC/TWS TCP connection can be terminated when a single UTB is lost, we used the assertion-checking mechanism of SPIN. In particular, the following assertion was checked:

```
assert(drop[tws] >= 2 | | tmo >=
1) when disconnect(tws)
```

where `| |` is SPIN syntax for disjunction. The assertion states that a TWS is disconnected only if at least one timeout has occurred or the TWS has experienced at least two message drops.

SPIN reported that this assertion is indeed violated during the execution of the protocol (entry “disconnect one” of Table 1). However this violation can only manifest with abnormal timing delays, which it was later determined could only be caused by a noisy channel. The scenario revealed by SPIN is depicted in Figure 5 for the case `MAX_SEQ = 10`, and can be understood as follows.

During the first phase of UTB transmission from the MC to TWS_1 (in which the retransmission flag is false), UTB number 8 is dropped. As a result, TWS_1 sends `ACK(7,F)` to the MC. Because of delay incurred during UTB data transmission and/or in the TCP transmission of TWS_1 ’s ACK, a timeout is triggered in the MC before it receives the ACK. Consequently, the MC sends another request for acknowledgement, after which it receives the first ACK. Since $7 < 10$, the MC retransmits UTBs numbered 8 through 10. Meanwhile, TWS_1 receives the MC’s second request for acknowledgement (the one transmitted as a result of the timeout in the MC), to which it responds with `ACK(7,F)`. The MC receives `ACK(7,F)` and misinterprets this message as the acknowledgement of its UTB retransmission. This forces the MC to send a `CLOSE_SOCKET` to TWS_1 , shutting down the connection despite the fact that TWS_1 correctly received the retransmitted UTBs, and hence all UTBs (1-10).

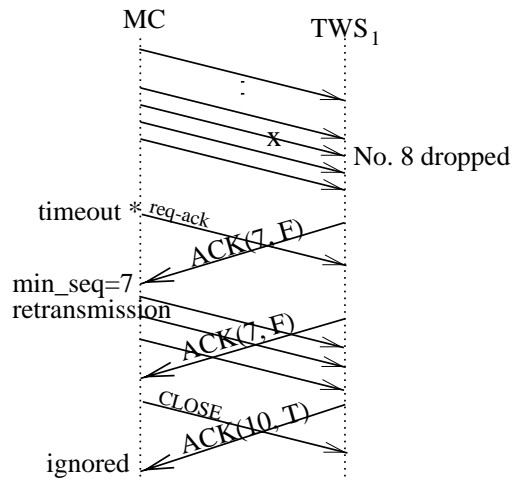


Figure 4. A scenario for disconnect on loss of a single message.

This situation theoretically would be improved by directing the MC to check the retransmission flag in the ACK message type. Specifically, after retransmission and after having received an ACK, the MC will check the retransmission flag in the ACK. If it is set, the MC accepts the acknowledged sequence number in the ACK as valid and based on this information, decides whether to disconnect the TWS. Otherwise, it will continue to wait for an ACK with the retransmission flag set until it times out. The connection is dropped by the MC upon timing out. This modification will not prevent disconnect upon a second timeout. Entry “modified protocol” of Table 1 gives the results of running SPIN in assertion-checking mode on the modified protocol.

Ultimately, it was decided not to recommend the protocol modification. The significant timing delays that are needed for disconnect to occur when a single UTB is lost are virtually impossible under normal E-2C operating conditions and are most likely due to noise on the MC/TWS LAN. Closing the TCP link and reinitializing to attempt to correct the noise problem is preferable to allowing the system to continue in degraded mode (with noise) because (1) it is critical that operators see up-to-date data, and (2) the disconnected TWS can be reinitialized with minimal delay¹ and minimal loss of information to the operator.

¹The time to shut down an MC/TWS connection is approximately 100 msec; the time to reconnect/re-populate the TWS is less than 2 seconds, on average; and the time to reconnect/re-populate is less than 5 seconds in the worst case.

6. Conclusions

We have shown how the MC/TWS protocol of the E-2C Hawkeye Early Warning aircraft can be modeled and verified using a temporal logic model checker. Unlike other datalink-layer protocols, such as the go-back-N protocol of [Tan96], the MC/TWS protocol only attempts one phase of data retransmission in the presence of transmission errors. The main rationale behind this decision is (i) the designers of the protocol did not want to jeopardize the timely delivery of track data to the TWSs in order to deal with a faulty connection between the MC and any one TWS; (ii) the protocol simply terminates the faulty connection if one phase of data retransmission is not sufficient and the TWS in question is immediately permitted to reestablish the connection; and (iii) the data is continuously updated.

A compelling direction for future work involves applying more quantitative methods such as [SS98] to the analysis of the MC/TWS protocol. In particular, it would be interesting to be able to quantitatively specify message loads and probabilities of message loss, and analyze the protocol’s performance as a function of these parameters.

Acknowledgements We thank Northrop Grumman engineers Mark Patalano and Steven Sandler from the E-2C Program for their support, explanations, and progress reviews. The MC/TWS protocol analysis could not have been accomplished without their help. We also thank Andrew Newman for his support. This research was supported in part by AFOSR grant F49620-96-1-0087.

References

- [Bae90] J. C. M. Baeten, editor. *Applications of Process Algebra*. Cambridge Tracts in Computer Science 17. Cambridge University Press, 1990.
- [CW96] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4), December 1996.
- [GR97] J. F. Groote and M. Rem, editors. *Science of Computer Programming, Special Issue on Verification and Validation Methods for Formal Descriptions*, volume 29(1-2), July 1997.
- [Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
- [NGC96] Northrop Grumman Corporation, Bethpage, NY. *Protocol Description: E-2C Display LAN*, November 1996.

- [NGC98] Northrop Grumman Corporation. *The E-2C Hawkeye Website*, 1998. http://www.northgrum.com/Corp_web/products_pages/e2c_hawkeye.html.
- [SS98] E. W. Stark and S. A. Smolka. Compositional analysis of expected delays in networks of probabilistic I/O automata. In *Proceedings of the 13th Annual IEEE Symposium on Logic in Computer Science (LICS '98)*, pages 466–477, Indianapolis, IN, June 1998. IEEE Computer Society Press.
- [Tan96] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.
- [Wol86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 184–192, St. Petersburg, January 1986.