Toward a secure TCP/IP stack — Technical report

Guillaume Cluzel

July 21, 2020

Contents

1 Introduction

TCP is a widely used network protocol to communicate in the Internet as it is used by the HTTP protocol, the FTP protocol, and so many others. Ensuring the safety of the TCP/IP stack is essential for the safety of a lot of machines. While a lot of work has been done on higher level protocols, such as SSL/TLS which is built on top of TCP and designed to provide a security layer to TCP with for example the work done in miTLS [1], or in lower level with the work on RecordFlux to safely parse data segment [2], nothing has been done to make a secure TCP implementation. Nonetheless, TLS security can only be ensured if the underlying TCP implementation is free of bugs.

CycloneTCP is a library developped by Oryx Embedded for embedded platforms. A large number of platforms are supported, and the library can be used with a dozen of OS. The code of the library is written in C and contains plenty protocols from low-level network layers like IPv4/IPv6, to transport layers and application layers like DHCP or HTTP. In particular an implementation of the TCP protocol is provided.

SPARK 2014 is a programming language designed as a subset of Ada that helps in making high reliability sofwares by providing powerful static analysis methods. SPARK is able to detect uninitialized variables with control flow analysis, and can ensure the absence of run-time errors, but also, based on SMT-solver, functionnal behavior can be specified for every functions and mathematically proved.

The aim of this work was to translate some parts of the CycloneTCP TCP/IP stack in Ada/SPARK to improve the safety of the code. In particular we focused on the TCP protocol to ensure that the norm is respected in the code.

This report is a synthesis of the challenge of verifying a such protocol and the solution we found during the internship to go toward a secure protocol.

2 The TCP protocol

2.1 Why verifying a TCP stack?

As mentionned in the introduction, TCP protocol is the base of a lot of other protocols, in particular SSL/TLS. These higher protocols cannot be secure if the underlying TCP protocol has bugs. An incorrect implementation of the TCP protocol can lead to a crash in the communication if two machines cannot communicate together, error in retransmissions, or even security failures. CycloneTCP is designed for embedded code, and a failure in the code can be critical.

2.2 Sockets

A socket is nothing more than a data structures that contains the connection information, like for example the local or the remote IP address as well as the state of the connection and other information require by the TCP protocol. In Ada, a socket will be represented by a pointer to a record type:

```
type Socket_Struct is record
    S_Descriptor     : Sock_Descriptor;
    S_Type           : Socket_Type;
    S_Protocol       : Socket_Protocol;
    S_Net_Interface  : System.Address;
    S_localIpAddr    : IpAddr;
    S_Local_Port     : Port;
    S_Remote_Ip_Addr : IpAddr;
    S_Remote_Port    : Port;
    S_Timeout        : Systime;
    State            : Tcp_State;
    -- Other fields
end record;
type Socket is access Socket_Struct;
```

A socket is also the structure that is manipulated by the users as an opaque structure, through an API to perform operations on the socket itself and on the global environment by sending messages in the network for example.

2.3 Presentation of the protocol

TCP is defined by the norm RFC 793 [3] in a high level language. TCP is a realiable, ordered and error-checked connection oriented protocol. It means that two computers have to open a connection first before sending data. The connection is closed once all the data has been transmitted. The flow of a connection (if no error happens) contains three main steps:

1. Opening the connection,
2. Sending & receiving the data,
3. Closing the connection.

This mechanism is decribed by a state machine.

As it can be seen in the state machine diagram in figure 1, a socket can take different states that are subject to change when a message (or a segment) is sent or received or when an action is performed by the user. A label A/B on an edge in the graph refers to a stimuli performed on the socket (the reception of a message, a user action...) for A and a flag sent for B. More precisely, for A, a text in italic shape refers to an action performed by the user (a call of a user function) or when it's a timeout the action is automatically performed by a timer, and a ACK, SYN or FIN refers to a flag contained in a received message. The flags in B refers to the flags sent in response of the stimuli A. Without giving unnecessary details on the format of a TCP header, we can describe some flags that can be contained in a TCP header in order to help the understanding of the figure 1:

ACK Acknowledgment field significant. The last message received by the sender is acknowledge. A field of the header contains the number of this segment.

