UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

REVISIÓN DE LA DESCRIPCIÓN GENERAL DEL PROTOCOLO DE COMUNICACIÓN SCHC Y PERFIL SIGFOX UTILIZANDO MODEL CHECKING

TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERA CIVIL EN COMPUTACIÓN

VALERIA ELISA VALDÉS RÍOS

PROFESOR GUÍA
JAVIER BUSTOS JIMENEZ

PROFESORA CO-GUÍA:
SANDRA CÉSPEDES UMAÑA

MIEMBROS DE LA COMISIÓN:
TOMÁS BARROS ARANCIBIA
RODRIGO ARENAS ANDRADE
MARTA BARRÍA MARTÍNEZ

SANTIAGO DE CHILE
2022

RESUMEN DE LA TESIS PARA OPTAR AL GRADO DE
MAGÍSTER EN CIENCIAS DE LA INGENIERÍA, MENCIÓN COMPUTACIÓN
MEMORIA PARA OPTAR AL TÍTULO DE INGENIERA CIVIL EN COMPUTACIÓN
POR: **VALERIA ELISA VALDÉS RÍOS**
FECHA: 2022
PROF. GUÍA: JAVIER BUSTOS JIMENEZ
PROF. CO-GUIA: SANDRA CÉSPEDES UMAÑA

REVISIÓN DE LA DESCRIPCIÓN GENERAL DEL PROTOCOLO DE COMUNICACIÓN SCHC Y PERFIL SIGFOX UTILIZANDO MODEL CHECKING.

La Internet de las Cosas o *Internet of Things* (IoT) permite conectar a Internet a dispositivos que no fueron originalmente diseñados para estarlo, permitiendo que estos puedan compartir datos a la nube. En este contexto, se crean las redes de bajo consumo y gran cobertura llamadas redes LPWAN de su definición en inglés *Low Power Wide Area Networks*. Estas redes buscan conectar dispositivos IoT de bajo consumo en grandes áreas geográficas. Para cumplir con una gran cobertura, las redes LPWAN deben sacrificar latencia y tasa de datos, por lo tanto, en estas redes se tiene una gran latencia, una baja tasa de datos y una baja unidad de transmisión máxima (MTU), restringiendo el número y tamaño de mensajes que un dispositivo puede enviar por estas redes. Debido a la restricción de MTU las redes LPWAN no soportan de forma nativa el direccionamiento IP, ya que esto implicaría un gran uso de memoria en los dispositivos y una mayor tasa de datos en la red.

La *Internet Engineering Task Foce* (IETF) diseña y publica en 2020 en el documento RFC 8724 [16] el protocolo *Static Context Header Compression and Fragmentation* (SCHC), el cual comprime y fragmenta encabezados de paquetes IPv6 para poder ser enviados en redes LPWAN, permitiendo así que dispositivos de bajo consumo conectados a redes LPWAN puedan enviar paquetes IPv6 a pesar de las restricciones que estas redes tienen.

Esta tesis presenta la revisión del protocolo SCHC mediante la técnica basada en modelos Model Checking. Se presentan los modelos para los tres modos de transmisión del protocolo. La revisión busca mostrar que los modos de transmisión necesitan ser más específicos para evitar que el protocolo tenga vulnerabilidades y errores.

Agradecimientos

Primero, agradezco a mi familia, por su constante apoyo y comprensión que me permitió finalizar mis estudios.

Agradezco al lab, por todas las oportunidades que me ha brindado y por permitirme realizar en conjunto este trabajo.

Gracias a mis amigos, por todo el apoyo, por todos esos días de reunirnos a avanzar en nuestros trabajos de título y sobre todo, gracias por las risas.

Finalmente, pero no menos importante, a mis mascotas Flor, Cejas y Pepa, quienes me acompañaron durante todas esas jornadas de escritura y sin que lo supieran se volvieron una parte importante de este trabajo.

Esta tesis ha sido apoyada por el Proyecto ANID FONDECYT Regular 1201893 y por ANID Proyecto Basal FB0008.

Tabla de Contenido

Índice de Ilustraciones

Capítulo 1

Introducción

La Internet de las cosas o *Internet of Things* (IoT) consiste en la conexión y comunicación de objetos cotidianos que tradicionalmente no estaban conectados a Internet. En este contexto, las redes *Low-Power Wide Area Network LPWAN* son utilizadas para conectar una alta densidad de dispositivos IoT. Los dispositivos conectados a redes LPWAN tienen la característica de que sus baterías no se cambiarán de forma periódica o no se reemplazarán. Las redes LPWAN corresponden a un grupo de tecnologías inalámbricas de red que se caracterizan por su capacidad de cubrir una gran área geográfica, tienen una baja tasa de datos, un bajo consumo energético y una alta latencia, es decir, pueden cubrir áreas geográficas grandes pero el tiempo que transcurre desde que un mensaje es enviado hasta que es recibido, es alto.

Debido a las características de las redes *LPWAN*, los dispositivos conectados están limitados en el número de mensajes que pueden enviar por hora o por día y el tamaño de los mensajes que envían y reciben también está limitado por el tipo de tecnología utilizado en la red.

Para poder conectarse a Internet sin necesidad de *middleboxes* y así enviar mensajes a las distintas aplicaciones que procesarán los datos enviados, los dispositivos conectados a las redes *LPWAN* deberían ser capaces de enviar paquetes mediante el protocolo IPv6, pero debido a la restricción en el tamaño de los mensajes que transitan por estas redes, los paquetes IPv6 no son soportados de forma nativa por redes *LPWAN*.

El protocolo *Static Context Header Compression and Fragmentation o SCHC* diseñado por la *Internet Engineering Task Force* (IETF) tiene como objetivo comprimir los encabezados de los paquetes IPv6 y UDP para poder ser enviados mediante redes LPWAN. Además, fragmenta los paquetes resultantes en caso de que aún con la compresión no puedan ser enviados debido a su tamaño. Así, el protocolo SCHC permite que los dispositivos IoT conectados a redes LPWAN puedan enviar mensajes IPv6 a la aplicación que los procesará, sin requerir de intermediarios.

Dado lo reciente de su definición, los estudios alrededor del protocolo SCHC son escasos y los que hay estudian el rendimiento del protocolo. Esta tesis se enfoca en definir modelos para los modos de transmisión del protocolo SCHC que permiten realizar la verificación del protocolo a través de la técnica de verificación *Model Checking*, para verificar que los modos

de transmisión no presentan *deadlocks*.

1.1. Motivación y antecedentes

El uso de IoT cada vez se expande más, cada vez requiere una mayor cobertura geográfica y por lo tanto su uso está en crecimiento. Se estima que para el año 2025 se encuentren 27 billones de dispositivos conectados [1]. Este crecimiento también se puede ver en la amplia inserción de la IoT, que va desde el uso de objetos cotidianos como relojes, ampolletas y electrodomésticos hasta el uso de sensores que recolectan datos para realizar investigaciones.

En un mundo donde abunda la recolección y las técnicas de procesamiento de datos, es importante asegurarse de que estos datos sean correctamente recolectados y recibidos. La recolección de datos puede ser dificultosa debido a que en muchas ocasiones la fuente puede estar en lugares lejanos como en el caso del estudio de suelos y agua. Además, se debe tener en cuenta que generalmente los dispositivos de recolección tienen restricciones energéticas, por lo que el envío de datos se vuelve un desafío debido a las restricciones que trae estar conectado a Internet desde lugares lejanos.

Las redes LPWAN poseen la característica de poder conectar dispositivos de bajo consumo en una gran área geográfica y permite que los dispositivos que están recolectando datos se conecten a Internet. A medida que más dispositivos están conectados, se vuelve más importante tener una red que asegure que la comunicación entre el dispositivo y las aplicaciones sea robusta y segura.

En [14] se menciona que debido a las restricciones de batería los dispositivos IoT, es difícil hacer un seguimiento en línea de los dispositivos para identificar si están bajo algún ataque, por lo que sugiere realizar verificaciones formales o model checking a los sistemas antes de ser implementados. Diversas investigaciones [7] [10] [17] muestran que actualmente la seguridad en IoT es un tema de gran importancia que debe seguir siendo estudiado.

El survey [22] muestra que entre las técnicas más utilizadas para estudiar la seguridad en IoT se encuentran *model checking*, *process algebra* y *theorem proving*, siendo *model checking* la más utilizada entre las investigaciones consideradas.

Así, con el crecimiento de IoT y la masificación de su uso es importante asegurarse que los protocolos de comunicación utilizados funcionen de manera correcta y que no tengan vulnerabilidades.

1.2. Definición del problema

Para que los dispositivos IoT puedan conectarse de forma nativa a Internet, necesitan soportar el protocolo IPv6, pero para que las redes LPWAN tengan un bajo consumo y alcancen una amplia cobertura, los dispositivos están restringidos en el tamaño de los paquetes

[1] https://iot-analytics.com/number-connected-iot-devices/

que pueden enviar.

Debido a la restricción en el ancho de banda las redes LPWAN ocurre que por si solas no tienen la capacidad de comprimir y fragmentar los paquetes IPv6, impidiendo que los dispositivos puedan conectarse a Internet de forma nativa.

SCHC funciona como una capa de adaptación entre las capas de red y enlace, permitiendo a los dispositivos comprimir y fragmentar los encabezados de paquetes IPv6, permitiendo así a los dispositivos IoT enviar paquetes IPv6 [16].

El protocolo SCHC propone tres modos de transmisión para los paquetes resultantes luego de la compresión y fragmentación, llamados Fragmentos SCHC. Estos modos corresponden a *No-ACK*, *ACK-Always* y *ACK-On-Error*, se diferencian en la forma de enviar y recibir los fragmentos. El protocolo recomienda el uso de alguno de estos tres modos según el tipo y cantidad de datos que se quieran enviar.

SCHC ha sido implementado para las tecnologías Sigfox [24], LoraWAN [9] y Narrowband IoT [19], donde cada implementación define un perfil específico. Cuando se encuentra una situación anómala que la descripción general del protocolo no ha abordado, cada perfil define la manera en que maneja esta situación, en lugar de tener una solución general que se pueda aplicar a todos los perfiles, por lo tanto, el escenario ideal correspondería a que la descripción general del protocolo indicara las situaciones anómalas que pudieran ocurrir, para evitar errores en las implementaciones.

Debido al incremento del uso de redes LPWAN, estándares como SCHC son implementados para permitir la conexión nativa de los dispositivos a Internet. SCHC fue publicado como estándar en Abril de 2020 y los estudios realizados hasta la fecha corresponden a medidas del rendimiento del protocolo. Sin embargo, no se ha demostrado mediante verificaciones formales ni verificaciones basadas en modelos que el protocolo no presenta errores en el envío y recepción de paquetes IPv6 sobre la red LPWAN.

1.3. Hipótesis

Al analizar los modos de transmisión del protocolo SCHC mediante el modelado y revisión utilizando la técnica de verificación *model checking*, se pueden exponer defectos en su especificación que pueden derivar en errores y vulnerabilidades en su implementación.

1.4. Objetivos

1.4.1. Objetivos Generales

- Esta tesis tiene como objetivo general realizar la revisión del modelo del protocolo de comunicación SCHC a través de la técnica de verificación *model checking*.

1.4.2. Objetivos Particulares

- Modelar los modos de transmisión del protocolo SCHC a través de máquinas de estados. Para cada modo de transmisión se debe modelar la máquina de estados para el proceso que envía datos y para el proceso que los recibe.
- Identificar propiedades en el protocolo que se quieren verificar, como ejemplo, las propiedades pueden corresponder al orden de llegada de los fragmentos, su integridad o la asusencia de *deadlocks*.
- Realizar la verificación de las propiedades mediante *model checking*.
- Determinar errores en el protocolo y proponer correcciones mediante restricciones en las propiedades verificadas.

1.5. Metodologías y herramientas

Siguiendo la metodología sugerida en [2], se estudia el sistema y sus requerimientos que en este caso corresponde al estudio del protocolo. Se modelan los modos de comunicación como máquinas de estado y se definen las propiedades a verificar. Luego se implementan las máquinas de estado y se realiza el model checking. Finalmente, se analizan los resultados obtenidos luego de realizar la verificación.

Para realizar estos pasos la metodología se divide en cuatro etapas: Investigación, Implementación, Ejecución y Análisis de resultados.

Investigación

Esta etapa consiste en la investigación del protocolo de comunicación SCHC, donde se estudia el contexto en donde se utiliza este protocolo. Esta etapa también considera el estudio del protocolo descrito en el documento *RFC 8724* [16] y su implementación en la tecnología Sigfox descrita en [24], donde se describen los tres modos de transmisión y los formatos de los distintos tipos de fragmentos que se pueden enviar.

Se revisa la documentación correspondiente al *model checker* Spin y se estudia el lenguaje de programación Promela. Además, se realiza una búsqueda de herramientas que faciliten y ayuden al modelado.

Implementación

Esta etapa tiene como objetivo crear los modelos de las máquinas de estado. Para esto, se crean máquinas de estado que representen el comportamiento del protocolo SCHC en sus tres modos de transmisión, para el programa que envía datos y para el que los recibe, en total, se modelan seis máquinas de estado.

Además, se modelan las estructuras de datos necesarias para representar la transmisión de fragmentos, que corresponde al modelado de los distintos tipos de fragmentos y el modelado de los canales de comunicación. Se definen mediante lógica temporal la propiedad que verifica que el protocolo no tiene *deadlocks*.

En esta etapa se implementan los programas que representan la comunicación del transmisor y del receptor en los tres modos de transmisión.

Ejecución

Esta etapa tiene como objetivo realizar la verificación de propiedades mediante *model checking*. Se ejecutan los programas implementados anteriormente y se obtienen los diagramas que representan las máquinas de estado finales, además, al ejecutar los programas se obtienen estadísticas como el número total de estados, el tiempo de cómputo y el uso de memoria.

Al ejecutar los programas y en caso de que el *model checker* encuentre errores, este genera archivos que permiten replicar el escenario en el cual se encuentra el error, por lo tanto, en esta etapa también se obtienen los errores encontrados.

Análisis de resultados

Esta etapa tiene como objetivo determinar si hay errores en el protocolo SCHC. Se revisan los resultados generados por el *model checker*, si este encontró errores, se ejecutan los archivos que permiten replicar el error encontrado.

Si el error encontrado por el *model checker* corresponde a un error en el protocolo entonces, la definición del protocolo debe ser modificada para que esto no ocurra.

En cambio, si el error encontrado corresponde a uno en el modelo, es decir, el modelo no representa correctamente la descripción del protocolo, se modifica el modelo para que represente al protocolo y se vuelve a ejecutar el *model checker*.

1.5.1. Herramientas de software y hardware

Software

Para realizar el *model checking* se utiliza el *model checker* Spin [12] (versión 6.5.1). Spin está diseñado para verificar sistemas distribuidos modelados como programas en el lenguaje Promela. Este lenguaje permite el uso de variables compartidas entre los programas ejecutados. Además, permite el envío de mensajes entre los programas a través de pilas. Esta última característica permite simular comunicación síncrona y asíncrona entre los programas ejecutados. Spin no permite ejecutar modelos que no estén en el lenguaje Promela.

Por si solo Spin no tiene una interfaz gráfica para su ejecución, por lo que se utiliza el *software* jSpin (versión 5.0), una interfaz que simplifica el proceso de ejecución del *model checker*, en donde todos los comandos necesarios para realizar el *model checking* son ejecutados por el *software*.

Hardware

Los programas son ejecutados en un MacBook Air (2017) con un procesador i5 con 6 GB de DDR3 RAM (1600 MHz) y un disco duro sólido con macOs Big Sur 11.5.2 como sistema operativo.

Capítulo 2

Marco Teórico y Estado del Arte

La primera parte de este capítulo muestra el marco teórico que permite entender las tecnologías LPWAN y su funcionamiento. También se presenta la técnica de verificación *model checking*, cómo se realiza y cómo se interpretan sus resultados.

La segunda parte del capítulo muestra el estado del arte tanto en *model checking*, como los análisis previos realizados al protocolo SCHC.

2.1. Marco Teórico

2.1.1. Low-Power Wide-Area Network (LPWAN)

En el contexto de IoT, cada vez se necesita una mayor cobertura geográfica para la cantidad de dispositivos que están conectados y las tecnologías inalámbricas disponibles no logran cumplir este objetivo. Las redes LPWAN definidas en [8] se crean para solucionar este problema y son diseñadas para conectar dispositivos que están distribuidos en grandes áreas geográficas y que requieren un bajo consumo energético.

Las redes LPWAN permiten a los dispositivos conectados comunicarse en un rango de decenas de kilómetros. Como consecuencia del alto rango geográfico, las redes LPWAN tienen una baja tasa de datos, entre los 0,3 kbps a los 50 kbps y una alta latencia. Por lo general, los dispositivos conectados a estas redes son dispositivos de bajo consumo los cuales tienen baterías que duran años y que no serán reemplazadas. Además, debido a las restricciones de la red los dispositivos solo pueden enviar y recibir un número determinado de mensajes.

Algunas tecnologías LPWAN son *LoRaWAN™, Narrowband IoT (NB-IoT) y Sigfox*. Sus descripciones están a continuación:

- LoRaWAN™: Corresponde a un estándar desarrollado bajo la modulación LoRa patentada por la empresa *Semtech*. Esta tecnología permite redes auto gestionadas que disminuyen el costo de mantención. LoRaWAN opera en la banda de radio de uso in-

dustrial, científico y médico (banda ISM). Además, soporta comunicación bidireccional y seis velocidades de transmisión, en donde hay tres tipos de dispositivos, estos se diferencian en el tiempo de recepción que están activos.

- Narrowband IoT (NB-IoT): Esta tecnología es distribuida por compañías de operadores móviles, ya que coexiste con redes LTE. A pesar de compartir su arquitectura con estas redes, NB-IoT está modificada para disminuir el consumo de energía. Posee una tasa de datos máxima de 20 kbps para mensajes de subida y 200 kbps para mensajes de bajada.
- Sigfox: Es una tecnología inalámbrica de pago que ofrece la infraestructura, su mantenimiento y backend. Opera en la banda ISM y los dispositivos están limitados a enviar 140 mensajes diarios con carga útil de 12 bytes y pueden recibir cuatro mensajes diarios con carga útil de 8 bytes.

La figura 2.1 muestra la arquitectura general de una red LPWAN, la cual consiste en dispositivos finales correspondientes a dispositivos IoT que se comunican con *Radio Gateways* a través de tecnologías LPWAN. Varios dispositivos finales pueden estar conectados a un mismo *Radio Gateway*. El *Radio Gateway* a la vez se conecta a un *Network Gateway* a través de IP. En este punto los dispositivos se pueden conectar a Internet, permitiendo que los datos enviados lleguen a un servidor de aplicación o *Application Server*. Este último es el encargado de la capa de aplicación y transmite los mensajes de respuesta a los dispositivos finales. El *Network Gateway* también se comunica con servidor LPWAN-AAA, que es el responsable de la autorización y autenticación de los usuarios.

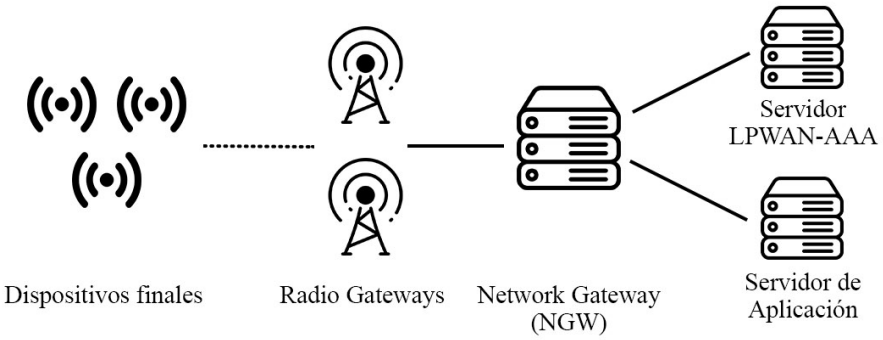


Figura 2.1: Arquitectura LPWAN

2.1.2. Model Checking

Con el paso del tiempo y el avance de la tecnología, la sociedad y su funcionamiento comienza a depender más y más en sistemas computacionales, llevando incluso a que objetos de uso cotidianos dependan de *softwares*. En la actualidad podemos encontrar soluciones computacionales para problemas de diversos ámbitos, como transporte, medicina, agricultura, energía, logística, etc.

