Layered Formal Verification of a TCP Stack.

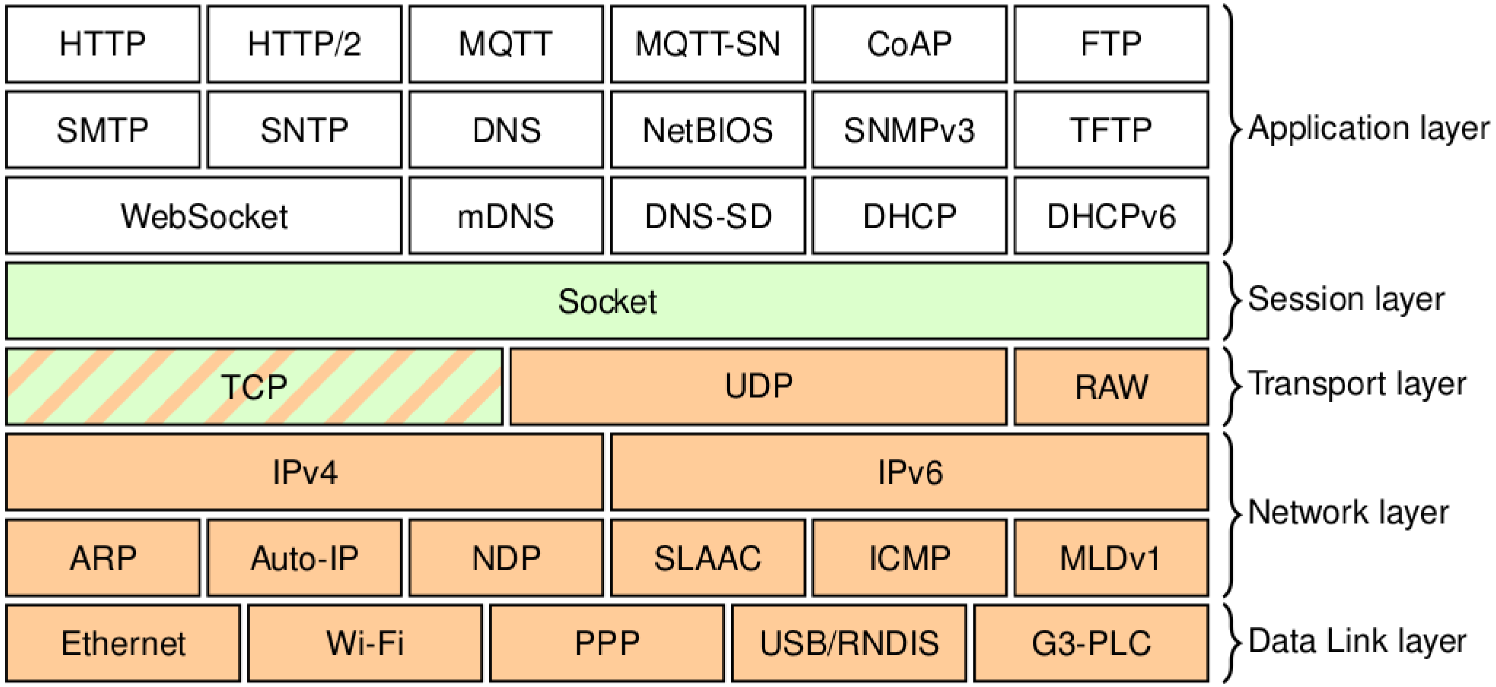
Guillaume Cluzel (AdaCore & ENS de Lyon)
Kyriakos Georgiou (AdaCore & University of Bristol)
Yannick Moy (AdaCore)
Clément Zeller (Oryx Embedded)

Introduction

- By 2030 ~ 10^{12} IoT devices
- 2019: critical security vulnerabilities in a TCP implementation that affect billion of IoT devices

=> Remote code execution

Network communications



=> Oryx Embedded stack: CycloneTCP

Ada programming language

- **Ada** is a Object Oriented Programming language.
- Great safety characteristics like
  - strong typing;
  - runtime checks;
  - Contracts (precondition and postconditions)

```
procedure Tcp_Connect
   (Sock           : in out Not_Null_Socket;
    Remote_Ip_Addr :        Ip_Addr;
    Remote_Port    :        Port;
    Error          :    out Error_T)
   with
     Pre => Sock.S_Type = SOCKET_TYPE_STREAM and then
            Is_Initialized_Ip (Remote_Ip_Addr) and then
            Remote_Port > 0 and then
            Sock.State = TCP_STATE_CLOSED,
     Post =>
         (if Error = NO_ERROR then
            Sock.S_Remote_Ip_Addr = Remote_Ip_Addr
         else
            Sock = Sock'Old);
```

SPARK programming language

- SPARK is a subset of Ada that is suitable for formal verification.