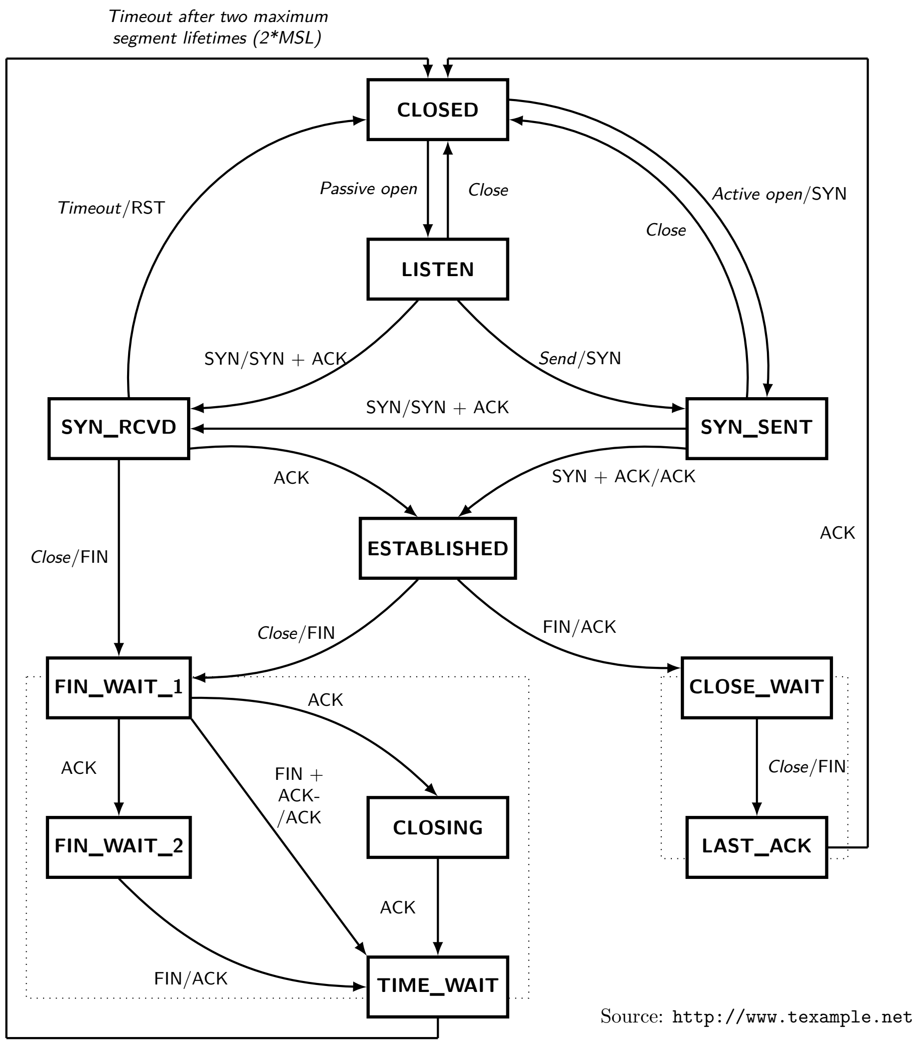
SYN Synchronize sequence number. This flag is sent in order to establish a connection.

FIN No more data from sender. This flag is sent in order to close the connection.

Different states can be taken by a socket during its lifetime depending on the state of the connection. The state also gives information about the state of the remote TCP. Each state has its own signification:

CLOSED represents no connection state at all.

LISTEN represents waiting for a connection request from any remote TCP and port.



Source: `http://www.texample.net`

Figure 1: TCP automaton.

SYN-SENT represents waiting for a matching connection request after having sent a connection request.

SYN-RECEIVED represents waiting for a confirming connection request acknowledgment after having both received and sent a connection request.

ESTABLISHED represents an open connection, data received can be delivered to the user. The normal state for the data transfer phase of the connection.

FIN-WAIT-1 represents waiting for a connection termination request from the remote TCP, or an acknowledgment of the connection termination request previously sent.

FIN-WAIT-2 represents waiting for a connection termination request from the remote TCP.

CLOSE-WAIT represents waiting for a connection termination request from the local user.