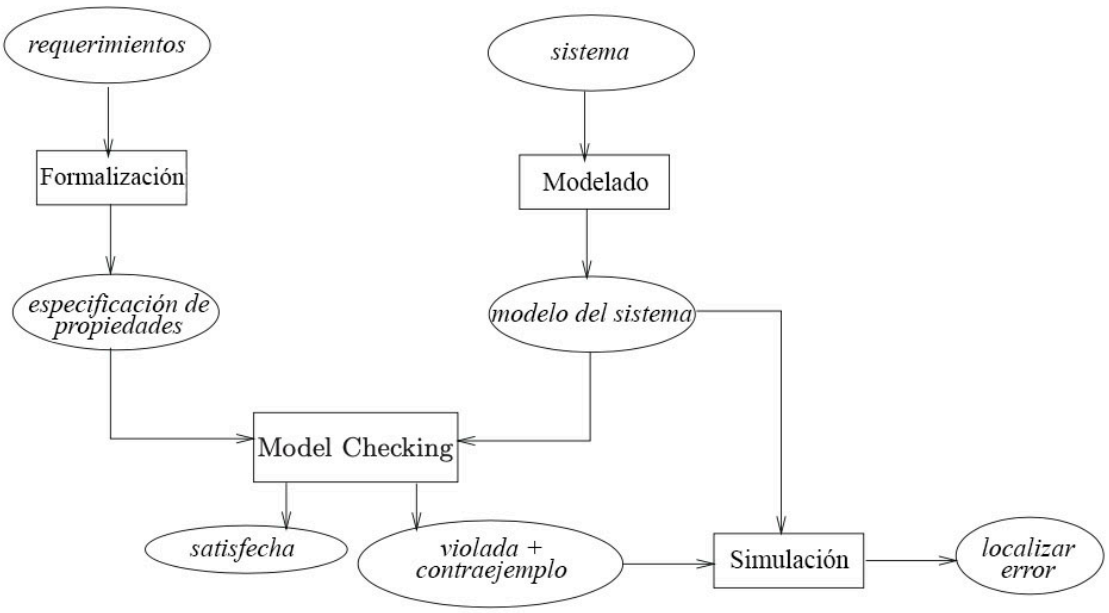


Figura 2.2: Esquema de acercamiento al model checking [2]

Es así como los sistemas computacionales cada vez se vuelven más complejos y cada vez están más presentes en actividades críticas de la sociedad, es por esto que se vuelve una necesidad tener sistemas de calidad que sean tolerantes a fallas y errores.

Para asegurar el buen funcionamiento de los sistemas computacionales existen técnicas de verificación que permiten verificar propiedades, identificar y prevenir fallas. Existen diversas formas de verificación de *software* como la revisión por pares o inspección y el testeo.

Los métodos formales buscan verificar propiedades en un sistema a través de herramientas matemáticas. La verificación formal no solo permite realizar la verificación sobre *software*, sino que también sobre *hardware*. El *model checking* es una técnica de verificación que permite verificar ciertas propiedades de un modelo de un sistema mediante la inspección de todos los estados posibles del modelo.

La figura 2.2 muestra un acercamiento al *model checking*, en donde a partir de requerimientos se formaliza la especificación de las propiedades que se quieren verificar. El sistema se debe modelar y junto con la especificación de las propiedades se puede realizar el *model checking*. Si la propiedad se satisface, el proceso termina, de lo contrario, se entregará un contraejemplo mostrando el escenario en donde no se satisface la propiedad. Con el modelo del sistema y el contraejemplo se puede realizar una simulación para localizar el error.

Modelo del sistema

El modelo del sistema que se desea verificar es un requisito para realizar la verificación mediante *model checking*. Generalmente, los sistemas son modelados como autómatas finitos que describen el comportamiento del sistema, donde los estados del autómata corresponden a los posibles estados del sistema y las transiciones describen como el sistema pasa de un

estado a otro. Los modelos son luego implementados en el lenguaje de programación del *model checker* que se esté utilizando [2] [6].

Especificación de propiedades

Las propiedades que se quieran verificar deben ser precisas y no deben ser ambiguas. Generalmente, son descritas mediante lógica temporal lineal (LTL). Esta es una extensión de la lógica proposicional tradicional que incluye operadores para referirse a ciertos comportamientos del sistema, como lo son [2] [3]:

- $\Box$: Operador unario que indica que una fórmula siempre se cumple.
- $\Diamond$: Operador unario que indica que una fórmula eventualmente se cumple.
- $\cup$: Operador binario *hasta*. La fórmula $\alpha_1 \cup \alpha_2$ se interpreta como: "α_1 se cumple hasta que se cumple α_2"

Algunas de las propiedades que se pueden verificar son la ausencia de *deadlocks*, propiedades que tienen relación a invariantes y respuestas a solicitudes. Entre los tipos de propiedades se encuentran [2] [3]:

- De seguridad o *Safety*: Estas indican que nada "malo" ocurrirá, es decir, que la propiedad siempre se cumple.
- Propiedades *Liveness*: Indican que algo "bueno" ocurrirá eventualmente, es decir, que la propiedad se cumplirá en algún estado del sistema.
- Propiedades de justicia o *Fairness*: Las propiedades o restricciones de tipo *fairness* se utilizan para descartar computaciones o escenarios que son considerados irrazonables para el sistema modelado.

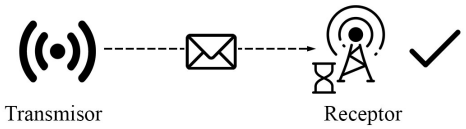
Comunicación Síncrona

Cuando el sistema que se quiere verificar puede ser dividido en subsistemas que interactúan, la forma en la que interactúan también debe estar incluida en el modelo. Si los subsistemas interactúan enviándose mensajes, el tipo de comunicación debe ser modelado, el cual puede corresponder a comunicación asíncrona o síncrona. En el caso de protocolos de Internet, la comunicación es síncrona y la literatura [2] [11] recomienda modelar este tipo de comunicación utilizando canales de tamaño 0.

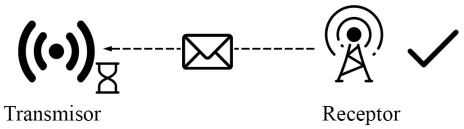
Un canal es una pila FIFO de tamaño fijo, en particular, en un canal de tamaño 0 no se pueden almacenar elementos, por lo tanto, para poder enviar un mensaje por un canal de tamaño 0, al momento de enviar el mensaje se necesita que algún programa esté escuchando por ese canal y así recibir el mensaje.

La figura 2.3 muestra cuatro escenarios de transmisión de un mensaje por un canal de tamaño 0. En las figuras, el reloj de arena indica que el transmisor o el receptor están escuchando por el canal de comunicación esperando un mensaje.

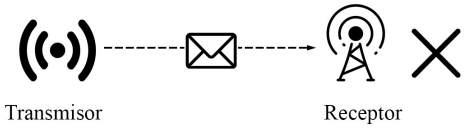
En la figura 2.3a el transmisor envía un mensaje al receptor que está en modo de espera, por lo tanto, la transmisión se realiza correctamente. De la misma forma en la figura 2.3b, la transmisión del mensaje también se realiza correctamente, la diferencia es que en este caso el mensaje es enviado desde el receptor al transmisor. En la figura 2.3c la transmisión del mensaje no se puede realizar debido a que el receptor no está escuchando por el canal. De la misma forma, en la figura 2.3d el receptor no puede transmitir correctamente el mensaje, debido a que el transmisor no está esperando un mensaje.



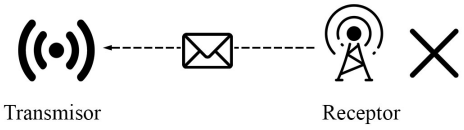
(a) Transmisión correcta



(b) Transmisión correcta



(c) Transmisión errónea



(d) Transmisión errónea

Figura 2.3: Envío de mensajes por canal de tamaño 0

La figura 2.4 muestra un caso de envío de mensajes por un canal de tamaño 0 en donde tanto el transmisor como el receptor intentan enviar un mensaje por el canal pero ninguno lo logra ya que ninguno está escuchando por el canal. Este ejemplo sirve para notar que una parte del sistema puede enviar un mensaje o escuchar por un canal a la vez, pero no las dos acciones al mismo tiempo, por lo tanto, el model checker reconocerá este escenario como un error.

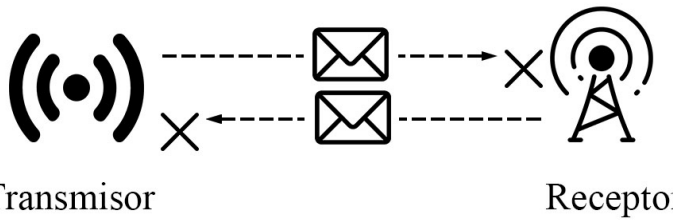


Figura 2.4: Caso especial de envío de mensajes por canal de tamaño 0

2.2. Estado del Arte

2.2.1. Model Checking para protocolos de comunicación

La técnica de verificación model checking es utilizada tanto para verificar *software* como para verificar *hardware*. En esta sección se mostrarán estudios que utilizan esta técnica para verificar protocolos de comunicación y estructuras que facilitan la verificación.

En [13] se verifica a través de model checking el protocolo de comunicación I^2C utilizando Promela y Spin para modelar las capas del protocolo. Esta investigación busca resolver el problema de incompatibilidad de algunos dispositivos presentando un framework que permite verificar la correcta interacción entre dispositivos.

En [23] se verifica la seguridad del protocolo DNS bajo el ataque de cache poisoning. Para realizar la verificación, se definen las máquinas de estados finita extendida (EFSM), la cual extiende la máquina de estados para describir comportamientos dinámicos mediante el uso de variables y pre-condiciones.

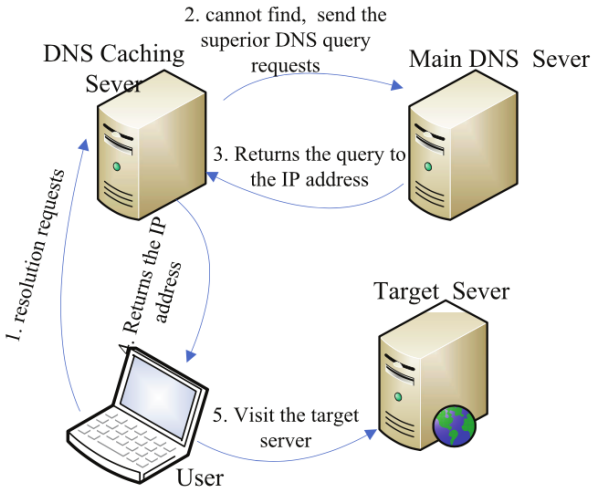
Una EFSM es definida como una tupla $(S, s_0, V, M_V, P, M_P, I, O, T)$ donde S corresponde al conjunto finito de estados, s_0 es el estado inicial, $s_0 \in S$; V es el conjunto finito de variables internas con rango D_V; M_V es el conjunto de valores iniciales de las variables en V, donde cada elemento puede ser expresado como una tupla (s, v), $s \in S$, $v \in D_V$; P es el conjunto de parámetros de entrada y salida; M_P es el conjunto inicial de los valores de las variables en P, donde cada elemento puede ser expresado como una tupla (p, u), $p \in I \cup O$, $u \in D_P$, D_P es el rango de los parámetros de entrada y salida; I es el conjunto de símbolos de entrada; O es el conjunto de símbolos de salida; T es el conjunto finito de transiciones.

Cada transición $t \in T$ es definida como una tupla $(s, x, y, g_P, g_E, op, e)$ donde s y t corresponden al estado inicial y final de la transición respectivamente ; g_P son las condiciones de entrada y salida que determinan si se ejecuta la transición; g_E son las condiciones de las variables que determinan si se ejecuta la transición; x e y son los símbolos de entrada y salida respectivamente; *op* corresponden a las operaciones de salida, es decir, las operaciones que se ejecutarán al pasar del estado s al t.

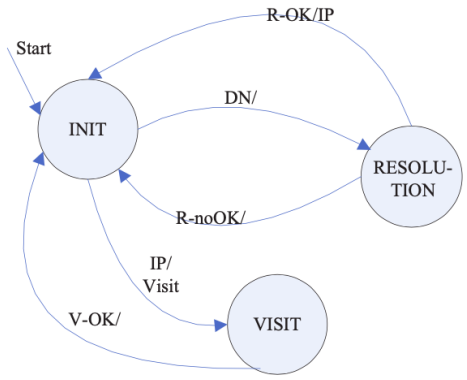
El modelo del protocolo DNS utilizado en [23] corresponde a una versión simplificada del protocolo, que elimina detalles que no son relevantes para el ataque, pero que sigue reflejando el comportamiento del protocolo bajo un ataque de cache poisoning. La figura 2.5a muestra el proceso de resolución de nombre de dominio DNS y la figura 2.5b muestra la máquina de estados finita extendida que representa la resolución de nombre.

En [4] proponen el uso de máquinas de estados dinámicas, las cuales consisten en máquinas de estado complejas que pueden iniciar nuevos procesos que activan otras partes del sistema. Así, las máquinas de estado dinámicas pueden realizar instancias de máquinas de estado paramétricas y permiten el uso de estructuras de datos más complejos. Además, permiten modelar procesos asíncronos y síncronos a través de pseudo nodos. La definición completa de las máquinas de estado dinámicas se encuentra detallada en el Anexo A.

Las máquinas de estado dinámicas separan el flujo del control del sistema correspondiente



(a) Resolución de nombre de dominio en DNS



(b) EFSM que representa el protocolo DNS

Figura 2.5: Modelo del protocolo DNS [23]

a los estados y sus transiciones del flujo de los datos que corresponden a los tipos de datos y estructuras de datos, permitiendo modelar sistemas más complejos. En particular, en [4] se utilizan las máquinas de estado dinámicas para modelar un sistema de control de trenes.

Como extensión al trabajo anterior, en [5] proponen un algoritmo para implementar las máquinas de estado dinámicas en el lenguaje de programación Promela. El objetivo de este algoritmo es utilizar buenas prácticas para evitar explosiones de estado e indicar varios estados finales y así facilitar la comprensión de los resultados entregados por el model checker.

Los nodos de las máquinas de estado dinámicas pueden ser:

- Nodo: Nodo básico de la máquina de estado
- Nodo entrante: Pseudo nodo inicial de la máquina de estado. Una máquina puede especificar múltiples nodos entrantes, que corresponden a diferentes condiciones iniciales.
- Nodo inicial: Nodo inicial predeterminado, se utiliza cuando no hay un nodo entrante especificado.
- Nodo final: Nodo de salida. Una máquina puede especificar múltiples nodos finales, que corresponden a distintas condiciones finales.
- Box: Nodo que modela la activación paralela de máquinas de estado asociados al nodo Box. Una transición a un Box representa la activación de la máquina de estado, mientras que una transición que sale de un nodo Box corresponde al fin de la ejecución de esa máquina de estado.
- Fork: Pseudo nodo de control que modela la activación de nuevos procesos, estos puedes ser síncronos o asíncronos.
- Join: Pseudo nodo de control utilizado para sincronizar el término de un proceso o para forzar su término.

El algoritmo propuesto en [5] consiste en convertir los pseudo nodos en nodos básicos a través de transformaciones que dependen de la naturaleza del pseudo nodo, así se eliminan las jerarquías entre nodos. Esta conversión es necesaria debido a las diferencias entre el lenguaje utilizado para definir la máquina de estados y el lenguaje de programación.

Los siguientes pasos del algoritmo describen como implementar la máquina de estado resultante luego de las transformaciones para obtener modelos en Promela. Se propone un código estructurado en una sección inicial donde se declaran los tipos, variables y canales por los que se comunicarán los procesos. En el anexo B, el código B.1 muestra el formato del programa principal que se encarga de manejar las comunicaciones entre los procesos y permite simular correctamente las ejecuciones de estos mediante tokens que indican cuando un proceso puede realizar una ejecución.

En el anexo B, el código B.2 muestra el formato para implementar cada máquina de estado que modele el sistema. Este programa considera el caso de que un proceso padre active un proceso hijo y la forma en la que se entrega el token al proceso hijo. Además, muestra como se deben implementar los distintos tipos de estados y sus transiciones.

2.2.2. Análisis de desempeño de SCHC

En la actualidad se han realizado varias investigaciones al protocolo SCHC. Estas estudian el rendimiento del protocolo y los factores que lo afectan. A continuación se presentan algunos de los estudios realizados al protocolo.

En [18], se estudia el protocolo SCHC sobre la tecnología LoRaWAN, en particular, se estudia la relación entre la eficiencia de ocupación del canal de transmisión, el *spreading factor* de LoRa™ y la probabilidad de un error en un mensaje SCHC. En esta investigación se modelan los factores de manera teórica y se comparan con resultados experimentales. Como resultado, se concluye que la eficiencia disminuye cuando la probabilidad de pérdida de un mensaje SCHC aumenta. También se encuentran los *spreading factors* en los cuales se tiene una menor y una mayor eficiencia y finalmente, se concluye que para toda probabilidad, la eficiencia aumenta a medida que el *spreading factor* aumenta.

En [21] se estudia la fragmentación y compresión del protocolo SCHC sobre la tecnología LoRaWAN, en donde se realizan experimentos enviando paquetes en redes reales en una implementación propia. Se analizan los factores de compresión, configuración de la red y tamaño de los fragmentos. Los resultados muestran que al reducir el tamaño del paquete y al reducir el tamaño de los fragmentos se tiene un mejor rendimiento pero se debe considerar que si bien se gana eficiencia, el uso energético es mayor.

En [1] se estudia el protocolo SCHC sobre la tecnología Sigfox, se modela y evalúa el tiempo de transferencia de los paquetes y el número de mensajes de subida y bajada necesarios para enviar por completo el paquete. El modelo teórico de esta investigación asume que no hay pérdida de fragmentos y los experimentos muestran que para los dos escenarios experimentales que fueron considerados, la tasa de pérdida de fragmentos es cercano a 0 %. Los resultados de este estudio muestran que pequeños cambios en el tamaño del paquete puede afectar significativamente el tiempo de transmisión y que bajo ciertas condiciones, a medida que

la tasa de pérdida de fragmentos incrementa puede disminuir el tiempo de transferencia del paquete y que el número de mensajes de subida y bajada no es proporcional a este incremento.

En estas investigaciones previas se considera la probabilidad de error en la transmisión de mensajes y si bien se encuentran pérdidas de paquetes, estas se atribuyen a la naturaleza de la red y no a algún error en el protocolo. Esta tesis sostiene que se pueden encontrar casos de borde en los cuales la transmisión de paquetes no se ejecuta correctamente y que pueden existir vulnerabilidades en el protocolo que aún no han sido estudiadas.

Capítulo 3

SCHC: Static Context Header Compression

En este capítulo se presentan los detalles más relevantes del protocolo SCHC, se describe el proceso de compresión y fragmentación, se presentan los distintos tipos de mensajes que se pueden enviar, se detallan los tres modos de transmisión y se muestran las características del protocolo de acuerdo al perfil Sigfox.

3.1. Compresión y fragmentación

SCHC puede ser considerado como una capa intermedia entre la capas de red, en este caso IPv6, y la capa de enlace correspondiente a la tecnología LPWAN utilizada. En la figura 3.1 se muestra la pila de protocolos que incluye SCHC y que permiten enviar paquetes IPv6 sobre una tecnología LPWAN.

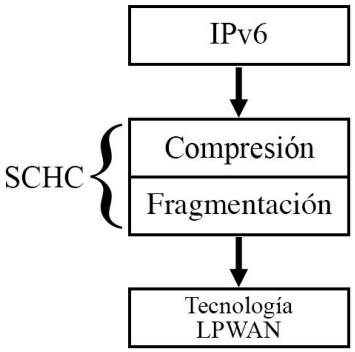


Figura 3.1: Pila de protocolos en donde se utiliza SCHC. [16]

El proceso para enviar y recibir un paquete utilizando el protocolo SCHC en una tecnología LPWAN se muestra en la figura 3.2. Este consiste en comprimir los encabezados del paquete según reglas de compresión. El resultado de la compresión es un *SCHC Packet*. En caso de que el *SCHC Packet* no cumpla los requisitos de tamaño para ser enviado directamente por la red,

se debe realizar la fragmentación de este. El resultado de este proceso son fragmentos SCHC. El receptor debe reensamblar el paquete SCHC en caso de que reciba más de un fragmento y descomprimirlo. Dependiendo del modo de transmisión, el receptor podría mandar un mensaje SCHC ACK que indica qué mensajes recibió correctamente.

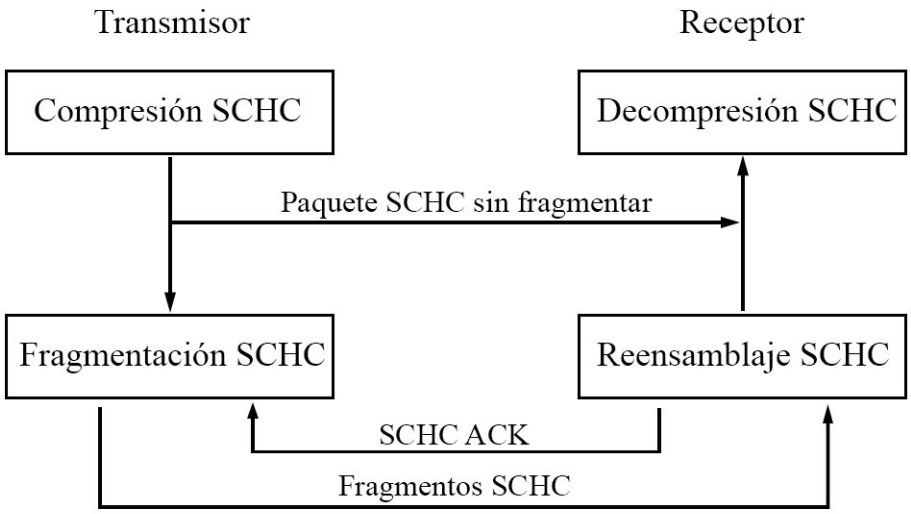


Figura 3.2: Envío y recepción de un paquete SCHC.[16]

3.2. Estructura de los paquetes y tipos

Un paquete SCHC puede ser dividido en partes debido al proceso de fragmentación. Estas partes son denominadas *Tiles*. Un grupo de *tiles* corresponde a una ventana. En la figura 3.3 se muestra un paquete SCHC que tiene tres ventanas de *tiles*, enumeradas desde 0 a 2, donde cada ventana está compuesta por cinco *tiles*, enumerados de forma decreciente desde 4 a 0.

