Test Bed for Plain C/C++ Protocol Implementations

Henry Haverinen and Lassi Lehtinen
Nokia Mobile Phones

Abstract This paper presents a protocol software test bed, which was developed for the testing of a Network Access Authentication Protocol (NAAP) implementation. We demonstrate how modular software design can help in protocol testing and we present a simple software interface that enables flexible protocol testing of protocol implementations. Besides normal use scenarios, the test bed allows automatic testing of various error cases, such as dropped, delayed, duplicated and modified packets.

The presented protocol software test bed was successfully used in the testing of a NAAP implementation, and several normally difficult-to-find errors were fixed during the testing. The same test bed design was also used to test a Mobile IP implementation, and it can be applied to other implementations as well.

Keywords: Protocol testing, modular software design

1. INTRODUCTION

Formal Description Techniques (FDTs) and high-level protocol implementation tools that are based on FDTs often include simulation and testing features. SDL [1] is an example of such a language. There also are C and C++ protocol implementation libraries that can assist in the testing or simulation of implementations, such as CVOPS [2] and x-kernel [3]. However, many communication protocols are still implemented without implementation frameworks or libraries, in plain C and C++. This paper describes a protocol software design, which can be used in the testing of such plain C/C++ protocol implementations. The goals of the software design are to make the protocol implementation portable to various platforms and to allow the testing of the implementation in different error

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35584-9_19

O. Martikainen et al. (eds.), *Smart Networks*

scenarios. Thanks to its simplicity, the underlying idea of the test bed implementation could be used to test almost any C/C++ protocol implementation, in many cases even after the implementation has been completed. This is contrary to the testing support of high-level protocol implementation frameworks, which only work for protocol software that has been implemented using the particular frameworks.

The protocol test bed was developed for the testing of a Network Access Authentication Protocol (NAAP) [4] implementation. NAAP is a client/server protocol that runs over the User Datagram Protocol (UDP).

An overview of the relevant parts of the NAAP software architecture is given in Section 2. The architecture and the operation of the test bed is described in Section 3. Section 4 presents the results of NAAP testing. The generality of the test bed is discussed in Section 5. Section 6 contains conclusions.

2. NAAP IMPLEMENTATION

The software architecture of the NAAP implementation, shown inFigure 1, uses a well-known principle in modular software design: the platform-independent parts have been separated from the platform-specific parts with a clearly defined interface. This principle is currently considered one of the advantages of object-oriented design, but software has been structured for portability already in the early 1970's [5].

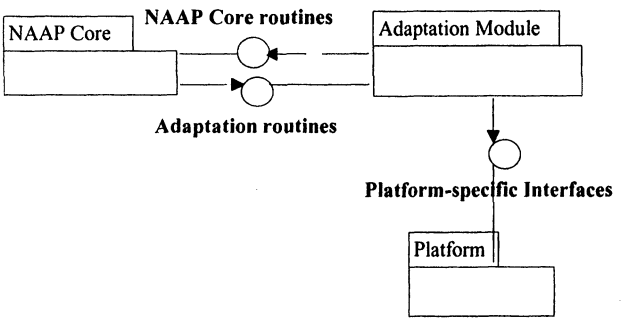

Figure1. Software architecture of the NAAP implementation

The NAAP Core module contains the platform-independent parts of the protocol implementation, while all the system dependent code is in a platform-specific adaptation module. This division makes it easy to port the

implementation to new platforms. One only needs to write an adaptation module for the new platform.

The NAAP Core is a collection of functions and data. It does not own any threads of execution. The adaptation module controls the operation of the NAAP Core by calling the NAAP Core routines in a suitable order. The NAAP Core routines are non-blocking; they complete without waiting for any external events to occur. All waiting needs to be implemented in the adaptation module.

Sending and receiving data are examples of system-specific tasks that are needed in all protocol implementations. For instance, when the protocol needs to send a packet to the network, the NAAP Core calls the `am_sendto()` routine in the adaptation module, which then invokes a platform-specific routine to send the packet, as illustrated in *Figure* When the conventional socket interface is used, the socket operation to send a packet is `sendto()`.