- GNATprove is a tool to formally verify SPARK code:
  - Flow analysis to check data dependancies and variables initialization
  - Deductive verification to prove functions contracts and runtime error via weakest-precondition calculus.

- SPARK and C code can be interfaced.

```
function Os_Get_System_Time return Systime
  with
   Import        => True,
   Convention    => C,
   External_Name => "osGetSystemTime",
   Global        => null;
```

TCP protocol (I)

- TCP is one of the most used protocol in the Internet.

- Reliable connection-oriented protocol:
  - All the messages are delivered
  - In the order they are sent
  - Error-checking mechanism

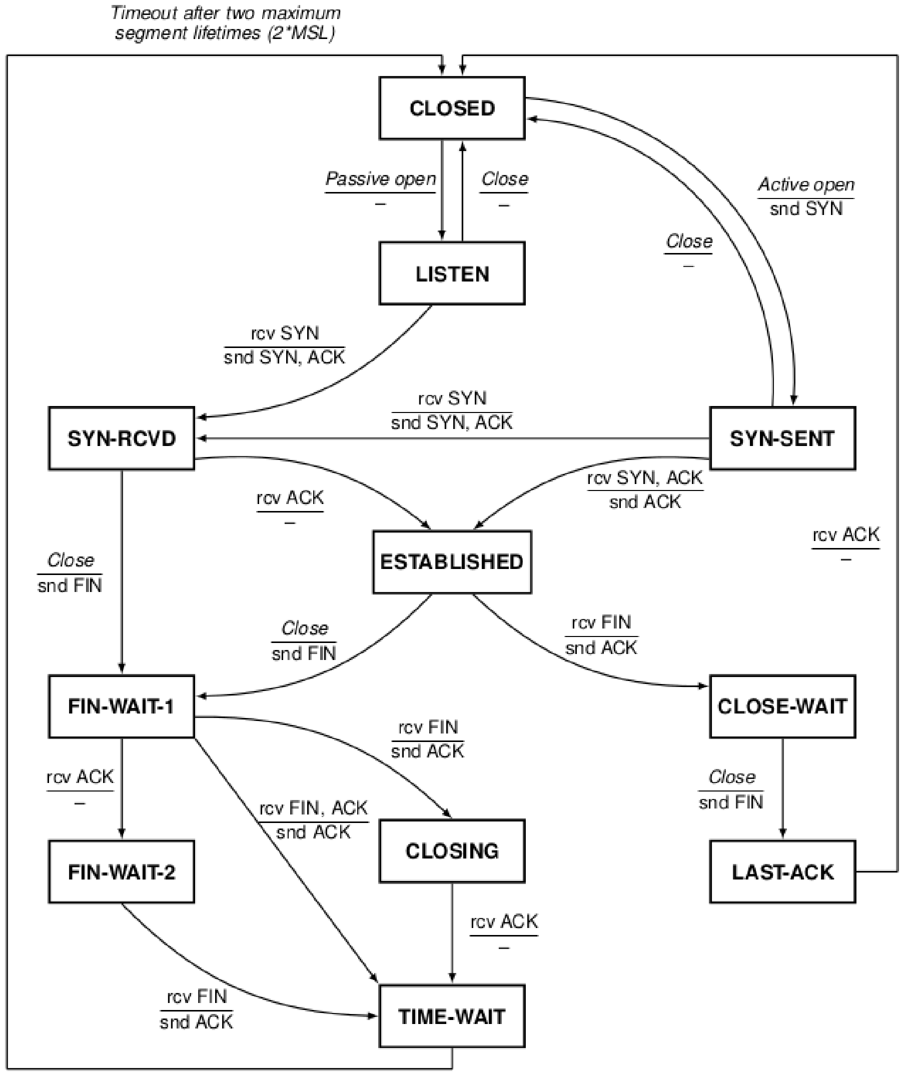
- Defined by a norm written in English: RFC 793 (84 pages).