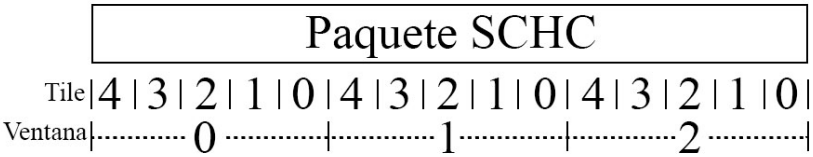


Figura 3.3: Paquete SCHC dividido en tres ventanas que contienen cinco *tiles*. [16]

Para enviar los *tiles* de un paquete, el protocolo envía Fragmentos SCHC desde el transmisor hacia el receptor. Estos fragmentos tienen como estructura general los siguientes encabezados [16]:

- RuleID: Presente en todos los mensajes. Permite identificar un fragmento SCHC. Todos los mensajes provenientes del mismo paquete deben tener el mismo Rule ID.
- Datagram Tag (DTag): Campo utilizado para diferenciar dos mensajes provenientes de distintos paquetes pero que tienen el mismo RuleID. El tamaño de este campo está indicado en bits por el campo T.

- W: Campo opcional, solo está presente cuando se utilizan ventanas de transmisión. Permite identificar el número de ventana al que pertenece el fragmento. El tamaño de este campo está indicado en bits por el campo M.
- Fragment Compressed Number (FCN): Este campo indica el número de secuencia de los *tiles* pertenecientes al fragmento. El tamaño de este campo está indicado en bits por el campo N.
- Reassembly Check Sequence (RCS): Transporta el código de detección de errores. Solo está presente en el último mensaje del paquete. El tamaño de este campo está indicado en bits por el campo U.
- C (Integrity Check): Utilizado en los fragmentos que confirman la recepción de los mensajes. El tamaño de este campo es de 1 bit. El valor 1 indica que los mensajes fueron recibidos correctamente. El valor 0 indica que el chequeo de integridad no se pudo realizar o fue fallido.
- Compressed Bitmap: Presente cuando se usan ventanas. Aparece en los fragmentos de confirmación de los *tiles* para informar en un mapa de bits cuales *tiles* fueron recibidos correctamente y cuales no.

Para lograr la correcta transmisión del paquete SCHC, el contenido es enviado en distintos tipos de mensajes que tienen distintas funciones:

- Fragmento SCHC: Fragmento enviado por el transmisor al receptor, puede transportar uno o más *tiles* del paquete SCHC.
- All-1: Corresponde a un caso particular de un fragmento SCHC donde el valor de todos los bits del campo FCN son 1. Se utiliza para indicar que es el último fragmento del paquete.
- All-0: Corresponde a un caso particular de un fragmento SCHC donde el valor de todos los bits del campo FCN son 0. Se utiliza para indicar que el fragmento corresponde al último de la ventana que se está enviando.
- SCHC ACK: Fragmento enviado por el receptor al transmisor. Permite confirmar cuales *tiles* fueron recibidos correctamente y cuales no.
- SCHC ACK Req: Fragmento enviado por el transmisor al receptor con motivo de solicitar un mensaje ACK para confirmar la recepción de los *tiles* de la ventana indicada en el mensaje ACK Req.
- SCHC Sender-Abort: Fragmento enviado por el transmisor al receptor indicando que la comunicación será terminada.
- SCHC Receiver-Abort: Fragmento enviado por el receptor al transmisor indicando que la comunicación será terminada.

3.3. Modos de transmisión

Esta sección presenta y detalla los tres modos de transmisión del protocolo SCHC, estos se diferencian en la forma en la se envían los mensajes. Para cada modo de transmisión se detalla el comportamiento del transmisor y la del receptor.

3.3.1. No-ACK

El modo de transmisión No-ACK tiene la característica de que la comunicación va únicamente desde el transmisor al receptor. En en este modo, el transmisor no espera confirmación de la recepción de los mensajes. Además, este modo asume que los mensajes llegan en el mismo orden en el que fueron enviados.

En este modo no se utilizan ventanas de envío por lo tanto no se utiliza el campo W del encabezado y no se considera ningún tipo de retransmisión. Además, se recomienda que el tamaño del campo FCN sea de tamaño 1 bit.

Cada perfil debe especificar los valores correspondientes para RuleID, DTag y el tiempo del temporizador de inactividad.

Transmisor

En el modo de transmisión No-ACK el transmisor debe seleccionar según el perfil, el par de campos RuleID y DTag correspondientes al paquete a enviar. Cada fragmento debe contener un *tile* en su *payload* y el transmisor debe enviarlos en el orden en el que aparecen en el paquete SCHC.

Cada fragmento que no contenga el último *tile* del paquete debe tener como valor 0 en el campo FCN. El último fragmento del paquete debe tener como valor 1 en el campo FCN, por lo tanto, el último fragmento enviado corresponde a un fragmento All-1.

El transmisor podría enviar fragmentos SCHC Sender-Abort.

Receptor

Para cada par de valores RuleID y DTag, el receptor debe mantener un *timer* de inactividad. Si no tiene los recursos para hacer esto, entonces debe descartar los mensajes.

- Al recibir un fragmento regular, el receptor debe iniciar un *timer* de inactividad y reensamblar el contenido del fragmento SCHC que recibió.
- Al recibir un fragmento All-1, el receptor debe ensamblar su contenido y realizar el chequeo de integridad. Si el chequeo falla, debe descartar el paquete recibido. Si el chequeo es correcto, el reensamblaje termina.

- Si el *timer* de inactividad expira, el receptor debe descartar el paquete recibido.
- Si el receptor recibe un fragmento SCHC Sender-Abort, podría descartar el paquete recibido.

3.3.2. ACK-Always

Este modo de transmisión asume que los mensajes son entregados en orden, que cada mensaje contiene el mismo número de *tiles* y que existe una forma de que el receptor pueda enviar mensajes al transmisor.

En el modo ACK-Always se utilizan las ventanas de transmisión, es decir, el paquete comprimido es dividido en ventanas de *tiles*. Además, el receptor puede enviar mensajes ACK positivos o negativos al término de recepción de cada ventana.

Cada perfil debe especificar los valores correspondientes para RuleID y DTag, el tamaño de cada ventana, el tiempo del *timer* de inactividad y del *timer* de retransmisión, además, debe especificar el valor de la variable *max_ack_requests* que corresponde al número de veces consecutivas que el transmisor puede solicitar mensajes ACK al receptor o el número de veces consecutivas que el receptor envía mensajes ACK para una misma ventana.

Transmisor

Cada fragmento enviado por el transmisor debe contener exactamente un *tile* en su *payload*, el campo W debe ser el bit menos significativo del número de ventana del paquete que se está enviando y el campo FCN debe ser el índice del *tile* que se está enviando. Los *tiles* se envían según el orden de aparición en la ventana, es decir, el campo FCN es decreciente. La primera ventana que envía el transmisor debe ser la número 0.

Al iniciar el envío de una ventana, el transmisor debe iniciar un contador de intentos en 0. Luego de enviada la ventana, el transmisor entra a una fase de retransmisión, debe iniciar el contador de intentos en 0 e iniciar un *timer* de retransmisión en la espera del mensaje SCHC ACK.

Cuando recibe el mensaje SCHC ACK:

- Si el mensaje indica que se perdieron *tiles*, el transmisor debe volver a enviar todos los *tiles* que han sido reportados como perdidos, debe incrementar el contador de intentos, volver a iniciar el *timer* de retransmisión y esperar el siguiente mensaje SCHC ACK.
- Si la ventana actual no es la última y el mensaje SCHC ACK indica que todos los *tiles* fueron recibidos correctamente, el transmisor debe detener el *timer* de retransmisión y avanzar al envío de la siguiente ventana.
- Si la ventana actual es la última y el mensaje SCHC ACK indica que se recibieron más *tiles* de los que fueron enviados, entonces, el transmisor debe enviar un mensaje SCHC Sender-Abort y puede terminar la transmisión en condición de error.

- Si la ventana actual es la última y el mensaje SCHC ACK indica que todos los *tiles* fueron recibidos correctamente, sin embargo, el chequeo de integridad falló, el transmisor debe enviar un mensaje SCHC Sender-Abort y puede terminar la transmisión en condición de error.
- Si la ventana actual es la última y el mensaje SCHC ACK indica que el chequeo de integridad se realizó correctamente, el transmisor termina la comunicación correctamente.

Si expira el *timer* de retransmisión:

- Si el contador de intentos es menor que *max_ack_requests*, el transmisor debe enviar un mensaje SCHC ACK REQ y debe incrementar el contador de intentos.
- Si el contador de intentos es mayor o igual a *max_ack_requests*, el transmisor debe enviar un mensaje SCHC Sender-Abort y puede terminar la transmisión en condición de error.

En cualquier momento

- Si recibe un mensaje SCHC Receiver-Abort, el transmisor puede terminar la transmisión en condición de error.
- Si recibe un mensaje SCHC ACK con un número de ventana distinto al actual, el transmisor debe descartar e ignorar este mensaje SCHC ACK.

Receptor

Cuando recibe un fragmento con un par de campos RuleID y DTag que no han sido procesados:

- El receptor debe revisar que el valor de DTag no haya sido utilizado recientemente, así se asegura que el fragmento recibido no es un fragmento restante de alguna transmisión anterior. Si este es el caso, el receptor debe ignorar y descartar este fragmento, de lo contrario, el receptor debe iniciar un *timer* de inactividad según lo indica el valor de DTag, iniciar el contador de intentos en 0 e iniciar un contador de ventanas en 0.
- El receptor debe iniciar el proceso de reensamblaje de un nuevo paquete SCHC.
- Si el receptor no cuenta con los recursos para realizar estas operaciones debe enviar al transmisor un mensaje SCHC Receiver-Abort.

Al recibir un mensaje con un par de valores RuleID y DTag siendo procesados, el receptor debe reiniciar el *timer* de inactividad.

El receptor debe iniciar un *bitmap* vacío para la primera ventana de recepción, a continuación entra a una fase de aceptación, en donde:

- Al recibir un fragmento SCHC o un fragmento SCHC ACK REQ con un W diferente al local, el receptor debe ignorar y descartar el mensaje.
- Al recibir un mensaje SCHC ACK REQ con un W igual al local, el receptor debe enviar el mensaje SCHC ACK para la ventana.
- Si recibe un fragmento SCHC con un W igual al local, el receptor debe ensamblar el *tile* recibido basado en la ventana y el número de FCN, luego debe actualizar el *bitmap*.
  - Si el fragmento SCHC corresponde a un mensaje All-0, la ventana recibida no es la última del paquete, el receptor debe enviar un mensaje SCHC ACK para esta ventana y entrar en fase de retransmisión para esta ventana.
  - Si el fragmento SCHC corresponde a un mensaje All-1, entonces la ventana recibida es la última del paquete. El receptor debe realizar el chequeo de integridad y enviar el mensaje SCHC ACK para esta ventana:
    * Si el chequeo de integridad indica que el paquete SCHC ha sido correctamente reensamblado, entonces el receptor entra en fase de "limpieza".
    * Si el chequeo de integridad indica que el paquete SCHC no pudo ser reensamblado correctamente, el receptor entra en fase de retransmisión para esta ventana.

En la fase de retransmisión, si la ventana no es la última del paquete SCHC:

- Si recibe un fragmento que no es All-0 ni All-1 que tiene un W diferente al local, el receptor debe incrementar el contador de ventanas, iniciar un *bitmap* vacío, ensamblar el *tile* recibido y actualizar el *bitmap*, luego entra en fase de aceptación para esta nueva ventana.
- Al recibir un mensaje All-0 con un W distinto al local, el receptor debe incrementar el contador de ventanas e iniciar un nuevo *bitmap*, debe ensamblar el *tile* recibido y actualizar el *bitmap*, además, debe enviar un mensaje SCHC ACK para esta nueva ventana y quedarse en la fase de retransmisión para esta nueva ventana.
- Al recibir un mensaje All-1 con un W diferente al local, el receptor debe incrementar el contador de ventanas, iniciar un nuevo *bitmap*, ensamblar el *tile* recibido y actualizar el *bitmap*. El receptor debe enviar un mensaje SCHC ACK para esta nueva ventana, este mensaje está determinado por esta última ventana, ya que debe realizar el chequeo de integridad.
  - Si el chequeo de integridad indica que el paquete SCHC fue ensamblado correctamente, el receptor entra en fase de "limpieza" para esta nueva ventana.
  - Si el chequeo de integridad indica que el paquete SCHC no pudo ser ensamblado correctamente, el receptor entra en fase de retransmisión para esta ventana.
- Al recibir un fragmento All-1 con un W igual al local, el receptor debe ignorarlo y descartarlo.

- Al recibir un fragmento que no sea All-1 con un W igual al local, el receptor debe ensamblar el *tile* recibido y actualizar el *bitmap*. Si el *bitmap* se llena de 1's, es decir, todos los *tiles* fueron recibidos o si el mensaje recibido corresponde a un All-0, el receptor debe enviar el mensaje ACK para esa ventana.
- Si recibe un mensaje SCHC ACK REQ con un W igual al local, el receptor debe enviar el mensaje ACK correspondiente a la ventana.

En la fase de retransmisión, si la ventana es la última del paquete SCHC:

- Si recibe un mensaje SCHC ACK REQ con un W distinto al local, el receptor debe ignorar y descartar el mensaje.
- Si recibe un mensaje SCHC ACK REQ con un W igual al local, el receptor debe enviar el mensaje ACK para la ventana correspondiente.
- Al recibir un fragmento All-0 con un W igual al local, el receptor debe ignorarlo y descartarlo.
- Al recibir un fragmento con un W igual al local, el receptor debe ensamblar el *tile* y actualizar el *bitmap*. Si el fragmento es All-1, entonces debe realizar el chequeo de integridad:
  - Si el chequeo indica que el paquete fue ensamblado correctamente, entonces el receptor debe enviar un mensaje ACK y entrar en fase de "limpieza".
  - Si el chequeo indica que no se pudo ensamblar correctamente el paquete SCHC, entonces debe enviar un mensaje ACK indicando esta condición.

En la fase de "limpieza":

- Si recibe un mensaje All-1 o un fragmento SCHC ACK REQ con un W igual al local, el receptor debe enviar un mensaje ACK.
- Cualquier otro fragmento debe ser ignorado y descartado.

En cualquier momento, al enviar un mensaje SCHC ACK, el receptor debe incrementar el contador de intentos.

En cualquier momento, al incrementar el contador de ventanas se debe reiniciar el contador de intentos.

Si expira el *timer* de inactividad, si recibe un mensaje SCHC Sender-Abort o si el número de intentos alcanza a *max_ack_requests*, el receptor debe enviar un mensaje SCHC Receiver-Abort y puede terminar el reensamblaje para ese paquete SCHC.

3.3.3. ACK-On-Error

El modo de transmisión ACK-On-Error soporta tecnologías que tienen MTU variable y entrega de paquetes fuera de orden. En este modo también se utilizan ventanas de transmisión y un fragmento SCHC puede tener en su contenido uno o más *tiles* contiguos.

Este modo se caracteriza por el comportamiento del receptor, este no envía fragmentos SCHC ACK para ventanas que han sido correctamente recibidas. Por lo tanto, el transmisor puede seguir con el envío de la ventana siguiente y enviar los *tiles* perdidos después.

La descripción de este modo indica que la transmisión termina cuando:

- El chequeo de integridad indica que el paquete fue recibido correctamente y esto se le ha indicado al transmisor o,
- Se realizaron muchos intentos de retransmisión o,
- El receptor determina que la transmisión ha estado inactiva por mucho tiempo.

Cada perfil debe especificar los valores correspondientes para RuleID y DTag, el tamaño de cada ventana, el tiempo del *timer* de inactividad y del *timer* de retransmisión, además, debe especificar el valor de la variable *max_ack_requests* que corresponde al número de veces consecutivas que el transmisor puede solicitar mensajes ACK al receptor o el número de veces consecutivas que el receptor envía mensajes ACK para una misma ventana.

El perfil también debe indicar si el último *tile* es enviado en un fragmento regular solo, en un fragmento regular con más *tiles* o solo en un fragmento All-1.

Transmisor

El transmisor puede enviar más de un *tile* en su contenido. Si esto ocurre entonces los *tiles* deben ir contiguos según aparecen en el paquete SCHC original. Los *tiles* que no son los últimos deben ir en un fragmento SCHC regular, donde el campo FCN debe ser el índice del primer *tile* en el fragmento y el campo W debe ser el índice de la ventana perteneciente al primer *tile* del fragmento.

En un fragmento All-1, el transmisor debe indicar el campo W como el número del último *tile* en el mensaje. Además, el transmisor debe encargarse de que el último *tile* sea enviado una sola vez, en alguna de las formas mencionadas anteriormente y que se envíe al menos un fragmento All-1, pero que no se envíe el último *tile* de dos maneras distintas.

Para un mismo par de valores RuleID y DTag, el transmisor debe mantener un contador de intentos y un *timer* de retransmisión.

El transmisor debe esperar un fragmento ACK luego de enviar un fragmento All-1 o un fragmento SCHC ACK REQ, además, cada vez que envía alguno de estos fragmentos, el transmisor debe incrementar el contador de intentos y reiniciar el *timer* de retransmisión.

Cuando expira el *timer* de retransmisión:

- Si el contador de intentos es estrictamente menor a *max_ack_requests*, el transmisor debe enviar un fragmento All-1 o un fragmento SCHC ACK REQ con el campo W correspondiente a la última ventana.
- De lo contrario, el transmisor debe enviar un fragmento SCHC Sender-Abort y puede terminar la transmisión en condición de error.

Al recibir un fragmento SCHC ACK con campo W correspondiente a la última ventana del paquete SCHC:

- Si el bit C indica que el paquete SCHC se recibió correctamente, entonces el transmisor termina la transmisión correctamente.
- Si el bit C indica que el paquete SCHC no fue recibido correctamente:
  - Si el perfil indica que el último *tile* debe ser enviado en un fragmento All-1:
    - Si el fragmento SCHC ACK indica que no hay *tiles* perdidos, el transmisor debe enviar un fragmento SCHC Sender-Abort y puede terminar la transmisión en condición de error.
    - Si el fragmento SCHC ACK indica que hay *tiles* perdidos, el transmisor debe enviar fragmentos SCHC con todos los *tiles* que fueron reportados como perdidos. Si el último de estos mensajes no es un mensaje All-1, el transmisor debe enviar enviar un mensaje SCHC ACK REQ con campo W correspondiente a la última ventana.
  - Si el perfil no indica que el último *tile* debe ser enviado en un fragmento All-1:
    - Si el mensaje SCHC ACK indica que no hay *tiles* perdidos, el transmisor debe enviar un mensaje All-1.
    - Si el mensaje SCHC ACK indica que hay *tiles* perdidos, entonces el transmisor debe enviar fragmentos SCHC regular con todos los *tiles* reportados como perdidos. Además, el transmisor debe enviar un mensaje All-1 o un mensaje SCHC ACK REQ con campo W correspondiente a la última ventana.

Al recibir un fragmento SCHC ACK con campo W distinto a la última ventana del paquete SCHC, el transmisor debe enviar fragmentos SCHC regulares con todos los *tiles* reportados como perdidos. Luego, puede enviar un fragmento SCHC ACK REQ con campo W correspondiente a la última ventana.

Receptor

Tal como en el modo Ack-Always, al recibir un fragmento SCHC con un par de campos RuleID y DTag que no han sido procesados, el receptor debe revisar que el valor para DTag

no haya sido utilizado recientemente para otro RuleID y así asegurarse que el fragmento recibido no es un fragmento restante de alguna transmisión anterior.

Luego de asegurarse que el fragmento corresponde a un nuevo paquete SCHC, el receptor debe comenzar el proceso para ensamblar este nuevo paquete. El receptor debe iniciar el *timer* de inactividad e iniciar un contador de intentos en 0. Si el receptor no cuenta con los recursos para realizar estas operaciones, entonces debe enviar un fragmento SCHC Receiver-Abort.

Al recibir cualquier fragmento con un par de campos RuleID y DTag que están siendo procesados, el receptor debe reiniciar el *timer* de inactividad perteneciente a ese par de campos.

Cuando recibe un fragmento SCHC, el receptor debe determinar que *tiles* han sido recibidos basados en el largo del contenido del fragmento, el campo W y el campo FCN del fragmento SCHC.

Al recibir un fragmento SCHC ACK REQ o un fragmento All-1:

- Si el receptor sabe de alguna ventana que tenga *tiles* perdidos debe responder con un mensaje SCHC ACK con campo W correspondiente al número menor de ventana que tenga *tiles* perdidos.
- En caso contrario, el receptor no sabe de ventanas que tengan *tiles* perdidos:
  - Si ha recibido al menos un *tile*, debe responder con un mensaje SCHC ACK con campo W correspondiente al mayor número de ventana para la cual ha recibido *tiles*.
  - De lo contrario, debe responder con un mensaje SCHC ACK para la ventana número 0.

Un perfil puede especificar otras circunstancias en las cuales el receptor envía mensajes SCHC ACK.

Al enviar un mensaje SCHC ACK, el receptor debe aumentar el contador de intentos.