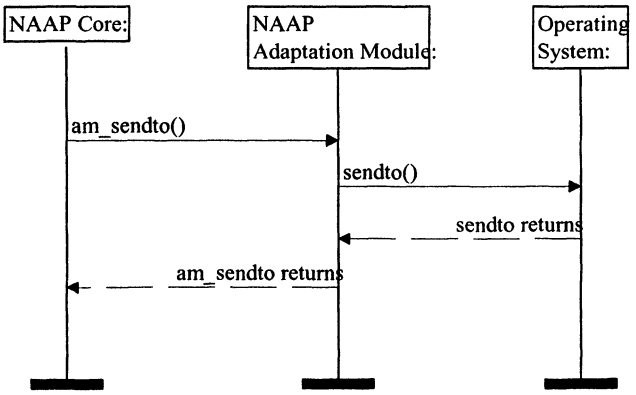

Figure 2. Packet sends in the NAAP implementation

Figure shows how packet receives are implemented. Because the implementation is run by a single thread of execution, non-blocking receives are used. First, the operating system signals the NAAP process that data is available in a socket. On the Unix adaptation, the NAAP adaptation module uses the `select()` system call to wait for multiple waitable objects. When the select call returns and indicates that data has been received to one of the sockets, the NAAP adaptation module passes the indication on to the NAAP Core by calling the NAAP Core `nc_handle_socket()` operation. The NAAP Core has allocated a data buffer, which it passes to the `am_recvfrom()` adaptation routine. This adaptation routine calls the

system call to receive the data to NAAP Core's buffer. The adaptation module does not need to manage buffers for NAAP packets.

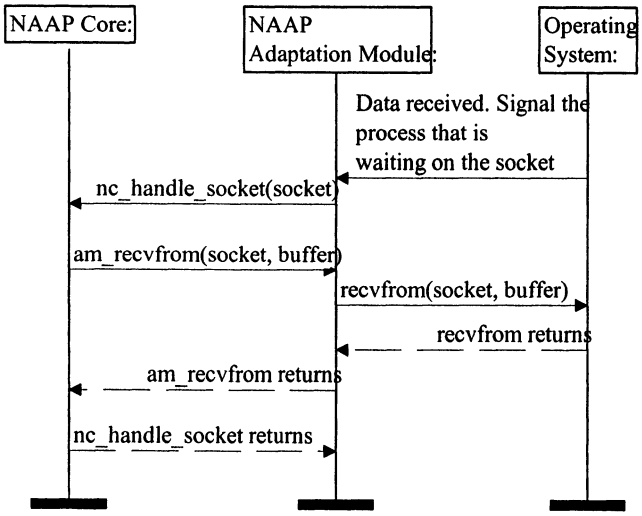

Figure 3. Packet receives in the NAAP implementation

Other examples of system-dependent functionality that is often needed in protocol implementations are memory operations, getting the current time of day for timestamps, and the scheduling of timeouts. In the NAAP implementation, the timeouts are implemented as an event queue in the NAAP Core. The event queue is a list of event descriptors, sorted in increasing order of when the event is to occur. There is a NAAP Core routine that the adaptation module can call to learn the timeout to the first event in the queue. It is the responsibility of the adaptation module to call the NAAP Core routine which processes the expired events after this timeout.

The NAAP Core adaptation interface is also to control the protocol. For example, all the commands the user can give, such as "connect", "cancel connect", "disconnect", are invoked by calling a NAAP Core routine. The NAAP Core gives indications of various events by calling an adaptation routine. There are indications for successful connection, unsuccessful connection, expired connection and a notification for all state transitions in the protocol so that progress bars can be updated and so on.

3. TEST BED IMPLEMENTATION

The implemented NAAP test bed is a special adaptation module that contains additional testing functionality. In other words, the test bed hooks up between the NAAP Core module and the actual adaptation module. The adaptation routine calls are routed through the test bed module, which can then generate various error conditions.