TCP protocol (II) - State machine

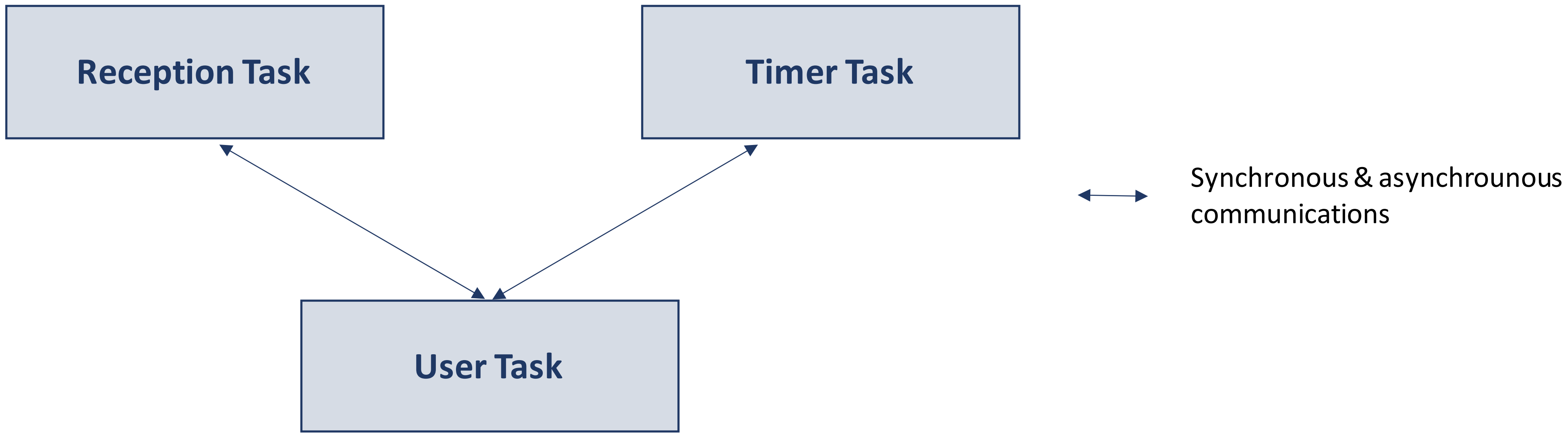
A state machine controls the lifetime of the connection.

The nodes are the states of the TCP connection.

The edges correspond to the actions that can be performed on the TCP.



TCP protocol (III) - Multi tasks model



Model adopted in CycloneTCP

=> We have developed a concurrency model to simulate the actions that can be done by the reception and the timer tasks.

Rewriting the TCP user functions

- Verification that the TCP user functions implementation respects their specifications.
- Rewriting the TCP in SPARK with contracts extracted from RFC 793.

- A function `TCP_Change_State` is called for each change of state and a contract ensures that only allowed transitions are performed.

TCP_Change_State

```
procedure Tcp_Change_State
   (Sock      : in out Not_Null_Socket;
    New_State : in     Tcp_State)
   with
     Pre =>
       (if New_State = TCP_STATE_LISTEN then
          Sock.State = TCP_STATE_CLOSED
       elsif New_State = TCP_STATE_SYN_SENT then
          Sock.State in TCP_STATE_CLOSED
                      | TCP_STATE_LISTEN
       elsif New_State = TCP_STATE_SYN_RECEIVED then
          Sock.State in TCP_STATE_CLOSED
                      | TCP_STATE_LISTEN
                      | TCP_STATE_SYN_SENT
       elsif New_State = TCP_STATE_ESTABLISHED then
          (...)),
     Post =>
       (if New_State = TCP_STATE_CLOSED then
          (if Sock.State'Old = TCP_STATE_LAST_ACK or else
              Sock.State'Old = TCP_STATE_TIME_WAIT
          then
             Model (Sock) = (Model (Sock)'Old with delta
                             S_State => TCP_STATE_CLOSED)
          else
             Model (Sock) = (Model (Sock)'Old with delta
                             S_State => TCP_STATE_CLOSED,
                             S_Reset_Flag => True)
          )
       else
          Model (Sock) = (Model (Sock)'Old with delta
                          S_State => New_State));
```

If the transition is not allowed, GNATprove emits a warning telling that it cannot prove the precondition when `TCP_Change_State` is called.