Al recibir un mensaje All-1, el receptor debe realizar el chequeo de integridad cada vez que se prepara para enviar un fragmento SCHC ACK para la última ventana.

Al recibir un fragmento SCHC Sender-Abort, el receptor puede terminar la conexión en condición de error.

Al expirar el *timer* de inactividad o si el contador de intentos es mayor a *max_ack_requests*, el receptor debe enviar un fragmento SCHC Receiver-Abort y puede terminar en condición de error.

El reensamblaje del paquete SCHC termina cuando:

- Recibe un fragmento SCHC Sender-Abort, o
- El *timer* de inactividad ha expirado, o

- El contador de intentos es mayor a *max_ack_requests*, o
- Al menos un mensaje All-1 ha sido recibido y el chequeo de integridad se realiza correctamente.

3.4. Perfil Sigfox

La descripción del protocolo SCHC da flexibilidad para que las tecnologías LPWAN lo puedan implementar según sus necesidades. Es así como para la tecnología Sigfox se describe en el documento [24] el funcionamiento del protocolo SCHC sobre esta tecnología, muestra la arquitectura de las redes Sigfox e indica los valores de las variables que la descripción general deja para definición en el perfil.

La arquitectura de la tecnología Sigfox está basada en la arquitectura LPWAN de la figura 2.1. En la figura 3.4 se presenta la estructura de la arquitectura Sigfox, en donde se muestra que el dispositivo Sigfox se conecta al *Sigfox Base Station* que representa al *Radio Gateway*. El *Sigfox Base Station* se comunica con el *Sigfox Network* que representa al *Network Gateway* de la arquitectura LPWAN. En caso de una red global, los *Radio Gateways* son distribuidos en múltiples países donde se provee del servicio. El *Network Gateway* corresponde a una sola entidad que conecta todos los *Sigfox base stations*.

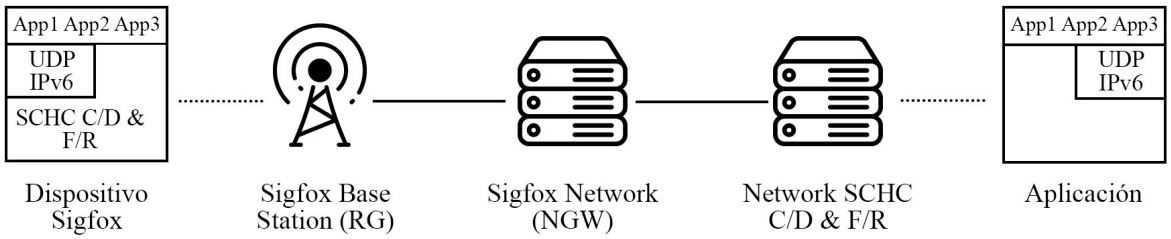


Figura 3.4: Arquitectura Sigfox. [24]

El proceso de envío de un mensaje *uplink*, desde el dispositivo a la aplicación, comienza cuando el dispositivo Sigfox tiene datos recolectados que necesita enviar. El mensaje pasa por la compresión y fragmentación SCHC (SCHC C/D & F/R) para reducir el tamaño de los encabezados. El mensaje resultante es enviado como un *frame Sigfox* a la *Sigfox Base Station*, donde el mensaje es enviado al *Sigfox Network*, este representa al *Network Gateway* (NGW). El *Network Gateway* se comunica con el *Network SCHC C/D & F/R* para la compresión/decompresión y/o fragmentación/reensamblaje. El *Network SCHC C/D & F/R* comparte el mismo set de reglas de compresión que el dispositivo Sigfox. Finalizada la decompresión y/o reensamblaje, el mensaje puede ser enviado a través de Internet a uno o más servidores LPWAN de aplicación. El proceso de envío de un mensaje es bidireccional, por lo tanto, para enviar un mensaje en la otra dirección se aplican los mismos principios, teniendo en consideración que el envío de mensajes *downlink*, es decir, aquellos que son enviados desde la aplicación al dispositivo Sigfox, son respuesta a uno *uplink* y no pueden ser enviados en cualquier momento.

Los mensajes enviados desde un dispositivo Sigfox a la red son entregados por el *Sigfox Network* (NGW) al *Network Sigfox SCHC C/D + F/R* y contienen la siguiente información:

- ID del dispositivo: Identificador global, único, asignado al dispositivo y está incluido en el encabezado Sigfox de cada mensaje.
- Payload: Contenido del mensaje.

Además, contienen información dada por la red, la cual corresponde a:

- Timestamp
- Ubicación (opcional)
- RSSI (opcional)
- Temperatura del dispositivo (opcional)
- Batería del dispositivo (opcional)

La figura 3.5 muestra la estructura de un mensaje SCHC enviado sobre la tecnología Sigfox, donde el mensaje SCHC puede ser un paquete completo (comprimido) o un fragmento.

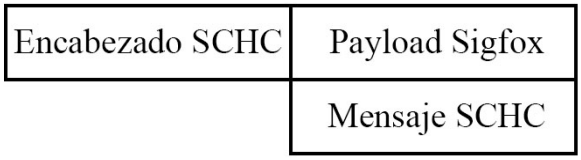


Figura 3.5: Mensaje SCHC en Sigfox.[24]

3.4.1. ACK-On-Error

El perfil Sigfox del modo ACK-On-Error completa las decisiones que quedan pendientes en la descripción general de este modo de transmisión.

El perfil indica que los mensajes de bajada (del receptor al transmisor) solo pueden ocurrir cuando el transmisor explícitamente indica que puede recibir mensajes. Esto ocurre cuando luego de enviar todos los mensajes de subida (desde el transmisor al receptor), el transmisor comienza/abre una ventana para recepción.

Si el transmisor no recibe ningún mensaje en esta ventana de recepción, este cierra la ventana y vuelve a su modo normal.

Cuando el transmisor recibe un mensaje de bajada, genera un mensaje para indicar que recibió el mensaje anterior y lo envía a la red mediante el protocolo de radio Sigfox y este es reportado en el *Sigfox Network backend*.

Para indicar que la ventana de recepción está abierta, el transmisor lo hace a través del último fragmento de cada ventana, es decir, cuando el valor del campo FCN es 0 el cual corresponde a un fragmento All-0 o un fragmento All-1 para el último del paquete.

El perfil también indica que cada fragmento debe contener exactamente un *tile*, que el último *tile* del paquete debe ser enviado en un fragmento All-1.

El modo ACK-On-Error tiene dos variantes en este perfil:

- Single-byte SCHC header: Recomendado para paquetes de tamaño medio a grande de hasta 300 bytes que necesitan ser enviados de manera segura. El tamaño del encabezado SCHC es de 8 bits. En esta versión el tamaño de una ventana corresponde a siete *tiles*, el valor de la variable *max_ack_requests* es 5 y el tiempo de los *timers* de inactividad y retransmisión dependen de la aplicación.
- Two-byte SCHC header: Recomendado para paquetes muy grandes de hasta 2250 bytes que necesitan ser enviados de manera confiable. En esta versión el tamaño de una ventana corresponde a 31 *tiles*, el valor de la variable *max_ack_requests* es 5 y el tiempo de los *timers* de inactividad y retransmisión dependen de la aplicación.

Capítulo 4

Modelos

En este capítulo se presentan los modelos de las máquinas de estado que representan el comportamiento del transmisor y del receptor para los tres modos de transmisión. En particular, para el modo ACK-On-Error los modelos corresponden a la descripción indicada por el perfil Sigfox. Además, en cada modo se muestra la estructura de los fragmentos.

4.1. No-ACK

4.1.1. Fragmentos

Los fragmentos en el modo No-ACK son definidos por la estructura *Fragment* que se muestra en el código 4.1, donde el atributo FCN es un *bit* que tiene valor 0 siempre, excepto para el último fragmento del paquete, donde tiene valor 1. El atributo *abort* se utiliza para indicar que el fragmento corresponde a un mensaje SCHC Sender-Abort.

Listing 4.1: Formato de fragmentos para el modo No-ACK

```
typedef Fragment{
    bit FCN;
    bit abort;
}
```

4.1.2. Transmisor

En la figura 4.1 se muestra la máquina de estados que representa el comportamiento del transmisor en el modo No-ACK. El transmisor comienza con un contador de los *tiles* que debe enviar, envía uno a uno los *tiles* en fragmentos All-0 hasta que llega al último, el cual es enviado en un fragmento All-1. En la descripción de este modo se indica que el transmisor podría enviar un fragmento Sender-Abort y terminar la comunicación, en este modelo se considera esa posibilidad.

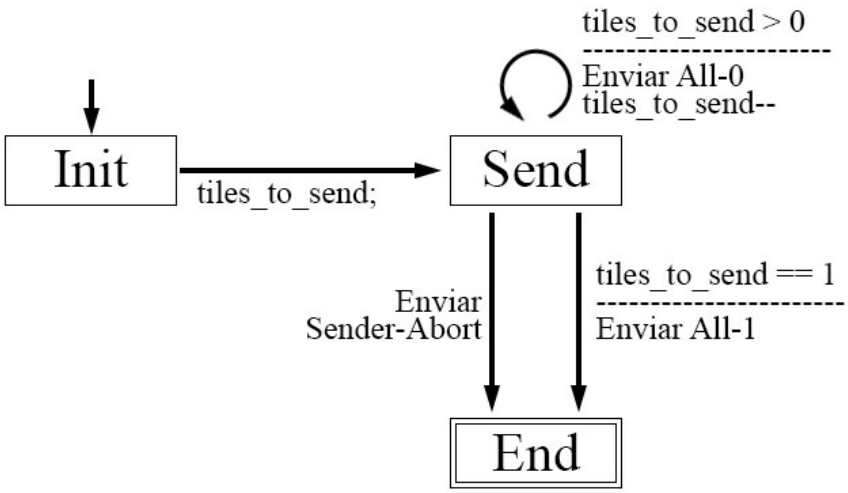


Figura 4.1: Máquina de estados del transmisor en modo No-ACK.

4.1.3. Receptor

La figura 4.2 muestra la máquina de estados para el receptor en el modo No-ACK. Este recibe y ensambla los *tiles* recibidos hasta que al recibir el último *tile* en un fragmento All-1 realiza el chequeo de integridad, si este es correcto, entonces termina la comunicación. Si el chequeo de integridad es incorrecto, entonces termina en estado de error. También puede terminar en estado de error si expira el *timer* de inactividad o si recibe un fragmento Sender-Abort.

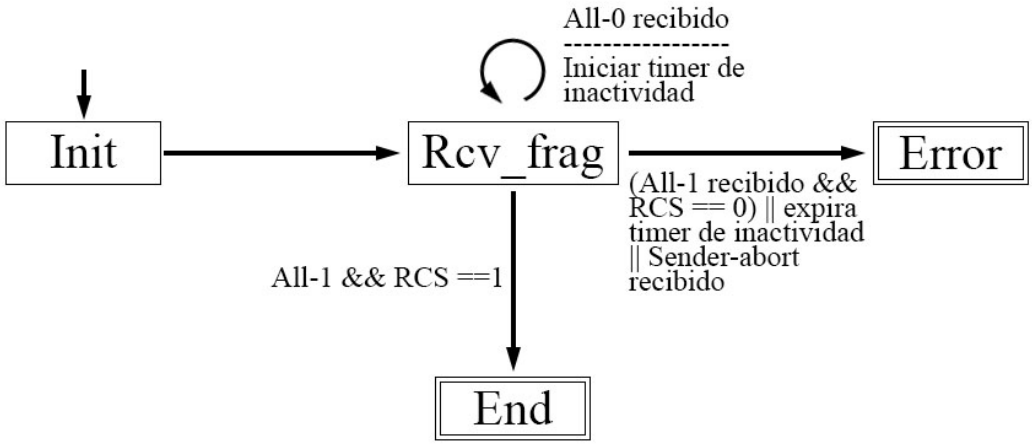


Figura 4.2: Máquina de estados del receptor en modo No-ACK.

4.2. ACK-Always

4.2.1. Fragmentos

Los fragmentos en el modo ACK-Always son definidos por la estructura *Fragment* que se muestra en el código 4.2, donde el atributo *bm* corresponde al *bitmap*, que es modelado como un entero que representa la cantidad de *tiles* recibidos. El campo *c* es un *bit* que solo se utiliza en un fragmento ACK. Los campos *ackRequest*, *abort* y *all*1 se utilizan para indicar que los fragmentos corresponden a fragmentos SCHC ACK REQUEST, SCHC Abort y All-1 respectivamente.

Listing 4.2: Formato de fragmentos para el modo ACK-Always

```
typedef Fragment{
    int bm;
    int FCN;
    bit W;
    bit c;
    bit ackRequest;
    bit abort;
    bit all1;
}
```

4.2.2. Transmisor

La figura 4.3 muestra la máquina de estados que representa al transmisor en el modo ACK-Always. Esta inicia sus variables locales y envía una ventana de *tiles*, luego espera la confirmación por parte del receptor, si este indica que se perdieron *tiles*, entonces reenvía los *tiles* perdidos, en caso contrario, sigue con el envío de la ventana siguiente. La comunicación termina cuando transmisor recibe el mensaje que indica que todos los *tiles* de la última ventana fueron recibidos correctamente o cuando muchos intentos para que el receptor responda se han realizado o cuando ha ocurrido un error en el lado del receptor.

La figura 4.4 muestra un posible modelo de la máquina de estados que reenvía los *tiles* perdidos, esta corresponde a una máquina de estados paramétrica que tiene como parámetro el fragmento ACK enviado por el receptor. La máquina de estados inicia una variable local que representa al mapa de bits enviado por el receptor, mientras este mapa de bits no sea igual al que tiene el transmisor, es decir, mientras no se hayan enviado todos los *tiles* perdidos, envía uno a uno los *tiles*. Cuando ambos mapas de bits son iguales, el proceso termina.

4.2.3. Receptor

La máquina de estados que representa al receptor se muestra en la figura 4.5. Comienza iniciando las variables locales y luego recibe los fragmentos enviados por el transmisor, al

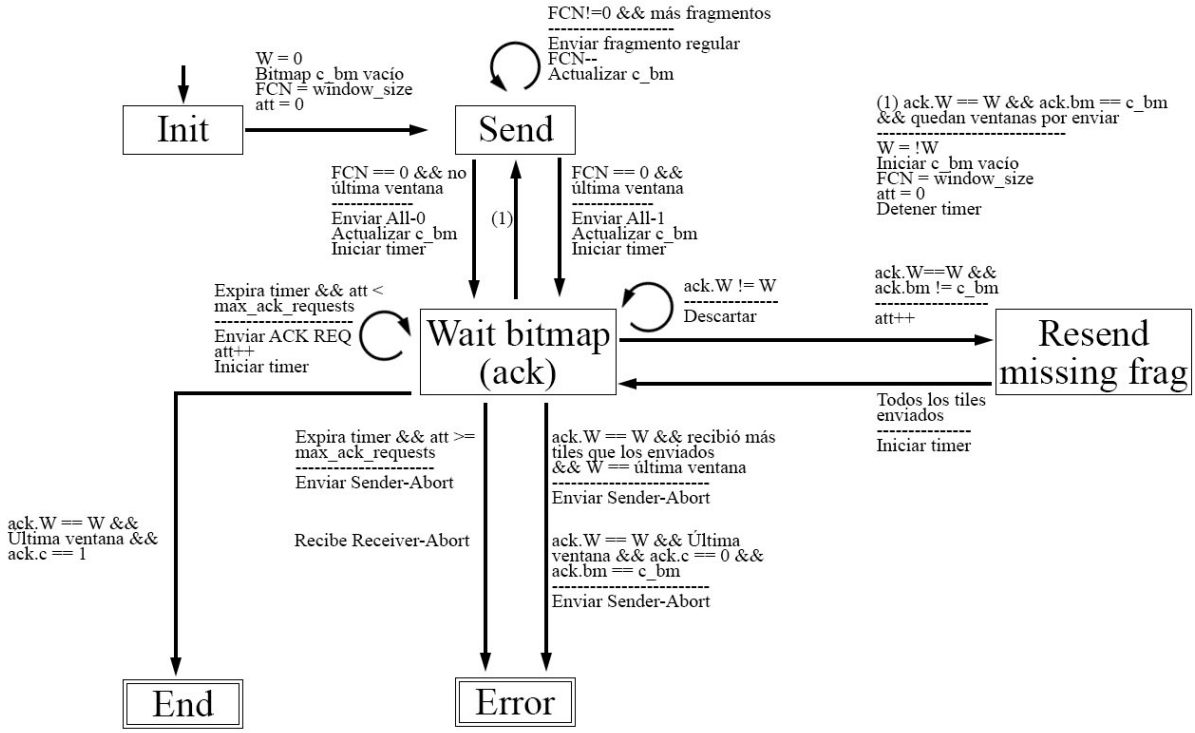


Figura 4.3: Máquina de estados del transmisor en modo ACK-Always.

recibir un fragmento All-0 o All-1 el receptor envía el fragmento ACK correspondiente. Si hay *tiles* perdidos, entonces estos serán enviados por el transmisor, de lo contrario recibirá la ventana siguiente. En la descripción se indica que el receptor pasa por una etapa de "limpieza" antes de terminar la comunicación y que esta solo se puede terminar si expira el *timer* de inactividad o recibe un fragmento Sender-Abort o se han enviado muchos fragmentos ACK consecutivos.

4.3. ACK-On-Error Perfil Sigfox

4.3.1. Fragmentos

Los fragmentos en el modo ACK-On-Error son definidos por la estructura *Fragment* que se muestra en el código 4.2, donde el atributo *bm* corresponde al *bitmap*, que es modelado como un entero que representa la cantidad de *tiles* recibidos. El campo *c* es un *bit* que solo se utiliza en un fragmento ACK. Los campos *abort* y *all*1 se utilizan para indicar que los fragmentos corresponden a fragmentos SCHC Abort y All-1 respectivamente.

Listing 4.3: Formato de fragmentos para el modo ACK-On-Error

```
typedef Fragment{
    int bm;
    int FCN;
```

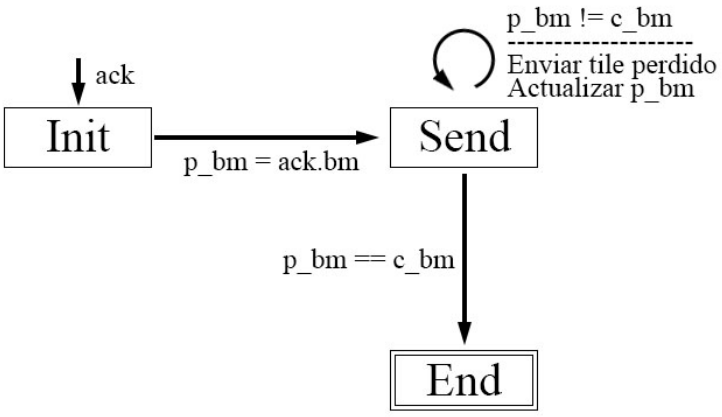


Figura 4.4: Máquina de estados que reenvía tiles perdidos para el modo ACK-Always.

```
    int W;
    bit c;
    bit abort;
    bit all1;
}
```

4.3.2. Transmisor

La figura 4.6 muestra la máquina de estados para el transmisor en el modo ACK-On-Error para el perfil Sigfox. Esta comienza iniciando variables locales y envía los fragmentos correspondientes a la ventana que está enviando. Al terminar de enviar una ventana de *tiles*, abre una ventana de recepción donde espera el fragmento ACK, el tiempo que esta ventana está abierta está definido por un *timer* de retransmisión. Si el *timer* expira y no ha recibido el fragmento ACK el transmisor sigue con el envío de la ventana siguiente. En caso de que reciba el fragmento ACK y este indique que se perdieron *tiles*, entonces reenvía los *tiles* perdidos. La comunicación termina cuando recibe el ACK para la última ventana del paquete y esta indica que todos los *tiles* fueron recibidos correctamente. El transmisor también puede terminar en estado de error si el fragmento ACK para la última ventana indica que hubo un error, o si expira el *timer* de retransmisión y se han realizado muchos intentos, o si recibe un fragmento Receiver-Abort.

La figura 4.7 muestra un ejemplo de la máquina de estados que reenvía los *tiles* perdidos, esta es similar a la figura 4.4 para el modo ACK-Always, pero se diferencia en que si el último *tile* del paquete corresponde a un *tile* perdido, este no se envía, ya que es enviado al terminar el proceso de reenvío. Esta modificación se hace para evitar que el último *tile* del paquete sea enviado dos veces consecutivas.

4.3.3. Receptor

El comportamiento del receptor se muestra en la figura 4.8, donde el receptor comienza iniciando variables locales y recibe los fragmentos enviados por el transmisor. Al recibir un fragmento All-0 o All-1 el receptor envía un fragmento ACK que indica si ha detectado *tiles* perdidos en la ventana actual o en una anterior, si este es el caso, entonces recibe los *tiles* perdidos. La transmisión termina cuando el receptor recibe un fragmento All-1 y determina que no hay *tiles* perdidos para ninguna ventana. La comunicación también termina cuando expira el *timer* de inactividad, o se han realizados muchos intentos de envío de fragmentos ACK, o cuando recibe un fragmento Sender-Abort. El perfil indica que el receptor solo puede enviar mensajes cuando el transmisor habilita una ventana de recepción, por lo tanto, el receptor no puede enviar mensajes Receiver-Abort en cualquier momento, sino que tiene que esperar a recibir un fragmento All-0 o All-1 que son los que indican que la ventana de recepción está habilitada.