CLOSING represents waiting for a connection termination request acknowledgment from the remote TCP.

LAST-ACK represents waiting for an acknowledgment of the connection termination request previously sent to the remote TCP (which includes an acknowledgment of its connection termination request).

TIME-WAIT represents waiting for enough time to pass to be sure the remote TCP received the acknowledgment of its connection termination request.

For more details on how two TCP can establish or close a connection, the reader can refer to the TCP norm where pleinty of scenario are described and explained, and more details are given on the state and segments that can be sent.

2.3.1 Multithreading in the implementation

Different threads can interact with a socket. The TCP norm gives an example with three threads: one for the *user calls*, one for the *arriving segments* and one for the *timers*. This design has also been adopted in CycloneTCP.

User calls User calls refers to functions that can be called by the user to control the connection, send, or received data: OPEN, CLOSE, ABORT, SEND, RECEIVED. Transition between state of the connection can happen during the call of these function since there are made to control the connection.

Arriving segments In this threads, the received segments are processed. The corresponding messages are sent back. Transitions between states can happen here when a message is received. For example, this is the case when a message containing a SYN is received when the socket is in the LISTEN state. It will automatically send a SYN + ACK in response, and change its state for SYN-RECEIVED.

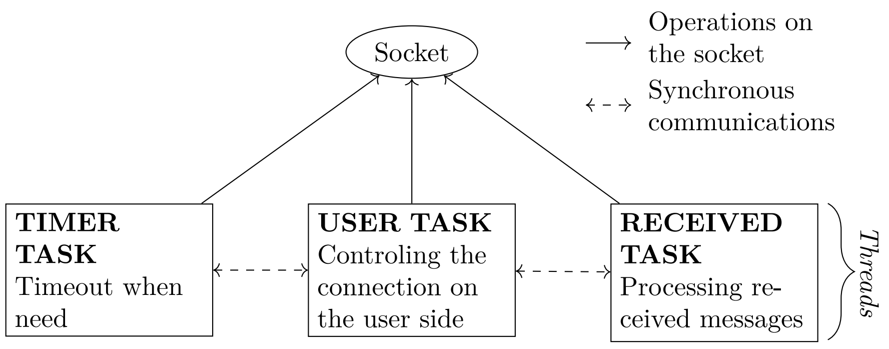

Socket
Operations on the socket
Synchronous communications
TIMER TASK
Timeout when need
USER TASK
Controling the connection on the user side
RECEIVED TASK
Processing received messages
Threads

Figure 2: Concurrency in the TCP protocol.

Timers The timers control the timeouts like the retransmission timeout to retransmit a message, or the time-wait timeout to close the connection after an amont of time. Then, transitions can also happen in this threads.

To sum up, a single socket can be seen as a shared structure over multiples threads, that can all manipulate it and change the value of its fields. The schema in figure 2 illustrates that.

Only one thread can perform an operation on the socket at the same time. The access to the fields of the socket are protected by mutex.

Two threads can communicate synchronously or asynchronously together. The synchronous communications are based on the interface provided by the OS, in particular with events.

3 Challenges

3.1 Main focus for the verification

The TCP norm defines a lot of behavior that could be verified by automatic proof. Here is a non exhaustive list of what could be done:

- Verification of the transitions between the states.
- Integrity of the messages sent, *i.e.* we could check that the sent message contains the correct flag and the correct sequence and acknowledgement numbers.
- Integrity of the messages received, *i.e.* we could check that the messages received are correctly processed in regard of the flags they contain.
- Functionnal correctness of the user functions in regard to their specifications.

Some choices have had to be done in regard to the short time devolved to the internship, and not every aspect of the TCP norm has been proved yet. However, some aspects have been selected for the challenge or the importance they represent. A list of these aspects is presented here and the choices will be motivated in the next subsections. First of all, we have wanted to improve the interface of the functions. The other point was to improved the safety of the protocol itself.

3.1.1 Improve functions interface

One of the main problems pointed by the main writter of the CycloneTCP library is that users do not correctly use the API supplied. As a result, some calls of functions can be incorrect, either because the arguments are incorrect, or because the return code of the previous functions has not been checked, which drive to an incorrect code.