For example, to test the protocol implementation when a packet is dropped, the test bed implementation of `am_sendto()` returns a successful value without actually sending the packet to the network, as illustrated in *Figure.* To duplicate packets, the test bed implementation of `am_sendto()` calls the underlying actual adaptation module routine twice.

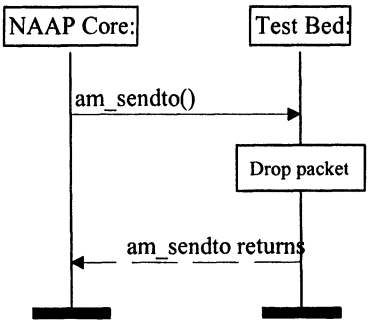

Figure 4. Testing of dropped sends

The testing of delayed sends is shown in Figure 5. For simplicity, the test bed and the actual adaptation module are shown as a single entity in the figures or this section. To delay a packet, the test bed `am_sendto()` routine schedules a timer to send the packet later. The test bed can only refer to NAAP Core's data buffer and other parameters given to the `am_sendto()` routine, such as the destination IP address and port, while in the context of `am_sendto()`. Therefore, the test bed needs to copy the data buffer and other parameters to a send descriptor it has allocated itself. While the timeout is pending, the adaptation module processes all events as usual. When the timeout elapses, it calls the underlying operating system routine to send the packet using the information in the send descriptor. After sending the packet, the test bed frees the send descriptor. Our implementation of the test bed delays packets using the same event queue that is used for NAAP timeouts.

The test bed `am_sendto()` routine can also truncate or otherwise modify the packet. Introducing random errors in the packet is trivial, but the

NAAP test bed is also able to parse NAAP packets and take actions that depend on the contents of the NAAP message. For example, the test bed is able to modify a given field, or insert a given extension to the packet.

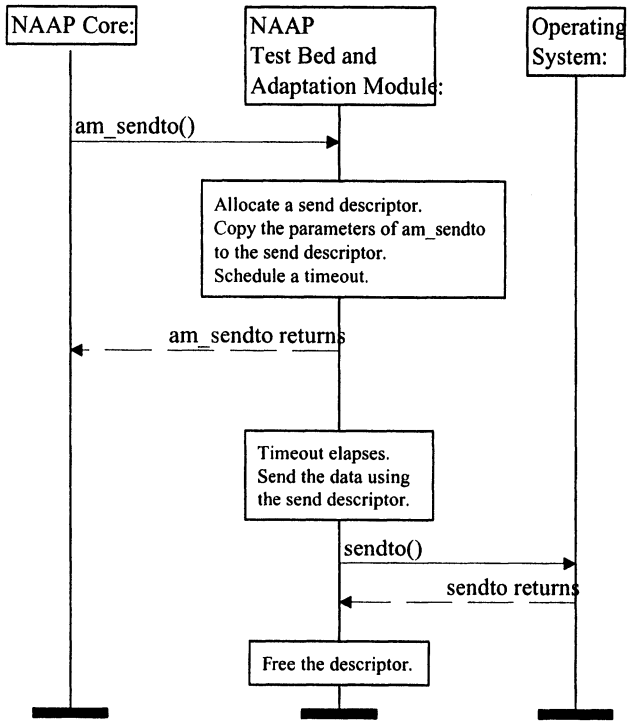

Figure 5. Testing delayed sends

Because we were testing both the NAAP client and the NAAP server entities, it was sufficient for us to implement testing functionality to sends only and not hook up to receives. Testing the sends in the client and in the server covers the NAAP packets both ways. However, if it was necessary, similar operations could be implemented when receiving packets too. In the NAAP implementation, the adaptation module is responsible for detecting when data is available to be read from sockets. Hence it is easy to layer test functionality for received data too. For a simple example, to drop a received packet, the adaptation module can simply fail to call the `nc_handle_socket()` routine, which is used to indicate the NAAP Core of received data, and ignore the received data.