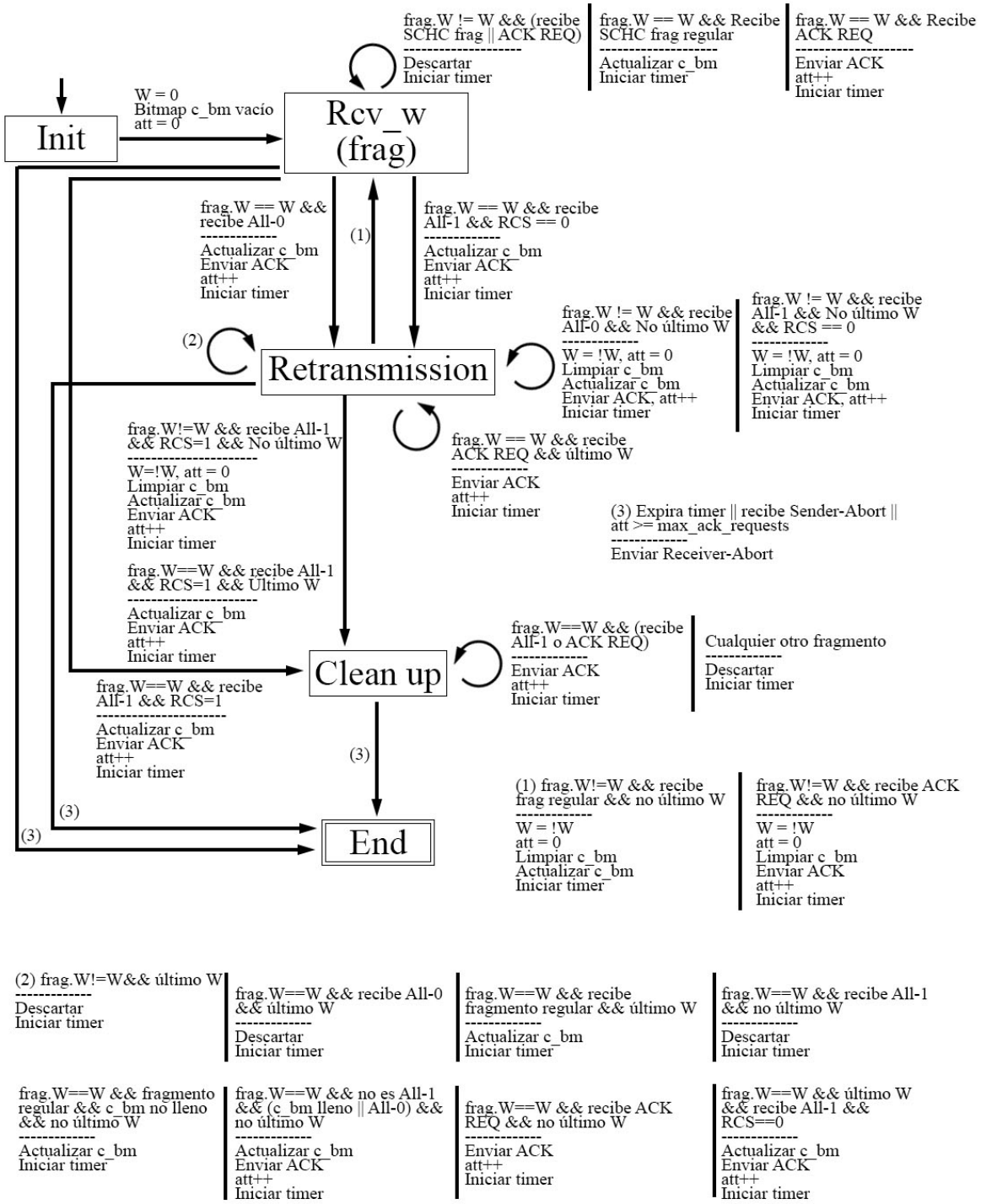


Figura 4.5: Máquina de estados del receptor en modo ACK-Always.

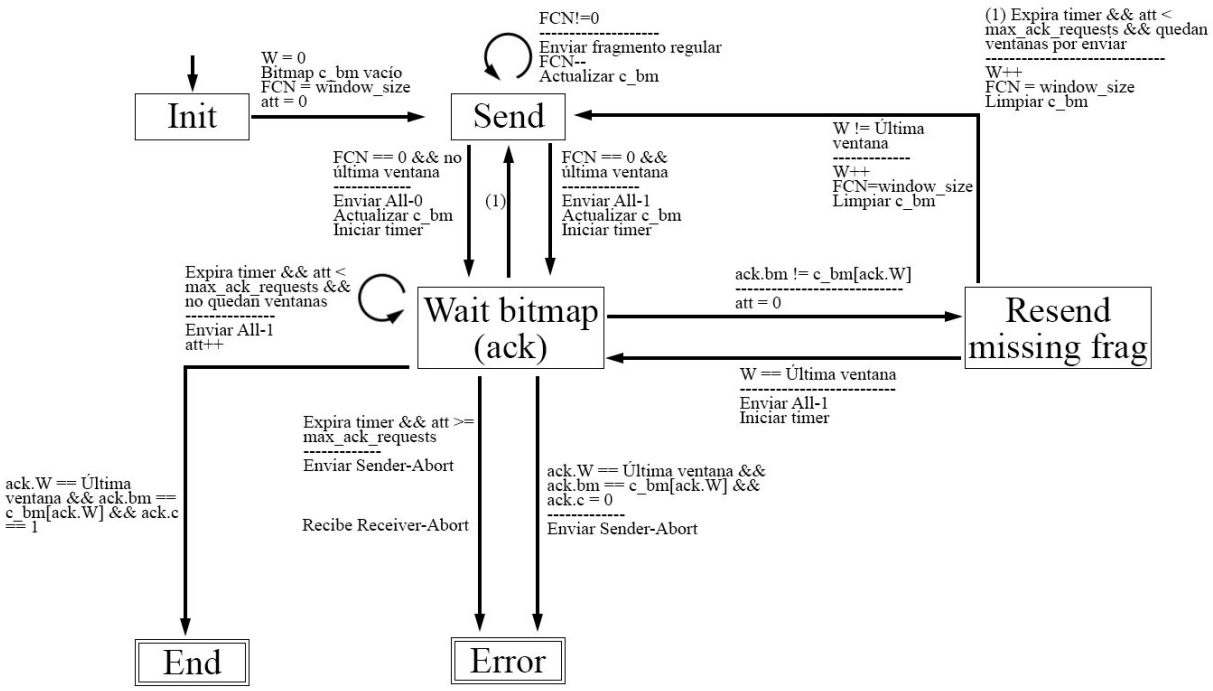


Figura 4.6: Máquina de estados del transmisor en modo ACK-On-Error para el perfil Sigfox.

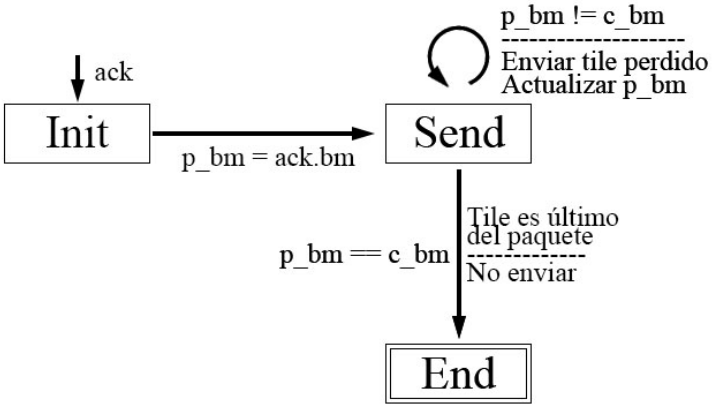


Figura 4.7: Máquina de estados que reenvía tiles perdidos para el modo ACK-On-Error en el perfil Sigfox.

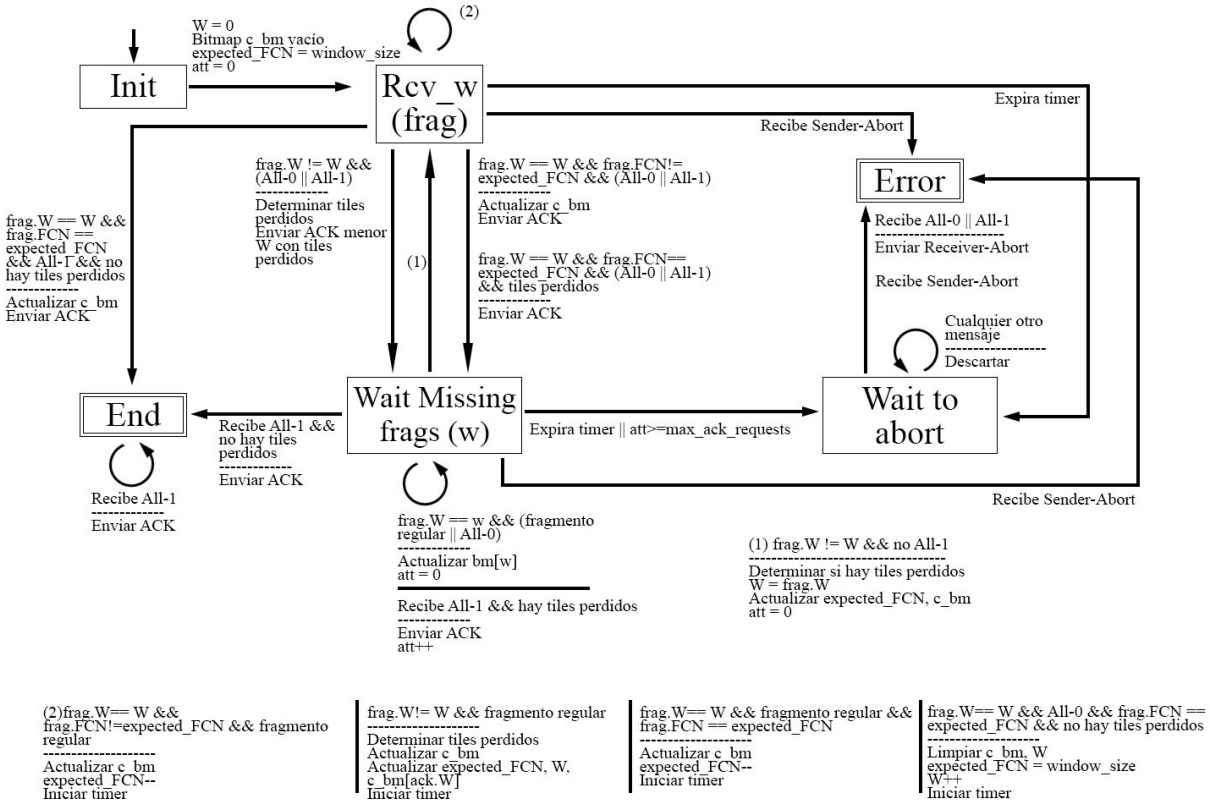


Figura 4.8: Máquina de estados del receptor en modo ACK-On-Error para el perfil Sigfox. La principal diferencia entre la descripción general del protocolo se encuentra al momento de término de envío de una ventana de *tiles*, el perfil Sigfox espera un mensaje ACK, sino lo recibe continúa con el envío de la siguiente ventana, en cambio, la descripción general del protocolo indica que debe enviar un mensaje ACK Request.

Capítulo 5

Resultados

En este capítulo se muestran los resultados obtenidos al realizar la revisión de los modelos presentados en el capítulo anterior utilizando el *model checker* Spin. Para cada modo de transmisión se ejecutan en paralelo las máquinas de estado del transmisor y del receptor.

Además, en los modos ACK-Always para la descripción general y ACK-On-Error para la implementación en Sigfox se presenta una propiedad que busca verificar que no hay errores en transmisiones "amigables" *half duplex*, es decir, se busca verificar que no hay errores en transmisiones de tipo *half duplex* en donde ni el transmisor ni el receptor tienen comportamiento de atacante.

5.1. No-ACK

Para el modo No-ACK se busca verificar la ausencia de *deadlocks*, propiedad que se verifica de forma automática por el *model checker* Spin. Se ejecutan los programas del transmisor del anexo C y el programa del receptor del anexo D en paralelo, donde el transmisor envía 10 *tiles* por el canal *ch*.

El resultado de esta ejecución se muestra en la figura 5.1. La novena línea de los resultados indica que el *model checker* ha encontrado un error.

El escenario en donde se produce el error se muestra en la figura 5.2 y se puede describir como:

1. El transmisor envía el primer *tile*. El receptor inicia su proceso y se encuentra en ventana de recepción (indicado por el reloj de arena), por lo tanto, recibe correctamente el mensaje, lo ensambla e inicia el *timer* de inactividad.
2. El transmisor espera, no envía el siguiente *tile*. El receptor se encuentra en ventana de recepción esperando el siguiente mensaje.
3. En el lado del receptor expira el *timer* de inactividad ya que el transmisor ha tomado

```
1  (Spin Version 6.5.1 -- 31 July 2020)
2  Warning: Search not completed
3                   + Partial Order Reduction
4  Full statespace search for:
5                   never claim                 - (none specified)
6                   assertion violations          +
7                   cycle checks                - (disabled by -DSAFETY)
8                   invalid end states           +
9  State-vector 44 byte, depth reached 17, ••• errors: 1 •••
10       15 states, stored
11        0 states, matched
12       15 transitions (= stored+matched)
13        2 atomic steps
14 hash conflicts:       0 (resolved)
15 Stats on memory usage (in Megabytes):
16    0.001                 equivalent memory usage for states (stored*(State-vector +
17 overhead))
18    0.288                 actual memory usage for states
19  128.000                   memory used for hash table (-w24)
20    0.107                 memory used for DFS stack (-m2000)
21  128.302                   total actual memory usage
22 pan: elapsed time 0 seconds
```

Figura 5.1: Resultados al ejecutar los programas C y D para el modo No-ACK.

mucho tiempo para enviar el fragmento. El receptor cierra su ventana de recepción y termina la comunicación.

4. El transmisor intenta enviar un fragmento, pero este no es recibido ya que el receptor ha terminado la comunicación.

5.2. ACK-Always y ACK-On-Error perfil Sigfox

Para realizar el *model checking* del modo ACK-Always, se ejecutan en paralelo los programas del anexo E para el transmisor y el programa del anexo F para el receptor. Se busca verificar la ausencia de *deadlocks*, propiedad que es verificada de forma automática por el *model checker* Spin.

El transmisor envía dos ventanas de dos *tiles* cada una por el canal *ch*. Por su parte, el receptor envía los fragmentos *SCHC ACK* y *SCHC Receiver-Abort* por el canal *ack_ch*.

La figura 5.3 muestra el resultado de esta ejecución. La línea 9 de esta figura indica que el *model checker* ha encontrado un error.

Para el modo ACK-On-Error en el perfil Sigfox, se ejecutan en paralelo el programa del anexo G para el transmisor y el programa del anexo H para el receptor, buscando verificar la ausencia de *deadlocks*.

En esta transmisión, el transmisor envía tres ventanas de cuatro *tiles* cada una por el canal *ch* y el receptor envía los mensajes *SCHC ACK* y *SCHC Receiver-Abort* por el canal *ach_ch*.

La figura 5.4 muestra el resultado de esta ejecución, en particular, la línea 9 indica que se ha encontrado un error.

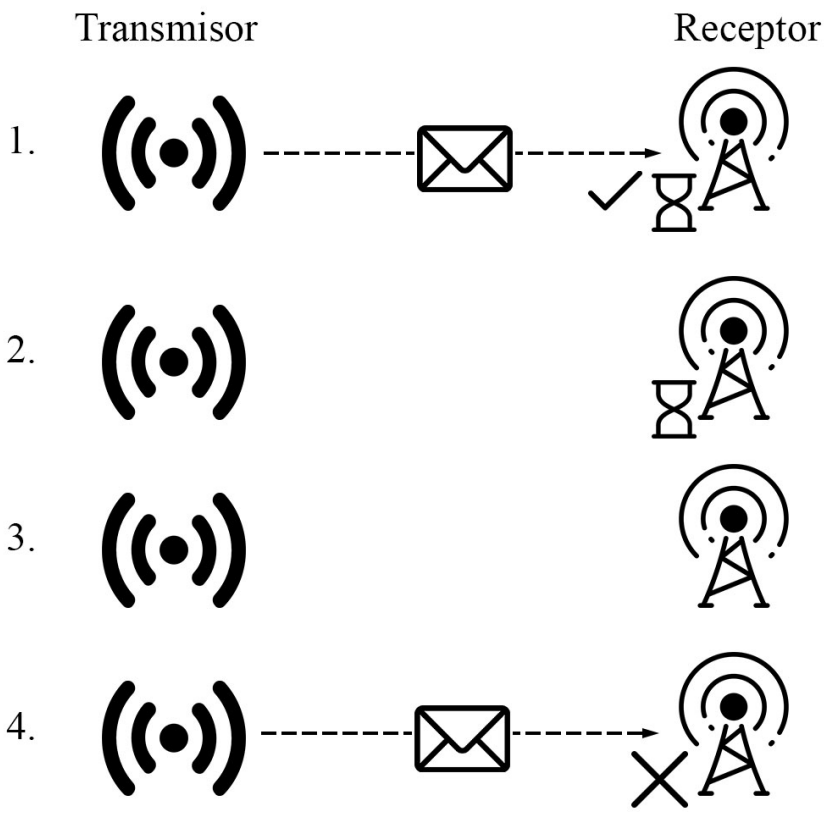


Figura 5.2: Representación del error encontrado en el modo No-ACK

Los errores encontrados en las figuras 5.3 y 5.4 para los modos de transmisión ACK-Always y ACK-On-Error respectivamente, tienen la misma naturaleza y son representados por la figura 5.5. Este error se puede describir como:

1. El transmisor envía el primer *tile*. El receptor inicia su proceso y se encuentra en ventana de recepción (indicado por el reloj de arena), por lo tanto, recibe correctamente el mensaje, lo ensambla e inicia el *timer* de inactividad.
2. El transmisor no envía el siguiente fragmento. El receptor se encuentra esperando el mensaje del transmisor en una ventana de recepción.
3. El transmisor sigue sin enviar el mensaje. En el lado del receptor expira el *timer* de inactividad y cierra la ventana de recepción.
4. El receptor entra a estado de error y para informar al transmisor que terminará la comunicación intenta enviar un fragmento *SCHC Receiver-Abort*, pero al mismo tiempo el transmisor intenta enviar el siguiente fragmento.

La figura 5.5 es una manera simple de ver lo que ocurre, ya que el perfil Sigfox indica que el receptor debe esperar un mensaje All-0 o All-1 para enviar el fragmento SCHC Receiver-Abort, sin embargo, la lógica es la misma.

```
(Spin Version 6.5.1 -- 31 July 2020)
Warning: Search not completed
                + Partial Order Reduction
Full statespace search for:
                never claim                 - (none specified)
                assertion violations          +
                cycle checks                - (disabled by -DSAFETY)
                invalid end states           +
State-vector 144 byte, depth reached 13, ••• errors: 1 •••
     11 states, stored
      0 states, matched
     11 transitions (= stored+matched)
      4 atomic steps
hash conflicts:       0 (resolved)
Stats on memory usage (in Megabytes):
   0.002                equivalent memory usage for states (stored*(State-vector +
overhead))
   0.246                actual memory usage for states
 128.000                  memory used for hash table (-w24)
   0.107                memory used for DFS stack (-m2000)
 128.302                  total actual memory usage
pan: elapsed time 0 seconds
```

Figura 5.3: Resultados al ejecutar los programas E y F para el modo ACK-Always.

5.2.1. Verificación de propiedad fairness

La verificación de una propiedad del tipo *fairness* permite realizar la verificación de una propiedad en un conjunto determinado de los casos totales. Debido a que los errores encontrados anteriormente corresponden a casos en donde se intentan enviar mensajes al mismo tiempo, se quiere verificar que en los casos en donde esto no ocurre no hay *deadlocks*. Estos casos corresponden a aquellos en donde la comunicación es *half duplex*, es decir, comunicaciones bidireccionales en donde no se envían mensajes al mismo tiempo.

Así, la propiedad puede ser definida en lógica temporal lineal como:

$$\begin{aligned}\Box((&listening\_ack == 1 \wedge sending\_ack == 1) \vee \\ &(listening\_ch == 1 \wedge sending\_ch == 1) \rightarrow \\ &\Diamond(Sender@end \wedge Receiver@end))\end{aligned} \tag{5.1}$$

La propiedad 5.1 busca verificar que los programas llegan a un estado final y utiliza variables enteras incluidas en la implementación del modelo en las secciones donde el transmisor y el receptor envían y reciben mensajes por los canales *ch* y *ack_ch*. La propiedad restringe los escenarios a solo los cuales en donde ocurra que si se está escuchando por el canal *ack_ch* (indicado por $listening\_ack == 1$), se está enviando un mensaje por el mismo canal (indicado por $sending\_ack == 1$) o si se está escuchando por el canal *ch* (indicado por $listening\_ch == 1$), se está enviando un mensaje por el mismo canal (indicado por $sending\_ch == 1$). Estas condiciones permiten que transmisor y receptor envíen mensajes pero impide que lo hagan al mismo tiempo. Además, la última parte de la propiedad ($\Diamond(Sender@end \wedge Receiver@end)$) indica que eventualmente el transmisor y el receptor llegarán a estados finales válidos sin *deadlocks*.