3.1.2 Respect to the protocol

To ensure the safety of the library, it is necessary to ensure that the functions really do what they are supposed to do. It is a part of the job that has been done. More especially we have wanted to check if the state machine is respected by all the user functions in the transition they do. We have also tried to ensure the correctness of the function called regarding to the TCP state of the socket before the call to respect the specifications given in the norm.

3.2 Technical problems encountered

The multiples tasks and the changes that can happen at every moment in one task or another make the verification hard. Multiples interactions exist between the multiples threads: synchronous and asynchronous. All these interraction have to be considered if we want to write correct contracts in regard of what could be done in other threads.

As SPARK does not have a native mode to deal with concurrency, we have to model these different interactions by hands by writing assertions that modify the state of the socket as another thread could do it. In particular the problem has been encountered when we tried to write contracts for the correctness of the user functions. More details and explainations will be given in the following section, where we will investigate the solution found to these technical problems.

4 Solutions found

4.1 Order to call the functions

This part mainly focussed on the high level user functions, the socket functions. The socket functions are the one manipulated by the user to perform operations on the network. They are located in the files `socket_interface.ad(b|s)`.

It is clear that there exist an order on which the functions have to be called. This order can be determined by the norm, and must respect the order given by the graph in figure 1. Then, we want post- and pre-conditions to model a partial order on the calls of the function. If two functions f_1 and f_2 are ordered such that $f_1 \preceq f_2$ where $\preceq$ is a relation over the oder of which functions have to be called, then we want that the post-condition of f_1 implies the precondition of f_2, *i.e.* $\text{Pre}(f_2) \subseteq \text{Post}(f_1)$.

We will give an example over functions `Socket_Connect` and `Socket_Send`. The function `Socket_Connect` try to connect to a distant TCP. If the connection succeeds, the remote IP address of the distant TCP is set in the field `S_Remote_Ip_Addr` of the `Sock` structure. When a user call the function `Socket_Send`,

```
procedure Socket_Connect
    (Sock           : in out Not_Null_Socket;
     Remote_Ip_Addr : in     IpAddr;
     Remote_Port    : in     Port;
     Error          :    out Error_T)
  with
   Pre => Is_Initialized_Ip (Remote_Ip_Addr),
   Contract_Cases => (
     Sock.S_Type = SOCKET_TYPE_STREAM =>
       (if Error = NO_ERROR then
         Sock.S_Type = Sock.S_Type'Old and then
         Sock.S_Protocol = Sock.S_Protocol'Old and then
         Is_Initialized_Ip (Sock.S_localIpAddr) and then
         Sock.S_Local_Port = Sock.S_Local_Port'Old and then
         Sock.S_Remote_Ip_Addr = Remote_Ip_Addr and then
         Sock.S_Remote_Port = Remote_Port and then
         Sock.State = TCP_STATE_ESTABLISHED
       else
         Sock.S_Type = Sock.S_Type'Old and then
         Sock.S_Protocol = Sock.S_Protocol'Old)
     others => True)

procedure Socket_Send
    (Sock    : in out Not_Null_Socket;
     Data    : in     Send_Buffer;
     Written :    out Natural;
     Flags   :        Socket_Flags;
     Error   :    out Error_T)
  with
   Pre  =>
     Is_Initialized_Ip(Sock.S_Remote_Ip_Addr)
```

Figure 3: An example of how functions calls can be ordered by Pre- and Post-conditions.

we want to ensure that the connection has already been established. This is why a precondition of this function is `Is_Initialized_Ip(Sock.S_Remote_Ip_Addr)`.

4.2 Check of return code

An observation made by Clément Zeller, the main programmer of the library is that customers do not always think to check the return code of the socket user functions, that can fail. If the return code is not checked, some assumption on the result cannot be done.