Modifying received packets can be implemented by reading the received data to a buffer allocated by the test bed, as shown in Figure 6. When the

operating system indicates that data has been received, the test bed allocates a receive descriptor. The test bed then reads the received data to a buffer that is included in the receive descriptor and saves all the related information of the received packet, such as the source IP address, to the receive descriptor. Before indicating the packet to NAAP Core with `nc_handle_socket()`, the test bed modifies the received data. When the NAAP Core calls `am_recvfrom()` to read the received data, the test bed does not pass the buffer to the underlying operating system, as the actual adaptation module does, but the test bed copies the contents of the modified buffer from the receive descriptor to NAAP Core's buffer, and also supplies the related information it has stored in the descriptor.

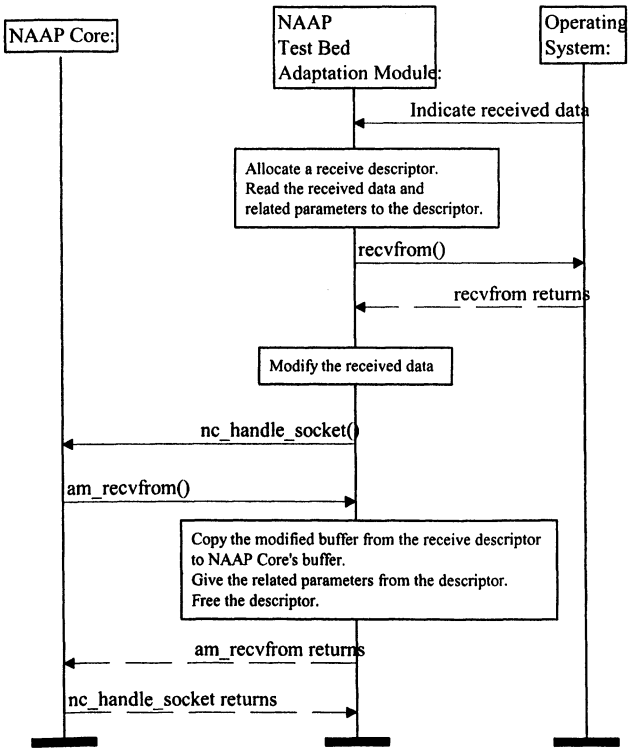

Figure 6. Modifying received packets in the test bed

Modifying data could also be implemented in the `am_recvfrom()` function of the test bed by reading the data to NAAP Core's buffer and then modifying it. However, using a receive descriptor is a more general way of implementing test functionality for receives as it also enables the testing of delayed and duplicated receives.

Duplicated receives can be implemented with a similar mechanism. The test bed needs to copy the received data in a buffer it has allocated, because the underlying `recvfrom()` routine would only give the data once. After reading the data to a buffer, the test bed can indicate the data to the NAAP Core twice, and copy it to the NAAP Core's buffer from its own buffer.