Para el modo ACK-Always, se vuelven a ejecutar los programas E y F con la propiedad 5.1 y se obtienen los resultados de la figura 5.6, en donde se indica que no se encontraron

```
(Spin Version 6.5.1 -- 31 July 2020)
Warning: Search not completed
                + Partial Order Reduction
Full statespace search for:
                never claim                 - (none specified)
                assertion violations          +
                cycle checks                - (disabled by -DSAFETY)
                invalid end states           +
State-vector 236 byte, depth reached 37, ••• errors: 1 •••
     29 states, stored
      0 states, matched
     29 transitions (= stored+matched)
     10 atomic steps
hash conflicts:       0 (resolved)
Stats on memory usage (in Megabytes):
   0.007                equivalent memory usage for states (stored*(State-vector +
overhead))
   0.259                actual memory usage for states
 128.000                  memory used for hash table (-w24)
   0.107                memory used for DFS stack (-m2000)
 128.302                  total actual memory usage
pan: elapsed time 0 seconds
```

Figura 5.4: Resultados al ejecutar los programas G y H para el modo ACK-On-Error.

errores, por lo tanto, la propiedad se cumple.

Para el modo ACK-On-Error en el perfil Sigfox, se vuelven a ejecutar los programas G y H con la propiedad 5.1 y se obtienen los resultados de la figura 5.7, donde se muestra que no se encontraron errores, por lo tanto, este modo de transmisión también cumple la propiedad.

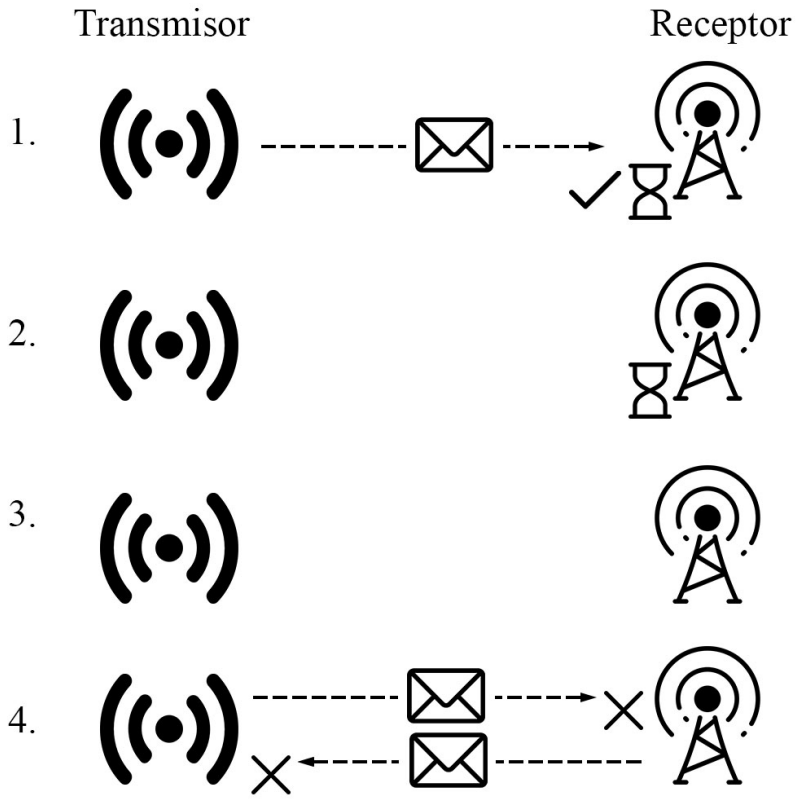


Figura 5.5: Representación del error encontrado en el modo ACK-Always y ACK-On-Error perfil Sigfox

```
1  (Spin Version 6.5.1 -- 31 July 2020)
2                  + Partial Order Reduction
3  Full statespace search for:
4                  never claim                   + (ltl_0)
5                  assertion violations             + (if within scope of claim)
6                  cycle checks                  - (disabled by -DSAFETY)
7                  invalid end states             - (disabled by never claim)
8  State-vector 168 byte, depth reached 96, ••• errors: 0 •••
9      307 states, stored
10     164 states, matched
11     471 transitions (= stored+matched)
12     178 atomic steps
13 hash conflicts:       0 (resolved)
14 Stats on memory usage (in Megabytes):
15    0.057                equivalent memory usage for states (stored*(State-vector
16 + overhead))
17    0.244                actual memory usage for states
18  128.000                  memory used for hash table (-w24)
19    0.107                memory used for DFS stack (-m2000)
20  128.302                  total actual memory usage
```

Figura 5.6: Resultados al ejecutar los programas E y F para propiedad 5.1 en el modo ACK-Always.

```
(Spin Version 6.5.1 -- 31 July 2020)
                + Partial Order Reduction
Full statespace search for:
                never claim                     + (ltl_0)
                assertion violations               + (if within scope of claim)
                cycle checks                    - (disabled by -DSAFETY)
                invalid end states               - (disabled by never claim)
State-vector 260 byte, depth reached 202, ••• errors: 0 •••
    1121 states, stored
    1141 states, matched
    2262 transitions (= stored+matched)
    1059 atomic steps
hash conflicts:       0 (resolved)
Stats on memory usage (in Megabytes):
   0.308                equivalent memory usage for states (stored*(State-vector
+ overhead))
   0.453                actual memory usage for states
 128.000                  memory used for hash table (-w24)
   0.107                memory used for DFS stack (-m2000)
 128.498                  total actual memory usage
```

Figura 5.7: Resultados al ejecutar los programas G y H con la propiedad 5.1 en el modo ACK-On-Error para la tecnología Sigfox.

Capítulo 6

Análisis

Este capítulo muestra un análisis a los resultados encontrados en el capítulo anterior, se describen las consecuencias de estos y se explican los riesgos que corre el protocolo ante posibles ataques en donde se explotan las vulnerabilidades encontradas.

6.1. No-ACK

El error representado en la figura 5.2 muestra el escenario en donde el transmisor demora en enviar un fragmento, generando que en el lado del receptor expire el *timer* de inactividad y entre a un estado de error. Cuando el transmisor intenta enviar el fragmento, no puede ya que el receptor ha terminado la comunicación y no se encuentra escuchando por el canal de transmisión.

Este tipo de error corresponde a un *deadlock*, ya que el receptor termina en un estado final, pero el transmisor se encuentra estancado intentando enviar un mensaje por el canal de transmisión.

En una transmisión real, esta situación no corresponde a un error, sino que a un caso en donde el mensaje enviado por el transmisor no es recibido por el receptor debido a que este ha terminado la comunicación. Si bien el resultado mostrado por el *model checker* muestra que el transmisor logra enviar un fragmento, en una transmisión esta situación podría ocurrir luego de que el transmisor haya enviado varios fragmentos.

La descripción del protocolo indica que cuando el receptor entra al estado de error, debe terminar el reensamblaje de este paquete, descartando los fragmentos ya recibidos, por lo que no recibir un fragmento y terminar la comunicación no genera un problema. La descripción no indica que debe hacer el receptor si está en estado de error y sigue recibiendo fragmentos correspondientes al paquete que se ha descartado.

La descripción del protocolo podría indicar que cuando el receptor está en estado de error, debe seguir recibiendo los mensajes correspondientes al paquete descartado y de la misma forma descartar estos nuevos fragmentos recibidos pero un atacante podría aprovechar esta

situación para mantener al receptor ocupado recibiendo estos fragmentos y realizar un ataque de denegación de servicio.

Para evitar este escenario y un posible ataque, la descripción debe indicar en el documento [16], en la sección 8.4.1.2, que corresponde a la sección que indica el comportamiento del receptor en el modo No-ACK, cual es el comportamiento del receptor al estar en estado de error, el que podría corresponder a terminar la comunicación con el transmisor y en caso de que el dispositivo siga activo, ignorar los mensajes que correspondan a transmisiones terminadas.

6.2. ACK-Always

Antes de analizar los resultados sobre el modo ACK-Always se mostrarán algunas observaciones de este modo de transmisión que se pueden hacer a partir de la máquina de estados de la figura 4.5 que representa al receptor.

Al realizar una inspección al modelo del receptor que se muestra en la figura 4.5, el estado "*recv_w*" que corresponde al estado en donde el receptor está en una ventana de recepción para que el transmisor le envíe una ventana de *tiles*, ocurre que al recibir un fragmento con el campo W que indica el número de ventana, distinto al número de ventana que actualmente se está recibiendo, el receptor descarta el fragmento y vuelve a iniciar el *timer* de inactividad.

Aquí se puede notar el riesgo de volver a iniciar el *timer* de inactividad, si bien, es una medida para impedir que el receptor se quede esperando un mensaje por mucho tiempo, un atacante puede aprovecharse de esto para enviar de forma consecutiva fragmentos con distinto valor de W y mantener al receptor ocupado recibiendo estos mensajes que no serán considerados en el reensamblaje, corriendo el riesgo de un ataque de denegación de servicio.

Otra observación que se puede realizar al modelo del receptor de la figura 4.5 corresponde al estado "*Clean up*". En este estado de "limpieza" el receptor espera a que expire el *timer* de inactividad, que le envíen un fragmento Sender-Abort o que se hayan realizado muchos intentos de re conexión para terminar la recepción de mensajes. Al igual que en la observación anterior, al volver a iniciar el *timer* de inactividad un atacante podría enviar varios fragmentos con un valor W distinto al que se está procesando y mantener al receptor ocupado pudiendo realizar un ataque de denegación de servicio.

Estas observaciones traen a discusión el número de veces seguidas que el receptor puede recibir un fragmento que no corresponde a la ventana que actualmente está procesando y que descartará. Así como se restringe el número de veces seguidas que el receptor puede recibir mensajes ACK Request, estas observaciones muestran que también se debería restringir el número de veces seguidas que el receptor descarta un fragmento para evitar posibles ataques.

A pesar de estas observaciones el error encontrado por el *model checker* tiene una naturaleza distinta. El error encontrado en la figura 5.5 es compartido por los modos ACK-Always y ACK-On-Error para el perfil Sigfox, en donde ocurre que al mismo tiempo el transmisor intenta enviar un fragmento del paquete y el receptor intenta enviar el fragmento SCHC

Receiver-Abort que indica el fin de la comunicación, pero como ninguno de los programas está escuchando por el canal de transmisión, entonces ninguno puede terminar sus ejecución. Este error corresponde a un *deadlock* en donde ni el transmisor ni el receptor pueden terminar en estados finales, en particular, ambos quedan estancados intentando enviar mensajes por canales de tamaño 0.

Si bien en la realidad ambos programas podrían enviar los mensajes, este error se puede interpretar como el caso en donde ni el transmisor ni el receptor reciben los mensajes que se intentan enviar.

En el caso del modo ACK-Always, lo que ocurriría en una transmisión como la mostrada en la figura 5.5 se muestra en la figura 6.1, en donde sucedería lo siguiente:

1. El transmisor envía el primer fragmento, este es recibido por el receptor.
2. El receptor está en ventana de recepción, pero el transmisor no envía el fragmento, esto puede ocurrir con cualquier fragmento, no necesariamente el segundo.
3. En el lado del receptor expira el *timer* de inactividad.
4. El transmisor envía un fragmento del paquete, al mismo tiempo el receptor envía un mensaje SCHC Receiver-Abort y termina la comunicación con el transmisor. Ninguno de estos mensajes son recibidos.
5. El transmisor envía fragmentos del paquete hasta que termina de enviar una ventana de *tiles*. Si el receptor no se encuentra en una ventana de recepción, entonces, no recibe el mensaje. Si el receptor se encuentra en una ventana de recepción, entonces, lo descarta.
6. El transmisor espera el mensaje ACK por parte del receptor.
7. Como el receptor no envía el mensaje ACK, ya que ha terminado la comunicación, en el lado del transmisor expira el *timer* de retransmisión y envía un mensaje ACK Request al receptor. Este mensaje puede ser recibido por el receptor, pero como ha terminado la transmisión, lo descarta y no responde.

   El transmisor repite los pasos 6 y 7 hasta enviar *max_ack_requests* mensajes ACK Request.
8. Al no recibir respuesta, el transmisor termina la comunicación enviando un mensaje SCHC Sender-Abort que también es descartado por el receptor.

En la sección 8.4.2.2 el documento [16] indica que cuando el receptor entra en estado de error debe terminar la transmisión, también indica que antes de iniciar una transmisión debe revisar que el fragmento recibido no sea resto de un paquete anterior, por lo tanto, al volver a recibir un fragmento de una transmisión terminada, lo ignorará. Si bien en este caso ambos programas pueden terminar, este escenario muestra la importancia que tiene la variable *max_ack_requests* y el tiempo del *timer* de inactividad en el lado del transmisor, ya que un atacante podría aprovechar estas variables para mantener al transmisor ocupado el mayor tiempo posible, impidiendo que este envíe mensajes a otra aplicación.

6.3. ACK-On-Error Perfil Sigfox

En el caso del modo ACK-On-Error en el perfil Sigfox se obtiene el mismo error que para el modo ACK-Always mostrado en la figura 5.5. En el caso del modo ACK-On-Error para el perfil Sigfox, lo que ocurriría en una transmisión real es similar a lo mostrado en la figura 6.1, pero se diferencian en que en el paso 6, el transmisor al no recibir un mensaje ACK, sigue con el envío de las ventanas siguientes. Luego de enviadas todas las ventanas ocurren los pasos 7 y 8, en donde el transmisor envía mensajes ACK REQ y al no recibir respuesta, pasa a estado de error y termina la comunicación enviando un mensaje SCHC Sender-Abort.

La descripción general del protocolo [16] indica en la sección 8.4.3.2 que el receptor debe ignorar mensajes de transmisiones que ya han terminado, por lo tanto, al entrar a estado de error y terminar la comunicación con el receptor, ignorará todos los mensajes siguientes, por lo tanto, esta situación no genera un error en el protocolo, pero muestra la importancia que tiene la variable *max_ack_request* y el valor del *timer* de inactividad.

Si bien la descripción general del protocolo indica que el receptor debe ignorar mensajes de transmisiones que ya han terminado, un perfil podría indicar lo contrario, es decir, que al volver a recibir un fragmento de una sesión que ya ha terminado. Este comportamiento no es recomendado ya que se debe definir si el receptor descarta los fragmentos recibidos al terminar la comunicación con el transmisor por pasar a un estado de error. Si el receptor descarta los fragmentos ya recibidos como se hace en el modo No-ACK, entonces, cuando vuelva a iniciar una sesión SCHC porque recibió un fragmento de una transmisión terminada, enviará un paquete ACK indicando que faltan las ventanas anteriores (las descartadas al terminar la sesión anterior), lo que le da más tiempo a un atacante para mantener al receptor iniciando sesiones de manera consecutiva. En cambio, si no se descartan los fragmentos ya recibidos, se tiene la discusión de por cuánto tiempo debe el receptor almacenar estos fragmentos, teniendo en cuenta que existe la posibilidad de que el transmisor nunca vuelva a enviar un fragmento de ese paquete.

6.4. Propiedad verificada

Para los modos ACK-Always y ACK-On-Error en el perfil Sigfox se verifica la propiedad 5.1 y las figuras 5.6 y 5.7 indican que ambos modos satisfacen esta propiedad. Esto significa que en transmisiones en donde la comunicación es *half-duplex* y en donde todos los mensajes enviados son recibidos, tanto el transmisor como el receptor terminan sus procesos en estados finales válidos sin *deadlocks*. Es importante notar que al verificar esta propiedad el transmisor y el receptor se comportan como indica la descripción y que ninguno tiene el comportamiento de un atacante.

Que esta propiedad se cumpla y que el protocolo presente errores cuando no se aplica le restricción indica que el protocolo está hecho para entornos *half-duplex* y que este podría presentar errores si alguna capa superior se comportara de manera *full-duplex*. El tipo de comunicación no se indica ni en la descripción general ni en el perfil Sigfox, aunque las tecnologías que han implementado este protocolo no soportan comunicaciones *full-duplex*

[15], esto no quiere decir que en el futuro no lo harán y el protocolo podría presentar errores debido a esto.

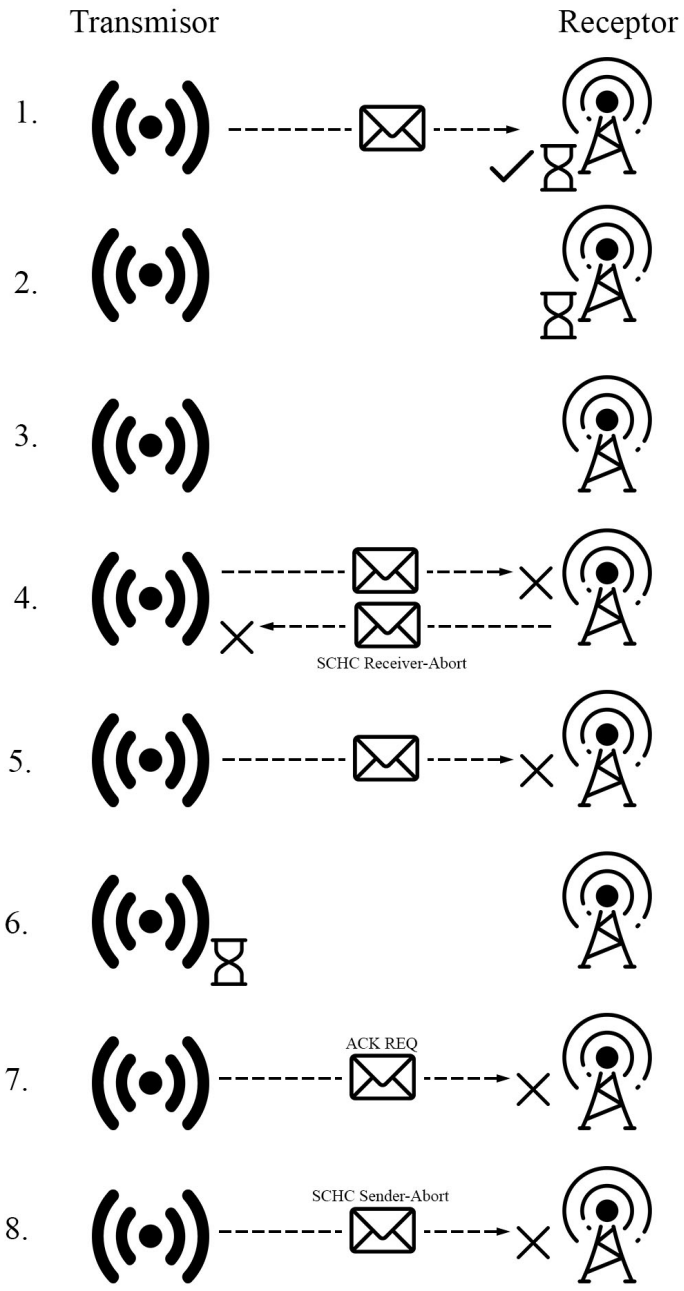


Figura 6.1: Comportamiento del modo ACK-Always cuando el transmisor no recibe mensaje SCHC Receiver-Abort

Capítulo 7

Conclusiones y trabajo a futuro

7.1. Conclusiones

En esta tesis se revisó el protocolo SCHC mediante la técnica de verificación *model checking*. Se modelaron los tres modos de transmisión del protocolo, No-ACK, ACK-Always para la descripción general del protocolo y ACK-On-Error para el Perfil Sigfox. Se revisaron los modos de transmisión por separado, buscando comprobar si el protocolo de comunicación SCHC presenta errores o *bugs* en su definición.

Además, para los modos ACK-Always y ACK-On-Error se añade una restricción *fairness* para disminuir los escenarios que verifica el *model checker* a solo los que consideran comunicación de tipo *half duplex*.

El protocolo SCHC fue publicado como RFC en el año 2020 y las implementaciones de este son pocas. Es por esta razón que esta investigación busca ser una revisión preventiva para evitar que las implementaciones recientes y las próximas presenten problemas o *bugs*.

Los resultados muestran que los tres modos de transmisión presentan comportamientos que podrían derivar en vulnerabilidades en el protocolo. Según el *model checking*, los resultados corresponden a *deadlocks* que se presentan cuando el receptor entra a estado de error y termina la comunicación con el transmisor. Sin embargo, se pudo ver que en transmisiones reales no corresponden a *deadlocks* pero que si dejan ver vulnerabilidades en el protocolo.

Los resultados de esta tesis muestran que la descripción del protocolo SCHC necesita ser complementada, en particular, se necesita más detalle sobre el comportamiento del receptor cuando entra a un estado de error. Esta tesis propone que al entrar en estado de error, el receptor termine todo tipo de comunicación con el transmisor para evitar cualquier tipo de ataque.

Sin la descripción explícita del estado de error, la solución a los problemas encontrados queda en manos de quien implementa el protocolo y por lo tanto, cada implementación podría solucionarlos de manera distinta, generando que cada implementación se comporte de manera distinta y se podrían generar problemas de compatibilidad dentro del mismo

protocolo. Además, en el caso de que las vulnerabilidades no sean solucionadas se deja abierta una puerta a que se realicen ataques a los dispositivos que utilizan el protocolo.