The post-conditions as we have written them in SPARK, ensure that the return code is checked before continuating processing. Figure 3 shows this mechanism for the procedure `Socket_Connect`. The post-condition filters the cases where `Error` is `NO_ERROR` and where `Error` takes another value. As a result, a user who would write an incorrect code such as

```
Socket_Connect (Sock, Remote_Ip_Addr, Port, Error);
Socket_Send (Sock, Data, Written, Flags, Error);
```

would be warned by gnatprove with a message such as

```
medium: precondition might fail.
```

whereas the following code is correct

```
Socket_Connect (Sock, Remote_Ip_Addr, Port, Error);
if Error /= NO_ERROR then
    return;
end if;
Socket_Send (Sock, Data, Written, Flags, Error);
```

4.3 Verification of the state machine

The aim of this part is to explain how we have done to check that the transitions done in a function respect the TCP automaton. In particular, we were interested in verifying the high level user functions. These functions are not the one in which most of the transitions are done. Even, the verification of these functions is important in order to guarantee the safety of the whole library. The functions of interest are located in the file `tcp_interface.ad(b|s)`.

In order to verify the transitions we have read the norm to extract informations about the allowed transitions, and we have added contracts in the function `tcpChangeState`. Since all the transitions are made through this function, an incorrect transition will be warned by SPARK.

As mentionned, since most of transitions are done outside the user functions, a big part of this work was to consider other transitions in other threads and we present how we have done it in the following sections.

4.3.1 Overview of the concurrency challenge

To ensure the safety of our library, we need to consider all what can happen everywhere in the library. All the functions are protected by mutex, which means that only one operation can be performed at the same time on a socket. Interractions must be considered at two locations: between the function calls, when the mutex is released and during the function call, when the program waits for an event. We will explore in the sections 4.3.3 and 4.3.4 how we have dealt with this two different concurrent mechanisms.

Before everything else, let's see where the interferences mainly come from, and how we have done to ensure that all have been considered.

4.3.2 Process segments functions

Almost all the transitions are done in the file `tcp_fsm.c`. This file has not been translated in SPARK by a lack of time. Then we do not have strong guaranties on its functions behavior. However, for formal verification we need to know what is done in the functions of this file, because a wrong contract can make all the verification wrong.

The file `tcp_fsm.c` is in charge of processing the incoming segments. The reader is strongly encouraged to have a look at this file. The main function of this file `tcpProcessSegment` looks for the socket corresponding to the received segment, and then according to the TCP state of the socket, process the segment as expected by calling one of the function `tcpState<StateName>`. The functions familly `tcpState<StateName>` check the information contained in the segment and can perform a change of state in the socket struct depending on the flags received.

In these functions, a change of state can be performed, as mentioned, thanks to the function `tcpChangeState`. To summarize what is done when a message is received, rather than reading the code and locate all the calls to functions `tcpChangeState`, we have used Klee[1]. It helped to find behaviors that had not been imagined at first. Anyway, we cannot state that all the work done with Klee is complete, and we would need to rewrite this part in SPARK to have a code formally proved.

All what is related to Klee is present in the folder `klee/` and can be compiled and run thanks to the makefile to reproduce the results. Roughtly, what is done is creating a random incomming message, put the socket in the desired state and call the desired function to see what final states can be obtain.

Finally, all the results found by Klee has been reported as a post-condition of the function `Tcp_Process_One_Segment`. This function is essential for all the verification of concurrent parts and is reused everywhere the concurrency is involved to compute possible interferences. It follows that the safety of the code rely on the confidence we have in this function.

4.3.3 Concurrency: asynchronous changes of state

Between the user functions calls, segments can be received and these receptions can drive to changes of the state of the socket. Between two function calls, an infinite number of segments can be potentially receive. The reception of one or zero segment is modeled by the function `Tcp_Process_One_Segment` in term of change of states. Then we have to consider the iteration of `Tcp_Process_One_Segment` in order to compute the result of the reception of multiple message.

Let $\rightarrow$ be the transition function in the TCP automaton restrited to the transitions that can be performed by the reception of a message and its automatic response mechanism. Then we have $\rightarrow =$ `Tcp_Process_One_Segment` and we need to consider the reflexive transitive closure $\rightarrow^*$ of $\rightarrow$, with

$$\rightarrow^* = \bigcup_{n \in \mathbb{N}} \rightarrow^n$$

Now, by examining the TCP automaton in figure 1, we see that all state is only reachable by a state at a distance less than 3 (the maximum path is between SYN-SENT and CLOSE-WAIT if we pass by state SYN-RECEIVED and ESTABLISHED) without user action. Since we only consider the transitive closure in term of states, we can significantly reduce the number of iteration of $\rightarrow$ to compute $\rightarrow^*$ and finally we get:

$$\rightarrow^* = \bigcup_{n=1}^{3} \rightarrow^n$$

[1] `https://klee.github.io`

Algorithm 1: Reflexive and transitive closure of `Tcp_Process_One_Segment`

```
function Tcp_Process_Segment(Socket)
    $S_{last}$ := Socket;
    $S$ := $S_{last}$;
    for $i = 1$ to 3 do
        $S_{last}$ := Tcp_Process_One_Segment($S_{last}$) ;
        $S := S \cup S_{last}$;
    end
    return $S$;
end
```

It follows that algorithm 1 is enough good to compute the transitive closure of the function `Tcp_Process_One_Segment` by taking advantage of the fact that SPARK can unroll small loops to obtain more precise results.

In the user functions, we have added a call to `Tcp_Process_Segment` everywhere there is a call to `Os_Acquire_Mutex (Net_Mutex)`. Doing that helps to ensure that all the possible that all the possible input states are have been considered.

4.3.4 Concurrency: Synchronous exchange

The other mechanism to deal with concurrency is probably more difficult to apprehend and the use of the function `Tcp_Process_One_Segment` is even more noticeable.

The C code contains a function `tcpWaitForEvents` in the file `tcp_misc.c` that checks if the event is true when the function is called, by calling the function `tcpUpdateEvents`. If the wait is not completed at the time the function is called, then the mutex that was previously locked is released until the waited event happens. Then, everything can happen meanwhile.

In the code, the function `tcpUpdateEvents` is called at different locations and can raise the desired event. In particular this function is called each time the state of the socket is changed. This is sufficient for our purpose since we are only interested in the state changes.

When a segment is received, the function `tcpUpdateEvents` is called if a change of state happens. Then, we can consider the algorithm 2 that reuse `Tcp_Process_One_Segment` to compute the most precise set of possible final states after the wait. *(This is done in the function `Tcp_Wait_For_Events_Proof`, the function dedicated to proof in file `tcp_misc_binding.adb`.)*

We can compute precisely the states reach for each event waited thanks to the fact that SPARK unrolls loops.

Weakness If two threads want to lock the same mutex at the same time, which one win? This is probably something to discuss with Clément, because the function `tcpWaitForEvents` could have a different semantic than what we imagine at first.

Algorithm 2: Function to compute the possible state after when waiting for a particular event.

```
function Tcp_Wait_For_Events(Socket, Event, Event_Mask)
    $S_{last}$ := Socket;
    $S$ := $S_{last}$;
    $E$ := Tcp_Process_One_Segment($S_{last}$);
    if $E \wedge Event\_Mask \neq 0$ then
        return $S$;
    end
    for $i = 1$ to 3 do
        $S_{last}$ := Tcp_Process_One_Segment($S_{last}$) ;
        $S := S \cup S_{last}$;
        $E$ := Tcp_Process_One_Segment($S_{last}$);
        if $E \wedge Event\_Mask \neq 0$ then
            return $S$;
        end
    end
    return $\emptyset$;
end
```

5 Future work

Other improvements that could be done in the future:

- Fully translate the file `tcp_misc.c` in Ada to have more guarantees.
- Same for the file `tcp_fsm.c` that has been investigate with klee. But it is not enough and we need more guarantees for this file, in order to prove the interactions between threads first, and also to prove the correctness of this part of code w.r.t. the TCP norm.

References

[1] Karthikeyan Bhargavan et al. "Implementing TLS with verified cryptographic security". In: *2013 IEEE Symposium on Security and Privacy.* IEEE. 2013, pp. 445–459.

[2] Tobias Reiher et al. "RecordFlux: Formal Message Specification and Generation of Verifiable Binary Parsers". In: *Lecture Notes in Computer Science* (2020), pp. 170–190. ISSN: 1611-3349. DOI: 10.1007/978-3-030-40914-2_9. URL: http://dx.doi.org/10.1007/978-3-030-40914-2_9.

[3] *Transmission Control Protocol.* RFC 793. Sept. 1981. DOI: 10.17487/RFC0793. URL: https://rfc-editor.org/rfc/rfc793.txt.