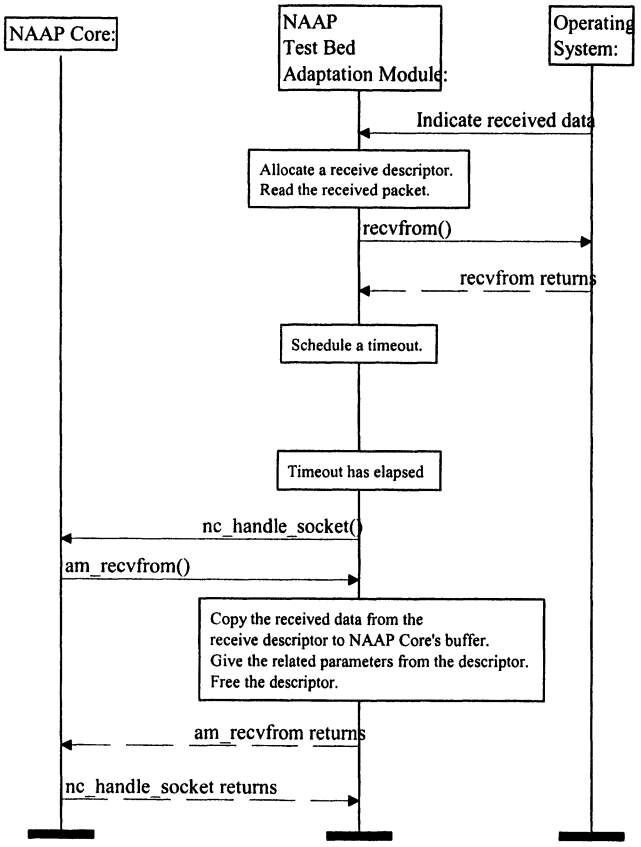

Figure 7. Testing delayed receives

The testing of delayed receives is illustrated in *Figure*7. To delay a received packet, the test bed schedules a timeout and indicates the packet to the NAAP Core only after the timeout has elapsed. The test bed implementation for delayed packets is also implemented by copying the received data and the related parameters to a receive descriptor when the operating system indicates that data has been received. If the test bed didn't copy the data but left the data in the operating system buffers, the packets would still be received in the correct order, which is not the desired

operation when testing delayed packets. After copying the received packet to a receive descriptor, the test bed then schedules a timeout after which NAAP Core is indicated. While the timeout is pending, the adaptation module processes all events as usual. Other packets can be received without delaying them. When the timeout eventually elapses, the test bed indicates the NAAP Core, and the received data from the receive descriptor is copied to NAAP Core's buffer.

Our test bed implementation includes a configuration file with which different test cases can be specified. There are different conditions when to employ any of the above mentioned error cases. For example, the configuration file can include an entry that specifies the probability of dropping a certain NAAP packet.

The NAAP test bed makes use of the notifications given by the NAAP Core. By hooking up to these notifications, the test bed is able to monitor the state of the protocol. This is exploited for example by employing test case after a certain state transition has occurred. The testing of the "cancel connect" feature is a good example of this. Because the user can click the Cancel button in any phase of the connection establishment, it is hard to test this feature manually. We had a test case for cancelling the connection in each intermediate state of the connection establishment, based on notifications from the NAAP Core. The notifications also enable the test bed to automatically tell if a test case has passed or failed. This makes it possible to prepare a set of test cases, which can be automatically performed when the protocol implementation has been changed.

4. NAAP TESTING IN PRACTISE

In total, the NAAP testing plan defines 88 test cases that make use of the test bed. Six cases were related to the basic use scenarios of the protocol. In 15 cases, random errors were introduced in different types of packets. There were 15 cases of truncating packets, 11 cases for dropping packets and 15 cases for duplicated packets. Certain fields of the packets were modified in 19 cases. In addition, there were 7 cases to test the cancel connection feature, already discussed in the previous section.

As usual in software development, many bugs and anomalies in the NAAP implementation were found and fixed during the implementation in informal "tryouts" performed by the programmer. The actual testing begun only after the programmer had tried out the new build by running the most typical use cases and fixed any problems encountered. These bugs were not counted, but they numbered in hundreds. In the actual testing that was performed using the test bed, circa 50 bugs were found and fixed.

In six bugs, the protocol implementation did not check for the validity of a field in a protocol message as required in the protocol specification. The implementation accepted a message, which had been modified by the test bed to be invalid and thereby should have been silently ignored. There were two bugs, which caused the implementation to crash when the test bed set a protocol field to an invalid value.

In NAAP, some of the messages are required to include the Authenticator extension, which provides for message integrity protection against tampering. The specification lists two cases when exactly one Authenticator extension must be present. Three different bugs were found by removing the mandatory Authenticator extension or inserting an extra Authenticator extension in the test bed. The protocol implementation accepted the messages with invalid or missing Authenticator extensions.

One category of the bugs was related to message retransmissions. When a packet was lost, there were cases when the sender did retransmit the packet as specified, but the recipient did not accept the retransmissions. In some of the cases, a lost packet caused the protocol to enter a wrong state or to misschedule retransmission timers.

About half of the bugs were individual cases, which cannot be categorized with other bugs. For example, there were only individual errors caused by delayed or duplicated packets.