La descripción del protocolo indica por separado el comportamiento del transmisor y del receptor, donde no indica si el tipo de comunicación es *half duplex* o *full duplex*, pero se puede concluir que corresponde a *half duplex* ya que mientras el transmisor está en ventana de envío, el receptor se encuentra esperando mensajes y vice versa. Los resultados muestran que los errores encontrados se originan cuando transmisor y receptor se encuentran en ventanas de envío, rompiendo el principio de la comunicación *half duplex*, por lo tanto, también es necesario que la descripción indique explícitamente el tipo de comunicación que soporta el protocolo, ya que al pasar de una a otra se podrían generar errores.

Tener un protocolo totalmente seguro, que no tenga fallas y no sea vulnerable a ataques sería un escenario ideal, pero es difícil cumplir con todos estos requisitos, más aún en redes LPWAN en donde se tienen restricciones en el tamaño de los mensajes y en la cantidad de mensajes que se pueden enviar. Es por esto que las verificaciones de los protocolos tienen gran importancia, ya que previenen fallas y ataques. En el contexto de redes LPWAN donde se busca que los dispositivos tengan un bajo costo monetario, armar una gran red de *bots* cada vez se vuelve más fácil si se tienen los recursos, por lo tanto, es necesario que los protocolos de comunicación utilizados en estas redes no tengan fallas ni vulnerabilidades que puedan ser utilizadas por atacantes.

7.2. Trabajo a futuro

Esta tesis cumple con sus objetivos, sin embargo, puede ser extendida para revisar y verificar otros aspectos del protocolo.

Se podrían modelar al transmisor o al receptor como atacantes, en donde se pueda ejecutar el *model checker* con el transmisor o el receptor como atacantes frente a un receptor o transmisor respectivamente con el comportamiento según indica la descripción del protocolo.

Se podrían modelar los paquetes perdidos, ya que esta tesis considera que todos los paquetes son enviados a la red y que estos no se pierden, se propone agregar un factor probabilístico en el envío de los paquetes.

Finalmente, se propone refinar el modelado de los fragmentos, modificando el tipo de los atributos en los fragmentos a bits, que corresponden al tipo original utilizado por el protocolo y así utilizar operaciones sobre los bits al momento de modificar atributos en los fragmentos, que es como indica la descripción que se debe hacer.

Bibliografía

[1] Sergio Aguilar, Diego S. Wistuba La-Torre, Antonis Platis, Rafael Vidal, Carles Gomez, Sandra Céspedes, and Juan Carlos Zúñiga. Packet fragmentation over sigfox: Implementation and performance evaluation of schc ack-on-error. *IEEE Internet of Things Journal*, pages 1–1, 2021.

[2] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[3] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. 1 edition, 2008.

[4] M. Benerecetti, R. De Guglielmo, U. Gentile, S. Marrone, N. Mazzocca, R. Nardone, A. Peron, L. Velardi, and V. Vittorini. Dynamic state machines for modelling railway control systems. *Science of Computer Programming*, 133:116–153, 2017. Formal Techniques for Safety-Critical Systems (FTSCS 2014).

[5] Massimo Benerecetti, Ugo Gentile, Stefano Marrone, Roberto Nardone, Adriano Peron, Luigi L. L. Starace, and Valeria Vittorini. From dynamic state machines to promela. In Fabrizio Biondi, Thomas Given-Wilson, and Axel Legay, editors, *Model Checking Software*, pages 56–73, Cham, 2019. Springer International Publishing.

[6] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer Publishing Company, Incorporated, 1st edition, 2018.

[7] Zheng Fang, Hao Fu, Tianbo Gu, Zhiyun Qian, Trent Jaeger, Pengfei Hu, and Prasant Mohapatra. A model checking-based security analysis framework for iot systems. *High-Confidence Computing*, 1(1):100004, 2021.

[8] Stephen Farrell. Low-Power Wide Area Network (LPWAN) Overview. RFC 8376, May 2018.

[9] Olivier Gimenez and Ivaylo Petrov. Static Context Header Compression and Fragmentation (SCHC) over LoRaWAN. RFC 9011, April 2021.

[10] Jiaxing Guo, Chunxiang Gu, Xi Chen, and Fushan Wei. Model learning and model checking of ipsec implementations for internet of things. *IEEE Access*, 7:171322–171332, 2019.

[11] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Inc., USA, 1990.

[12] G.J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[13] Lukas Humbel, Daniel Schwyn, Nora Hossle, Roni Haecki, Melissa Licciardello, Jan Schaer, David Cock, Michael Giardino, and Timothy Roscoe. A model-checked i2c specification. In Alfons Laarman and Ana Sokolova, editors, *Model Checking Software*, pages 177–193, Cham, 2021. Springer International Publishing.

[14] K. Keerthi, Indrani Roy, Aritra Hazra, and Chester Rebeiro. *Formal Verification for Security in IoT Devices*, pages 179–200. Springer International Publishing, Cham, 2019.

[15] Kais Mekki, Eddy Bajic, Frederic Chaxel, and Fernand Meyer. A comparative study of lpwan technologies for large-scale iot deployment. *ICT Express*, 5(1):1–7, 2019.

[16] Ana Minaburo, Laurent Toutain, Carles Gomez, Dominique Barthel, and Juan-Carlos Zúñiga. SCHC: Generic Framework for Static Context Header Compression and Fragmentation. RFC 8724, April 2020.

[17] Mujahid Mohsin, Muhammad Usama Sardar, Osman Hasan, and Zahid Anwar. Iotriskanalyzer: A probabilistic model checking based framework for formal risk analytics of the internet of things. *IEEE Access*, 5:5494–5505, 2017.

[18] Rodrigo Muñoz, Juan Saez Hidalgo, Felipe Canales, Diego Dujovne, and Sandra Céspedes. Schc over lorawan efficiency: Evaluation and experimental performance of packet fragmentation. *Sensors*, 22(4), 2022.

[19] Edgar Ramos and Ana Minaburo. SCHC over NB-IoT. Internet-Draft draft-ietf-lpwan-schc-over-nbiot-07, Internet Engineering Task Force, February 2022. Work in Progress.

[20] Usman Raza, Parag Kulkarni, and Mahesh Sooriyabandara. Low power wide area networks: An overview. *IEEE Communications Surveys Tutorials*, 19(2):855–873, 2017.

[21] Jesus Sanchez-Gomez, Jorge Gallego-Madrid, Ramon Sanchez-Iborra, Jose Santa, and Antonio F. Skarmeta. Impact of schc compression and fragmentation in lpwan: A case study with lorawan. *Sensors*, 20(1), 2020.

[22] Alireza Souri and Monire Norouzi. A state-of-the-art survey on formal verification of the internet of things applications. *Journal of Service Science Research*, 11(1):47–67, Jun 2019.

[23] Wei Zhang, Meihong Yang, Xinchang Zhang, and Huiling Shi. Model checking the dns under dns cache-poisoning attacks using spin. *ScienceAsia*, 42S:49, 01 2016.

[24] Juan-Carlos Zúñiga, Carles Gomez, Sergio Aguilar, Laurent Toutain, Sandra L. Cespedes, Diego S. Wistuba La Torre, and Julien Boite. SCHC over Sigfox LPWAN. Internet-Draft draft-ietf-lpwan-schc-over-sigfox-09, Internet Engineering Task Force, February 2022. Work in Progress.

Anexos

Anexo A

Definición de máquinas de estado dinámicas

Una máquina de estado dinámica o DSTM D es definida como una tupla $\langle M_1, ..., M_n, X, C, P\rangle$ donde:

- X, C, P son conjuntos finitos de variables, canales y parámetros respectivamente.
- M_1 es la máquina inicial definida sobre X, C. En esta máquina no se permiten parámetros.
- M_i con $i \in 1, ..., n$ es una máquina definida sobre X, C y P de la forma: $\langle P_i, N_i, En_i, df_i, Ex_i, Bx_i, Y_i, Fk_i, Jn_i, \Lambda_i\rangle$, donde:
  - $P_i \subseteq P$ es el conjunto local de parámetros de la máquina M_i (P_1 debe ser el conjunto vacío).
  - N_i es un conjunto finito de nodos.
  - En_i es el conjunto finito de pseudo nodos entrantes.
  - $df_i \in En_i$ es el pseudo nodo inicial por defecto.
  - $Ex_i \subseteq N_i$ es el conjunto de nodos de salida
  - Bx_i es el conjunto finito de nodos Box.
  - $Y_i : Bx_i \to \{1, ..., n\}^\star$ asigna a cada nodo box una lista de índices de máquinas.
  - Fk_i es el conjunto finito de pseudo nodos fork.
  - Jn_i es el conjunto finito de pseudo nodos join.
  - $\Lambda = \langle T_i, Src_i, Dec_i, Trg_i, Inst_i\rangle$ es la estructura que define el conjunto de transiciones de M_i, donde:
    - T_i es el conjunto finito de etiquetas de las trancisiones.
    - $Scr_i : T_i \to Source_i$ asigna un origen a cada etiqueta de transición. Donde $Source_i = (N_i \backslash Ex_i) \cup En_i \cup Bx_i \cup (Bx_i \times Ex(D)) \cup Fk_i \cup (Fk_i \times \{\downarrow\}) \cup Jn_i$ y $Ex(D) = \bigcup_{1 \leqslant j \leqslant n} Ex_j$

* $Dec_i : T_i \to \Xi_{P_i} \times \Phi_{P_i} \times A_{P_i}$ asocia cada transición con su decorador, denominados trigger, guarda y acción de Λ_i
* $Trg_i : T_i \to Target_i$ asigna un destino a cada transición. Donde $Target_i = N_i \cup Bx_i \cup (Bx_i \times En(D)) \cup Fk_i \cup Jn_i \cup (Jn_i \times \{\otimes\})$ y $En(D) = \bigcup_{1 \leqslant j \leqslant n} En_j$
* $Inst_i : T_i \rightharpoonup (\Upsilon_P)^*$ es la función parcial que asigna una secuencia de sustitución de parámetros sobre P a una transición.

La relación de Fk_i con el símbolo $\downarrow$ y la relación de Jn_i con el símbolo $\otimes$ son utilizadas para determinar un origen fork como asíncrono y para determinar un destino fork como preemptivo respectivamente.

Anexo B

Esquema para implementar máquinas de estado dinámicas en Promela

Listing B.1: Formato programa principal [5]

```
active proctype Engine(){
    pid PidMain; byte i;
    chan chT_Main = [1] of {bit};
    chan chT_Main_ex = [1] of {bit};
    PidMain = run Main(_pid, initial, chT_Main, chT_Main_ex);
    ChildrenMatrix[_pid].children[PidMain]=1;

    nextStep: // starts a new step
        atomic {
            // handle external channels management
            HasFired=0;
            for (i: 0 .. MAX_PROC-1){
                HasExecuted[i]=0; descendantExecuted[i]=0;
                HasToken[i] = ChildrenMatrix[_pid].children[i];
            }
        }
        goto waitTimeout;

    nextPhase: // starts a new phase in the current step
        atomic{
            HasFired=0;
            for ( i : 0 .. MAX_PROC - 1){
            // given token to children
                HasToken[i] = ChildrenMatrix[_pid].children[i];
            }
        }
        goto waitTimeout;

    waitTimeout:
```

```
        do
        :: timeout -> //deadlock
            if
            :: (!HasFired) -> goto nextStep;
            :: (HasFired) -> goto nextPhase;
            fi;
        od unless (chT_Main_ex?[_]) -> {chT_Main!<term>}
}
```

Listing B.2: Formato para cada máquina de estados [5]

```
proctype M(pid parent; mtype initial;chan chT;chan chT_ex){
    // declare channels for termination synch. with children here
    byte i; mtype state=ready, DSTMstate=initial;
    do
    // for each state S ∈ N_i ∪ En_i
    :: (DSTMstate==S && HasToken[_pid] && state==ready) ->
    atomic {
        HasToken[_pid]=0;
        if
        // for each transition t with
        // Src(t)=S, Trg(t)=T, Dec(t)=⟨ξ,φ,α⟩
        :: (ξ && φ && !descendantExecuted[_pid]) ->
            α; DSTMstate = T; HasFired=1;
            HasExecuted[_pid]=1; descendantExecuted[_pid]=1;
        :: else -> // no transition is executable
            if
            :: (!HasExecuted[_pid]) ->
            // did not exec, in this step
                for (i:0..MAX_PROC-1) { // pass token to children
                    if
                    ::(ChildrenMatrix[_pid].children[i]) ->
                        HasToken[i]=1;
                    ::else->skip;
                    fi;
                }
            ::else->skip;
            fi;
        fi;
        state = backProp;
    }
    // for each existing state ex ∈ Ex_i
    ::(DSTMstate==ex && chT?[term])->{chT?term; goto die}
    // handle upwards propagation of descendantExecuted
    ::(state==backProp && descendantExecuted[_pid] &&
        !descendantExecuted[parent]) ->
        { descendantExecuted[parent] = 1 }
    // handle original state restoring after backProp
```

```
    ::(state==backProp && HasToken[_pid])->{state=ready}
    od unless (dyingPid[parent] || chT?[interrupt]) -> {
        if
        :: (chT?[_]) -> chT?<_>
        :: else->skip
        fi;
        goto die
    }
    die: dyingPid[_pid]=1

}
```

Anexo C

Código para transmisor en modo No-ACK

```
active proctype Sender(){
    init_variables:
    fragment.abort = 0;
    int tiles_to_send = 10; /* Number of tiles to send */
    send:
    if
    :: tiles_to_send > 0 -> atomic{ /* Regular SCHC fragment */
        fragment.FCN = 0;
        ch ! fragment;
        tiles_to_send--;
        goto send
    }
    :: tiles_to_send == 0 -> atomic{ /* Last fragment of packet */
        fragment.FCN = 1;
        ch ! fragment;
        goto end
    }
    :: true -> atomic{ /* Send Sender-Abort */
        fragment.abort = 1;
        ch ! fragment;
        goto error
    }
    fi;

    error:
    printf("Error state\n");

    end:
    printf("End state\n");
}
```

Anexo D

Código para receptor en modo No-ACK

```
active proctype Receiver(){
    int n_fragments = 0;
    Fragment rcv_fragment;
    bit tile_received = 0;
    rcv_frag:
    if
    :: tile_received -> goto error; /* Inactivity timer */
    :: ch ? rcv_fragment ->
        tile_received = 1;
        if
        :: rcv_fragment.FCN == 0 && rcv_fragment.abort == 0 ->
        atomic{
            n_fragments++;
            goto rcv_frag
        }
        :: rcv_fragment.FCN == 1 && rcv_fragment.abort == 0 ->
            if
            :: goto error /* RCS wrong */
            :: n_fragments++; goto end /* RCS right */
            fi;
        :: rcv_fragment.abort == 1 -> goto error /* Sender-Abort
        received */
        fi;
    fi;

    error:
    printf("Error state\n");

    end:
    printf("End state\n");
```

```
}
```

Anexo E

Código para transmisor en modo ACK-Always

```
active proctype Sender(){
    Fragment fragment;
    Fragment recv_ack;

    bit W = 0;
    int c_bm = 0;
    int FCN = WINDOWS_SIZE;
    int attempt = 0;
    int N_WINDOWS = WINDOWS; /* Numbered from 0 to N_WINDOWS */
    int p_FCN;

    send:
    if
    :: FCN != 0 -> atomic{
        /* Send regular fragment */
        fragment.FCN = FCN;
        fragment.W = W;
        fragment.ackRequest = 0;
        fragment.abort = 0;
        fragment.all1 = 0;
        sending_ch++;
        ch ! fragment;
        sending_ch--;
        c_bm++;
        FCN--;
        goto send
    }
    :: FCN == 0 && N_WINDOWS > 0 -> atomic{
        /* Send All-0 */
        fragment.FCN = 0;
```

```
        fragment.W = W;
        fragment.ackRequest = 0;
        fragment.abort = 0;
        fragment.all1 = 0;
        sending_ch++;
        ch ! fragment;
        sending_ch--;
        c_bm++;
        goto wait_bitmap
    }
    :: FCN == 0 && N_WINDOWS == 0 -> atomic{
        /* Send All-1 */
        fragment.FCN = 0;
        fragment.W = W;
        fragment.ackRequest = 0;
        fragment.abort = 0;
        fragment.all1 = 1;
        sending_ch++;
        ch ! fragment;
        sending_ch--;
        c_bm++;
        goto wait_bitmap
    }
    fi;

    wait_bitmap:
    if
    :: empty(ack_ch) -> /* Retransmission timer expiration */
        if
        :: attempt < MAX_ACK_REQUESTS -> atomic{
            /* Send ACK REQUEST */
            fragment.FCN = 0;
            fragment.W = W;
            fragment.ackRequest = 1;
            fragment.abort = 0;
            fragment.all1 = 0;
            sending_ch++;
            ch ! fragment;
            sending_ch--;
            attempt++;
            goto wait_bitmap
        }
        :: else -> atomic{
            /* Send Sender-Abort */
            fragment.FCN = 0;
            fragment.W = W;
            fragment.ackRequest = 0;
```

```
            fragment.abort = 1;
            fragment.all1 = 0;
            sending_ch++;
            ch ! fragment;
            sending_ch--;
            goto error
        }
        fi;

    :: atomic{
        listening_ack++;
        ack_ch ? recv_ack
        listening_ack--;} -> /* Receiving a SCHC ACK */
        if
        :: recv_ack.abort == 1 ->
            goto error

        :: recv_ack.W == W && recv_ack.bm < c_bm &&
            recv_ack.abort == 0 -> atomic{
            /* Current window, some tiles are missing */
            attempt++;
            goto resend_missing_frag
        }
        :: recv_ack.W == W &&  c_bm == recv_ack.bm &&
            N_WINDOWS > 0  && recv_ack.abort == 0 -> atomic{
            /* Change to next window */
            W = !W;
            N_WINDOWS--;
            c_bm = 0;
            FCN = WINDOWS_SIZE;
            attempt = 0;
            goto send;
        }
        :: recv_ack.W == W && N_WINDOWS == 0 && recv_ack.bm > c_bm
            && recv_ack.abort == 0 -> atomic{
            /* Last window, received more tiles than sent */
            /* Send abort */
            fragment.FCN = 0;
            fragment.W = W;
            fragment.ackRequest = 0;
            fragment.abort = 1;
            fragment.all1 = 0;
            sending_ch++;
            ch ! fragment;
            sending_ch--;
            goto error;
        }
```

```
    :: recv_ack.W == W && N_WINDOWS == 0 &&
        c_bm == recv_ack.bm && recv_ack.c == 0
        && recv_ack.abort == 0 -> atomic{
        /* Last window, all tiles were received,
        wrong integrity check */
        /* Send abort */
        fragment.FCN = 0;
        fragment.W = W;
        fragment.ackRequest = 0;
        fragment.abort = 1;
        fragment.all1 = 0;
        sending_ch++;
        ch ! fragment;
        sending_ch--;
        goto error;
    }
    :: recv_ack.W == W && N_WINDOWS == 0 && recv_ack.c == 1
        && recv_ack.bm == c_bm && recv_ack.abort == 0 ->
        /* Last window, all tiles were received,
        right integrity check */
        goto end;

    :: recv_ack.W != W && recv_ack.abort == 0 ->
        /* Not expected window */
        goto wait_bitmap; /* discard frag */
    fi;
fi;

resend_missing_frag:
p_FCN = WINDOWS_SIZE; //Partial FCN
do
:: p_FCN >= recv_ack.bm -> atomic{
    fragment.FCN = p_FCN;
    fragment.W = W;
    fragment.ackRequest = 0;
    fragment.abort = 0;
    if
    :: N_WINDOWS == 0 && p_FCN == 0 -> fragment.all1 = 1;
    :: else -> fragment.all1 = 0;
    fi;
    sending_ch++;
    ch ! fragment;
    sending_ch--;
    p_FCN--;
}
:: else -> break
od;
```

```
    goto wait_bitmap;

    error:
    printf("Error state\n");
    end:
    printf("Error state\n");

}
```