Modeling the concurrency

- User functions wait for incomming segments. It releases a mutex and allows incomming segments to be processed by the *Reception Task* until the waited event happens.
- For that, we modelized the effects of the reception of an incomming segment.
- We extracted a contract thanks to KLEE from the C Code.
- We add ghost functions in SPARK that respect this contract and that model the effect of the of the reception of a segment.

Using KLEE

```
// Create a fake incoming segment
TCPheader *segment = malloc(sizeof(TCPheader));
klee_make_symbolic(segment, sizeof(segment), "seg");
klee_assume(segment->flag <= 31);
// Create the socket
Socket *sock = malloc(sizeof(Socket)), oldSock;
klee_make_symbolic(sock, sizeof(sock), "sock");
memcpy(&oldSock, sock, sizeof(Socket));

// Call the function to process the segment
tcpProcessSegment(sock, segment);

// Check the expected postcondition
klee_assert(
    (oldSock.state == TCP_STATE_ESTABLISHED) ?
        sock->state == TCP_STATE_ESTABLISHED ||
        sock->state == TCP_STATE_CLOSE_WAIT ||
        sock->state == TCP_STATE_CLOSED :
    (oldSock.state == ...) ? ... )
)
```

User functions contracts

```
Pre => Sock.S_Type = SOCKET_TYPE_STREAM and then
       Sock.State /= TCP_STATE_LISTEN,
Post =>
 (if How in SOCKET_SD_SEND | SOCKET_SD_BOTH and then
     Sock.State'Old = TCP_STATE_CLOSED
 then
    Error = ERROR_NOT_CONNECTED)
 and then
 (if Error = NO_ERROR then
    (if How = SOCKET_SD_SEND then
       (if Sock.State'Old = TCP_STATE_SYN_SENT then
          Model (Sock) = Model (Sock)'Old
        elsif Sock.State'Old = TCP_STATE_CLOSE_WAIT then
          Model (Sock) = (Model (Sock)'Old with delta
             S_State => TCP_STATE_CLOSED)
        else
          Model (Sock) = (Model (Sock)'Old with delta
             S_State => TCP_STATE_FIN_WAIT_2) or else
          Model (Sock) = (Model (Sock)'Old with delta
             S_State => TCP_STATE_TIME_WAIT) or else
```

Possible entry states of the function

Effects that the function produces

Hardening the User's API

- Error when the `Error` code has not been tested.
- Catch incorrect order of calls

```
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),
  Post =>
   (if Sock.S_Type = SOCKET_TYPE_STREAM then
     (if Error = NO_ERROR then
       Sock.S_Remote_Ip_Addr = Remote_Ip_Addr)
    else
      Sock.S_Remote_Ip_Addr = Remote_Ip_Addr)
```

```
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)
```

Bugs captured

We captured two bugs:

- A memory leak
- A violation of the TCP protocol. An incorrect transition has been found thank to the function `TCP_Change_State`.

They have been corrected.

Results

25 % of the functions that implement the TCP protocol have been rewritten in SPARK = All the user functions

Function	Number of ASM instructions C	SPARK	Δ num. of instr.
TCP_Listen	18	11	−39%
TCP_Accept	153	167	+9%
TCP_Connect	103	114	+10%
TCP_Send	94	124	+31%
TCP_Receive	113	153	+35%
TCP_Shutdown	107	122	+14%
TCP_Abort	42	50	+19%

Conclusion & future work

- We have proved the absence of run-time error in a part of the TCP implementation and the conformance with the protocol.
- Go to further, we can rewrite more code in SPARK:
  - Use the RecordFlux DSL to parse incomming segments,
  - Rewrite more TCP functions in SPARK
  - Rewrite other protocols in SPARK to be sure that the code is correct