5. GENERALITY OF THE TEST BED

To prove that the concept of the test bed implementation can be generalized to other protocol implementations, we tested an earlier Mobile IP implementation, described in [6] with a version of the test bed. Thanks to the modularity of the Mobile IP implementation, it was easy to hook up the test bed to the software. Because we had used the same software interface for packet operations as in the NAAP implementation, we were able to plug in the test bed without changing the platform-independent part of the Mobile IP implementation.

We could not re-use the portions of the test bed that tested NAAP-specific functionality or hooked up to NAAP specific notifications. Obviously, writing tests that are aware of Mobile IP features requires Mobile IP specific work.

The test bed could be used to test other modular protocol implementations with small modifications. In most operating systems, the socket interfaces are descendants of the Berkeley Software Distribution (BSD) socket interface and hence the routines for sending and receiving data are very similar. If the original designer of the software has used wrapper

functions around the socket operations, it should be easy to hook up the protocol test bed.

Some protocol implementations do not separate the platform-specific parts in a different place but call socket operations directly. If the socket calls are all over the code, then preprocessor directives can be used to hook up a test bed. For example, there can be a preprocessor definition that replaces calls to the socket function `sendto()` with calls to `testbed_sendto()`. Some protocol implementations that have multiple threads or processes use blocking receives and call the `recv()` or `recvfrom()` routine without waiting for any separate indications first. Simple preprocessor definitions are sufficient to implement test functionality for receives in these cases.

Testing non-blocking receives requires more work. It is hard to come up with a general purpose preprocessor directive to take care of the `select()` system routine that is used for waiting for events to occur in sockets or file descriptors. Fortunately, there usually is only one or two places in the source code where `select()` is used, so it should be quite easy to hook up a test bed by modifying these parts. If we want to test delayed, duplicated or dropped receives, then the test bed may need to use a separate thread which waits in `select()`, and wakes up the protocol implementation thread when it wishes to indicate an event to the protocol.

Alternatively to protocol test beds that are linked with the protocol implementation, test functionality can also be implemented lower in the protocol software stack. For example on Windows, we could implement test functionality in an intermediate network driver, which resides between the TCP/IP stack and network interface card drivers. This test bed would be able to test any protocol implementation, even a binary executable that cannot be modified. However, when the test bed is linked with the protocol implementation, it is more convenient to debug the software when an error is found. It would also be harder to automatically detect passed and failed test cases in a driver-level protocol test bed. In order to test high-level issues, it is better to hook to higher level in the protocol stack.

6. CONCLUSIONS

The protocol test bed design described in this paper has been successfully used in the testing of a NAAP implementation and a Mobile IP implementation. Several errors that would have been hard to find in black box testing were fixed and the test bed proved to be easy to use and flexible.

Our experience is that modular software design facilitates not only implementation and porting but also testing. It is advantageous to implement

packet sends and receives, as well as protocol control and notifications so that test functionality can easily be hooked up.

REFERENCES

[1] "Specification and Description Language (SDL)", ITU-T Recommendation Z.100, November 1999

[2] J. Harju, A. Karila, J. Kuittinen, J. Malka: "CVOPS, a tool for the implementation and testing of computer communications software", Technical Research Centre of Finland, Telecommunications Laboratory, 1986

[3] N. Hutchinson, L. Peterson, "The x-Kernel: An Architecture for Implementing Network Protocols", IEEE Transactions on Software Engineering, Vol. 17, No.1, January 1991

[4] H. Haverinen, "NAAP: A User-to-Network authentication Protocol", Proceedings of Smartnet 2002 Conference, April 2002

[5] P.C. Poole, W.M. Waite, "Portability and Adaptability", Advanced Course on Software Engineering, Lecture notes in Economics and Mathematical Systems 81, Springer 1973

[6] H. Haverinen, A. Kuikka, T. Määttänen, "A Portable Mobile IP Implementation", Proceedings of the IEEE Local Computer Networks 2000 Conference, November 2000