Anexo F

Código para receptor en modo ACK-Always

```
active proctype Receiver(){
    bit W = 0;
    int local_W = 0;
    int c_bm = 0;
    int attempt = 0;
    bit tile_received = 0;
    Fragment fragment;
    Fragment ack;

    rcv_w:
    if
    :: empty(ch) && tile_received -> atomic{ /* Inactivity timer */
        ack.abort = 1;
        sending_ack++;
        ack_ch ! ack;
        sending_ack--;
        goto end
    }
    :: atomic{
        listening_ch++;
        ch ? fragment;
        tile_received = 1;
        listening_ch--;} ->
        if
        :: fragment.W != W && fragment.abort == 0 ->
            goto rcv_w /* Discard fragment */

        :: fragment.W == W && fragment.ackRequest == 1 -> atomic{
            /* Receiving ACK REQUEST for current window */
            ack.bm = c_bm;
```

```
        ack.W = W;
        ack.abort = 0;
        sending_ack++;
        ack_ch ! ack;
        sending_ack--;
        attempt++;
        goto rcv_w
    }
    :: fragment.W == W && fragment.all1 == 0 &&
        fragment.FCN != 0 && fragment.ackRequest == 0
        && fragment.abort == 0 -> atomic{
        /* Current window, regular fragment */
        c_bm++;
        goto rcv_w
    }
    :: fragment.W == W && fragment.FCN == 0 &&
        fragment.all1 == 0 && fragment.ackRequest == 0 &&
        fragment.abort == 0 -> atomic{/* Current window,
        last fragment of window (All-0) */
        c_bm++;
        ack.bm = c_bm;
        ack.W = W;
        ack.abort = 0;
        sending_ack++;
        ack_ch ! ack;
        sending_ack--;
        attempt++;
        goto retransmission
    }
    :: fragment.W == W && fragment.all1 == 1 &&
        fragment.ackRequest == 0 && fragment.abort == 0 ->
        /* Current window All-1 */
        if
        :: true -> atomic{ /* RCS Wrong */
            c_bm++;
            ack.bm = c_bm;
            ack.W = W;
            ack.c = 0;
            ack.abort = 0;
            sending_ack++;
            ack_ch ! ack;
            sending_ack--;
            attempt++;
            goto retransmission
        }
        ::  true -> atomic{/* RCS Right*/
            c_bm++;
```

```
                ack.bm = c_bm;
                ack.c = 1;
                ack.W = W;
                ack.abort = 0;
                sending_ack++;
                ack_ch ! ack;
                sending_ack--;
                attempt++;
                goto clean_up
            }
            fi;
        :: fragment.abort == 1 -> atomic{
            /* On receiving sender-abort */
            ack.abort = 1;
            sending_ack++;
            ack_ch ! ack;
            sending_ack--;
            goto end
        }
        fi;
    :: attempt >= MAX_ACK_REQUESTS -> atomic{
        ack.abort = 1;
        sending_ack++;
        ack_ch ! ack;
        sending_ack--;
        goto end
    }
    fi;

    retransmission:
    if
    :: empty(ch) -> atomic{/* Inactivity timer */
        ack.abort = 1;
        ack_ch ! ack;
        goto end
    }
    :: atomic{
        listening_ch++;
        ch ? fragment
        listening_ch--;}->
        if
        :: fragment.W != W && fragment.FCN != 0 &&
            fragment.all1 == 0 && fragment.ackRequest == 0 &&
            fragment.abort == 0  && local_W != WINDOWS -> atomic{
            /* Different W, Not-All and Not last w */
            W = !W;
            local_W++;
```

```
        c_bm = 1;
        attempt = 0;
        goto rcv_w
    }
    :: fragment.W != W && fragment.ackRequest == 1 &&
        fragment.abort == 0 && local_W != WINDOWS -> atomic{
        /* ACK Request from diferent window and not
        last window */
        W = !W;
        local_W++;
        c_bm = 0;
        attempt = 0;
        ack.bm = c_bm;
        ack.W = W;
        sending_ack++;
        ack_ch ! ack;
        sending_ack--;
        attempt++;
        goto rcv_w
    }
    :: fragment.W != W && fragment.FCN == 0 &&
        fragment.ackRequest == 0 && fragment.abort == 0 &&
        fragment.all1 == 0 && local_W != WINDOWS -> atomic{
        /* Different window, All-0 and not last window */
        W = !W;
        local_W++;
        attempt = 0;
        c_bm = 1
        ack.bm = c_bm;
        ack.W = W;
        ack.abort = 0;
        sending_ack++;
        ack_ch ! ack;
        sending_ack--;
        attempt++;
        goto retransmission
    }
    :: fragment.W != W && fragment.all1 == 1 &&
        fragment.ackRequest == 0 && fragment.abort == 0 &&
        local_W != WINDOWS-> /* Different window, All-1 */
        if
        :: true -> atomic{ /* RCS right */
            W = !W;
            local_W++;
            attempt = 0;
            c_bm = 1;
            ack.bm = c_bm;
```

```
            ack.W = W;
            ack.c = 1;
            ack.abort = 0;
            sending_ack++;
            ack_ch ! ack;
            sending_ack--;
            attempt++;
            goto clean_up
        }
        :: true -> atomic{ /* RCS wrong */
            W = !W;
            local_W++;
            attempt = 0;
            c_bm = 1;
            ack.bm = c_bm;
            ack.W = W;
            ack.c = 0;
            ack.abort = 0;
            sending_ack++;
            ack_ch ! ack;
            sending_ack--;
            attempt++;
            goto retransmission
        }
        fi;
    :: fragment.W == W && fragment.ackRequest == 0 &&
        fragment.abort == 0 && fragment.all1 == 0 &&
        local_W != WINDOWS ->
        /* Current window, not last window, not all-1 */
        c_bm++;
        if
        :: c_bm == WINDOWS_SIZE + 1 || fragment.FCN == 0->
            atomic{ /* c_bm full or All-0 */
            ack.W = W;
            ack.bm = c_bm;
            ack.abort = 0;
            sending_ack++;
            ack_ch ! ack;
            sending_ack--;
            attempt++;
            goto retransmission
        }
        :: else -> goto retransmission
        fi;
    :: fragment.W == W && fragment.all1 == 0 &&
        fragment.FCN == 0 && fragment.ackRequest == 0 &&
        fragment.abort == 0 && local_W == WINDOWS ->
```

```
        /* Current window, All-0, Last window */
        goto retransmission
    :: fragment.W == W && fragment.ackRequest == 1 &&
        local_W == WINDOWS -> atomic{
        /*ACK Request from current window, last window */
        ack.bm = c_bm;
        ack.W = W;
        ack.c = 0;
        ack.abort = 0;
        sending_ack++;
        ack_ch ! ack;
        sending_ack--;
        attempt++;
        goto retransmission;
    }
    :: fragment.abort == 1 -> atomic{
        /* On receiving sender-abort */
        ack.abort = 1;
        sending_ack++;
        ack_ch ! ack;
        sending_ack--;
        goto end
    }
    :: fragment.W == W && fragment.FCN == 0 &&
        fragment.all1 == 1 && fragment.ackRequest == 0 &&
        fragment.abort == 0 && local_W == WINDOWS ->
        /* All-1, current window, last window */
        if
        :: true -> atomic{ /* RCS right */
            c_bm++;
            ack.bm = c_bm;
            ack.W = W;
            ack.c = 1;
            ack.abort = 0;
            sending_ack++;
            ack_ch ! ack;
            sending_ack--;
            attempt++;
            goto clean_up
        }
        :: true -> atomic{ /* RCS wrong */
            c_bm++;
            ack.bm = c_bm;
            ack.W = W;
            ack.c = 0;
            ack.abort = 0;
            sending_ack++;
```

```
                ack_ch ! ack;
                sending_ack--;
                attempt++;
                goto retransmission
            }
            fi;
        :: fragment.W != W && fragment.abort == 0 &&
            local_W == WINDOWS -> atomic{
            /* Different window, currently in last window */
            goto retransmission;
        }
        :: fragment.W == W && fragment.FCN != 0 &&
            fragment.all1 == 0 && fragment.ackRequest == 0 &&
            fragment.abort == 0 && local_W == WINDOWS -> atomic{
            /* Regular fragment for last window, currently in last
            window */
            c_bm++;
            goto retransmission;
        }
        :: fragment.W == W && fragment.FCN == 0 &&
            fragment.all1 == 1 && fragment.ackRequest == 0 &&
            fragment.abort == 0 && local_W != WINDOWS -> atomic{
            /* All-1 for current window, but not last window */
            goto retransmission;
        }
        :: fragment.W == W && fragment.all1 == 0 &&
            fragment.ackRequest == 1 && fragment.abort == 0 &&
            local_W != WINDOWS -> atomic{
            /* ACK REQ for current window, not last w */
            ack.bm = c_bm;
            ack.W = W;
            ack.abort = 0;
            sending_ack++;
            ack_ch ! ack;
            sending_ack--;
            attempt++;
            goto retransmission
        }
        fi;

    :: attempt >= MAX_ACK_REQUESTS -> atomic{
        ack.abort = 1;
        sending_ack++;
        ack_ch ! ack;
        sending_ack--;
        goto end
    }
```

```
fi;

clean_up:
if
:: empty(ch) -> atomic{ /* Inactivity timer */
    ack.abort = 1;
    sending_ack++;
    ack_ch ! ack;
    sending_ack--;
    attempt++;
    goto end
}

:: atomic{
    listening_ch++;
    ch ? fragment
    listening_ch--;} ->
    if
    :: fragment.abort == 1 -> atomic{
        /* On receiving sender-abort */
        ack.abort = 1;
        sending_ack++;
        ack_ch ! ack;
        sending_ack--;
        goto end
    }
    :: fragment.W == W && fragment.all1 == 0 &&
        fragment.ackRequest == 1 && fragment.abort == 0 ->
        atomic{
        ack.bm = c_bm;
        ack.W = W;
        ack.abort = 0;
        sending_ack++;
        ack_ch ! ack;
        sending_ack--;
        attempt++;
        goto clean_up
    }
    :: fragment.W == W && fragment.all1 == 1 &&
        fragment.ackRequest == 0 && fragment.abort == 0 ->
        if
        :: true /* RCS right */ -> atomic{
            ack.bm = c_bm;
            ack.W = W;
            ack.c = 1;
            ack.abort = 0;
            sending_ack++;
```

```
                ack_ch ! ack;
                sending_ack--;
                attempt++;
                goto clean_up
            }
            :: true /* RCS wrong */ -> atomic{
                ack.bm = c_bm;
                ack.W = W;
                ack.c = 0;
                ack.abort = 0;
                sending_ack++;
                ack_ch ! ack;
                sending_ack--;
                attempt++; /* Increment attempts counter */
                goto clean_up
            }
            fi;
        :: else -> goto clean_up;
        fi;
    :: attempt >= MAX_ACK_REQUESTS -> atomic{
        ack.abort = 1;
        sending_ack++;
        ack_ch ! ack;
        sending_ack--;
        goto end
    }
    fi;

    end:
    printf("end state\n");
}
```

Anexo G

Código para transmisor en modo ACK-On-Error

```
active proctype Sender(){
    int TILES_PER_FRAGMENT = 1; /* Sigfox value = 1 */
    int MAX_ACK_REQUESTS = 5; /* Given by the profile */
    int MISSING_WINDOWS = N_WINDOWS; /* Windows left to send */
    int c_bm = 0;
    int p_FCN; /* Partial FCN used when resending missing tiles */
    Fragment ack;
    Fragment frag;

    int W = 0;
    int FCN = WINDOWS_SIZE;
    int attempts = 0;

    send_state:
    if
    :: FCN != 0 -> atomic{
        frag.bm = TILES_PER_FRAGMENT;
        frag.FCN = FCN;
        frag.W = W;
        frag.all1 = 0;
        frag.abort = 0;
        sending_ch++;
        ch ! frag;
        sending_ch--;
        FCN = FCN - TILES_PER_FRAGMENT;
        c_bm = c_bm + TILES_PER_FRAGMENT;
        goto send_state;
    }
    :: FCN == 0 && MISSING_WINDOWS == 0 -> atomic{
        /* Last fragment, send all-1 */
```

```
        frag.bm = TILES_PER_FRAGMENT;
        frag.W = W;
        frag.FCN = 0;
        frag.all1 = 1;
        frag.abort = 0;
        sending_ch++;
        ch ! frag;
        sending_ch--;
        c_bm = c_bm + TILES_PER_FRAGMENT;
        MISSING_WINDOWS--;
        goto wait_for_ack;
    }
    :: FCN == 0 && MISSING_WINDOWS > 0 -> atomic{
        /* Last tile of window */
        frag.W = W;
        frag.FCN = 0;
        frag.all1 = 0;
        frag.abort = 0;
        frag.bm = TILES_PER_FRAGMENT;
        sending_ch++;
        ch ! frag;
        sending_ch--;
        c_bm = c_bm + TILES_PER_FRAGMENT;
        MISSING_WINDOWS--;
        goto wait_for_ack;
    }
    fi;

    wait_for_ack:
    if
    :: empty(ack_ch) -> /* Retransmission timer expiration */
        if
        :: attempts < MAX_ACK_REQUESTS && MISSING_WINDOWS < 0 ->
            atomic{ /* Send ACK Request for current window */
            frag.W = W;
            frag.bm = TILES_PER_FRAGMENT;
            frag.FCN = 0;
            frag.all1 = 1;
            frag.abort = 0;
            sending_ch++;
            ch ! frag;
            sending_ch--;
            attempts++;
            goto wait_for_ack;
        }
        :: attempts < MAX_ACK_REQUESTS && MISSING_WINDOWS >= 0 ->
            atomic{ /* There are windows to send */
```

```
            W++;
            FCN = WINDOWS_SIZE;
            c_bm = 0;
            goto send_state;
        }
        :: attempts >= MAX_ACK_REQUESTS -> atomic{
            /* Send Sender Abort */
            frag.abort = 1;
            frag.bm = 0;
            frag.FCN = 0;
            frag.W = W;
            frag.all1 = 0;
            sending_ch++;
            ch ! frag;
            sending_ch--;
            goto error;
        }
        fi;

    :: nempty(ack_ch) ->
        listening_ack++;
        ack_ch ? ack;
        listening_ack--;
        if
        :: ack.abort == 1 -> /*Receiver Abort*/
            goto error;

        :: ack.W == N_WINDOWS && ack.bm == c_bm && ack.c == 0 &&
            ack.abort != 1 -> atomic{
            /* Last window of packet, bad reasseambly */
            frag.abort = 1;
            frag.bm = 0;
            frag.FCN = 0;
            frag.W = W;
            frag.all1 = 0;
            sending_ch++;
            ch ! frag;
            sending_ch--;
            goto error;
        }

        :: ack.W == N_WINDOWS &&  ack.bm == c_bm && ack.c == 1 &&
            ack.abort == 0 -> atomic{
            /*Last window of packet all tiles received */
            goto end;
        }
```

```
    :: ack.bm != c_bm && ack.abort == 0 -> atomic{
        attempts = 0;
        p_FCN = WINDOWS_SIZE - ack.bm;
        goto resend_missing_frags;
    }
    fi;
fi;

resend_missing_frags:
if
:: p_FCN >= 0 -> atomic{
    frag.FCN = p_FCN;
    frag.W = ack.W;
    frag.bm = TILES_PER_FRAGMENT;
    frag.abort = 0;
    frag.all1 = 0;
    if
    :: ack.W == N_WINDOWS -> /* All-1 */
        frag.all1 = 1;
    :: else -> /* All-0 */
        frag.all1 = 0;
    fi;
    sending_ch++;
    ch ! frag;
    sending_ch--;
    p_FCN = p_FCN - TILES_PER_FRAGMENT;
    goto resend_missing_frags;
}
:: else -> skip;
fi;

if
:: W != N_WINDOWS -> atomic{
    /* Not last window, keep sending next windows */
    W++;
    FCN = WINDOWS_SIZE;
    c_bm = 0;
    goto send_state;
}
:: W == N_WINDOWS && ack.W != N_WINDOWS -> atomic{
    /* Last window sent and not last window resend*/
    frag.FCN = 0;
    frag.W = N_WINDOWS;
    frag.bm = TILES_PER_FRAGMENT;
    frag.abort = 0
    frag.all1 = 1;
    sending_ch++;
```

```
            ch ! frag;
            sending_ch--;
            goto wait_for_ack;
        }
        :: W == N_WINDOWS && ack.W == N_WINDOWS -> atomic{
            /* Not to send last tile twice */
            goto wait_for_ack;
        }
        fi;

        error:
        printf("Error state\n");

        end:
        printf("End state\n");

}
```

Anexo H

Código para receptor en modo ACK-On-Error

```
active proctype Receiver(){
    chan missingFragments = [ 4 ] of { int, int }; /* Queue of
    amount of missing fragments per window. Format: (window,
    amount of received tiles) */
    int m_window;
    int m_window_bm;
    bit tile_received = 0;

    Fragment frag;
    Fragment ack;
    int attempts = 0;
    int W = 0;
    int c_bm = 0;
    int expected_FCN = WINDOWS_SIZE;
    int MAX_ACK_REQUESTS = 5; /* Given by the profile */

    rcv_window:
    if
    :: empty(ch) && tile_received -> /* Inactivity timer */
        goto wait_to_abort;

    :: atomic{
        listening_ch++;
        ch ? frag;
        tile_received = 1;
        listening_ch--;} ->
        if
        :: frag.abort == 1 -> goto error;

        :: frag.W == W && frag.all1 == 0 && frag.FCN != 0 &&
```

```
        frag.abort == 0 -> atomic{
        /* Regular fragment for current window */
        c_bm = c_bm + frag.bm;
        expected_FCN = frag.FCN - frag.bm;
        goto rcv_window;
    }

    :: frag.W == W && frag.FCN != expected_FCN &&
        (frag.all1 == 1|| frag.FCN == 0) && frag.abort == 0 ->
        atomic{
        /* Not expected All-0 or All-1, current window */
        c_bm = c_bm + frag.bm;
        missingFragments ! W, c_bm;
        missingFragments ? m_window, m_window_bm;
        ack.bm = m_window_bm;
        ack.W = m_window;
        ack.c = 0;
        ack.abort = 0;
        sending_ack++;
        ack_ch ! ack;
        sending_ack--;
        goto wait_missing_frags;
    }

    :: frag.W == W && frag.FCN == expected_FCN &&
        (frag.FCN == 0||frag.all1 == 1) && frag.abort == 0 ->
        atomic{ /* Expected All-0 or All-1 */
        c_bm = c_bm + frag.bm;
        if /* Check if tiles lost in this window */
        :: c_bm != WINDOWS_SIZE + 1 ->
            missingFragments ! W, c_bm;
        :: else -> skip;
        fi;

        if
        :: nempty(missingFragments) ->
            missingFragments ? m_window, m_window_bm;
            ack.bm = m_window_bm;
            ack.W = m_window;
            ack.c = 0;
            ack.abort = 0;
            sending_ack++;
            ack_ch ! ack;
            sending_ack--;
            goto wait_missing_frags;
        :: empty(missingFragments) ->
            if
```

```
            :: frag.all1 == 1 -> /* If All-1 then end */
                ack.bm = c_bm;
                ack.W = W;
                ack.c = 1;
                ack.abort = 0;
                sending_ack++;
                ack_ch ! ack;
                sending_ack--;
                goto end;
            :: else -> /* If All-0 then receive next window */
                c_bm = 0;
                W++;
                expected_FCN = WINDOWS_SIZE;
                goto rcv_window;
            fi;
        fi;
    }

    :: frag.W != W && frag.abort == 0 -> atomic{
        missingFragments ! W, c_bm; /* Last window received */
        W++;
        do
        :: W != frag.W ->
            missingFragments ! W, 0;
            W++;
        :: else ->
            c_bm = frag.bm; /* Bitmap of current window */
            break;
        od;

        if
        :: frag.all1 == 1 || frag.FCN == 0 ->
            /* All-1 or All-0 */
            missingFragments ! W, c_bm; /* This window is
            also lost */
            missingFragments ? m_window, m_window_bm;
            ack.bm = m_window_bm;
            ack.W = m_window;
            ack.c = 0;
            ack.abort = 0;
            sending_ack++;
            ack_ch ! ack;
            sending_ack--;
            goto wait_missing_frags;

        :: else -> /* Regular fragment */
            expected_FCN = frag.FCN - frag.bm;
```

```
                goto rcv_window;
            fi;
        }
        fi;

fi;

wait_to_abort:
atomic{
listening_ch++;
ch ? frag;
listening_ch--;}
if
:: frag.all1 == 1 || frag.FCN == 0 -> atomic{
    /* All-1 or All-0 */
    ack.abort = 1;
    sending_ack++;
    ack_ch ! ack;
    sending_ack--;
    goto error;
}

:: frag.abort == 1 -> /* Sender abort received */
    goto error;
:: else -> /* Reg frag rcv, keeps waiting */
    goto wait_to_abort
fi;

wait_missing_frags:
if
:: empty(ch) || attempts >= MAX_ACK_REQUESTS ->
    /* Timer expiration or too many attempts */
    goto wait_to_abort;

:: atomic{
    listening_ch++;
    ch ? frag
    listening_ch--;} ->
    if
    :: frag.abort == 1 -> goto error;

    :: frag.W == m_window && (frag.FCN != 0 || frag.all1 == 0)
        && frag.abort == 0 -> atomic{
        m_window_bm = m_window_bm + frag.bm;
        attempts = 0;
        goto wait_missing_frags;
    }
```

```
    :: frag.all1 == 1 && frag.abort == 0 &&
        m_window_bm + frag.bm != WINDOWS_SIZE -> atomic{
        ack.bm = m_window_bm;
        ack.W = m_window;
        ack.c = 0;
        ack.abort = 0;
        sending_ack++;
        ack_ch ! ack;
        sending_ack--;
        attempts++;
        goto wait_missing_frags;
    }

    :: frag.all1 == 1 && frag.abort == 0 &&
        m_window_bm + frag.bm == WINDOWS_SIZE -> atomic{
        ack.bm = m_window_bm;
        ack.W = m_window;
        ack.c = 1;
        ack.abort = 0;
        sending_ack++;
        ack_ch ! ack;
        sending_ack--;
        goto end;
    }

    :: frag.W != m_window && frag.abort == 0 &&
        frag.all1 == 0 -> atomic{
        /* Determine if lost tiles */
        W++;
        do
        :: W != frag.W ->
            missingFragments ! W, 0;
            W++;
        :: else ->
            c_bm = frag.bm; /* Bitmap of current window */
            break;
        od;
        expected_FCN = frag.FCN - frag.bm;
        attempts = 0;
        goto rcv_window;
    }

    fi;

fi;
```

```
    error:
    printf("Error state\n");

    end:
    atomic{
    listening_ch++;
    ch ? frag;
    listening_ch--;}
    if
    :: frag.all1 == 1 && frag.abort == 0 -> atomic{
        ack.bm = c_bm;
        ack.W = W;
        ack.c = 1;
        ack.abort = 0;
        goto end;
    }
    :: else -> goto end;
    fi;
}
```