Universidad Tecnológica Nacional
Facultad Regional Bahía Blanca

Ing. Electrónica

Proyecto Final de Carrera

Diseño, desarrollo y verificación de software para comunicar dos o más computadoras mediante puerto Ethernet, con un dispositivo dedicado.

Autor
Joel Ermantraut

Director
Ing. Christian, Galasso

Tutor
Mg. Guillermo Friedrich

Bahía Blanca | 14 de Agosto de 2024

Resumen

En el presente documento se describe el diseño e implementación de una aplicación que sucede el desarrollo de un software que permitió validar la hipótesis de que una computadora naval de propósito específico podría vincularse con una computadora moderna. Respecto a su versión previa, la aquí detallada fue optimizada y ampliada para trabajar en dos computadoras, conectadas mediante Ethernet, y a su vez comunicadas, una directa y la otra indirectamente (a través de la primera), con la computadora naval del buque. Cabe aclarar que la consola original a reemplazar (cuyo modelo 3D se aprecia en la parte IZQ de la Figura n°1) es operada en simultáneo por dos operadores y con el presente desarrollo se pretende lograr el reemplazo por dos consolas mono-usuario (cuyo modelo 3D se aprecia en la parte DER de la Figura n°1), dado que esto último coincide con el estado del arte de este tipo de sistemas.

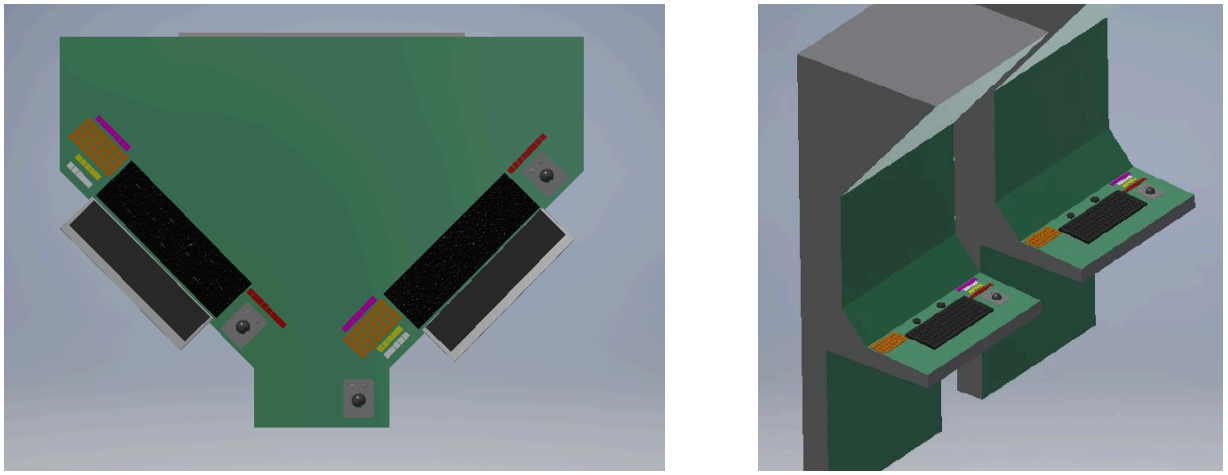


FIGURA (1). IZQ. CONSOLA ORIGINAL PARA 2 OPERADORES. DER. 2 CONSOLAS PARA UN OPERADOR CADA UNA.

Palabras clave: software, optimización, verificación, computadora naval, Ethernet.

Índice

1. Introducción

La Base Naval de Puerto Belgrano (BNPB) es una base militar de la Armada Argentina. Se localiza en el Partido de Coronel de Marina Leonardo Rosales, en las proximidades de la ciudad de Punta Alta, al sur de la provincia de Buenos Aires. Cuenta con infraestructura suficiente para alojar y reparar buques de gran tamaño. La principal tarea de estas embarcaciones es vigilar y resguardar el espacio marítimo argentino, controlar que se respeten los límites de la zona costera, expulsando tripulaciones ajenas. Un grupo de estos barcos fueron adquiridos en la década del 80, por lo que su tecnología es antigua en comparación con las disponibles hoy. Si bien, se han adquirido nuevas unidades con tecnologías actuales, es deseable desde el punto de vista del conocimiento y para desarrollar capacidades, poder practicar modernizaciones de sus distintos subsistemas.

Actualmente, los barcos poseen una computadora naval de propósito específico que toma los datos provenientes de periféricos de entrada[1], de sensores[2], y de otros subsistemas del mismo barco, los procesa y los muestra en los periféricos de salida[3]. Los periféricos de entrada y salida se encuentran distribuidos en consolas de manera que sea más sencilla la experiencia del usuario encargado de manejarla. El objetivo de este trabajo es diseñar e implementar una aplicación de software que sea capaz de reemplazar dicha consola sin que la computadora naval note la sustitución.

Idealmente, también se debería reemplazar la computadora naval, pero es un trabajo mucho más complejo ya que se debe implementar su lógica interna, por lo tanto se optó por comenzar solo con los periféricos. Este reemplazo es esencial para modernizar los sistemas de a bordo de manera progresiva.

[1] Los cuales son los teclados, tracking ball y botoneras de las consolas de operaciones.
[2] Radar, Sonar, otros.
[3] Las pantallas de información escrita (Que llamaremos AND) y de información gráfica (Que llamaremos LPD).

Antecedente de desarrollo

Este proyecto tiene su punto de partida en un trabajo previo [1], presentado como Proyecto Final de Carrera por el Ing. Manuel Fernandez, año 2021. El mismo consistió en el desarrollo de una primera versión de una aplicación estándar que permitió validar parcial e indirectamente el buen funcionamiento de un hardware diseñado "ad-hoc" para enlazar una computadora comercial con la computadora dedicada del buque. El desarrollo de este hardware también fue en modalidad de Proyecto Final de Carrera [2] de los Ing. Loidi y Lutri. Esta primer aplicación fue desarrollada en lenguaje de Python, empleando PyQt5[4] como soporte gráfico. La misma fue implementada y depurada hasta un correcto funcionamiento. Sin embargo durante las pruebas a bordo, los usuarios del sistema notaron que ante una cierta carga de información, colapsaba y dejaba de funcionar. Un análisis minucioso de la falla, por parte del equipo de trabajo, determinó que el lenguaje empleado no estaba optimizado para los requerimientos del problema, especialmente para responder a la necesidad de generar gráficos dinámicos en tiempo real. Esto motivó la búsqueda de una tecnología alternativa de software que permitiera obtener una aplicación estable en todas las condiciones de uso y/o carga del sistema que pudieran darse a bordo. Se impuso además los siguiente requerimientos:

1) Que permita lograr una aplicación estable y con muy baja latencia en los tiempos de respuesta al operador.
2) Que se pudiera obtener una aplicación multiplataforma (Linux/Windows).
3) Que fuera un lenguaje de alto nivel, para poder hacer el desarrollo en un tiempo aproximado de 10 meses.
4) Que permitiera aprovechar los recursos de hardware de la computadora sobre la que se lo ejecute (PE: la GPU).

[4] https://pypi.org/project/PyQt5/

5) Que permita la división de la aplicación para su uso simultáneo por dos operadores en dos puestos de trabajo o computadoras. Esto se describe con más detalle en la siguiente sección.

División de los puestos de operaciones

Además del cambio de tecnología, se planteó la división de la aplicación para que funcionara en 2 puestos de trabajo individuales. La consola original a reemplazar, como puede verse en la figura 2, es un hardware preparado para albergar 2 operadores en simultáneo.

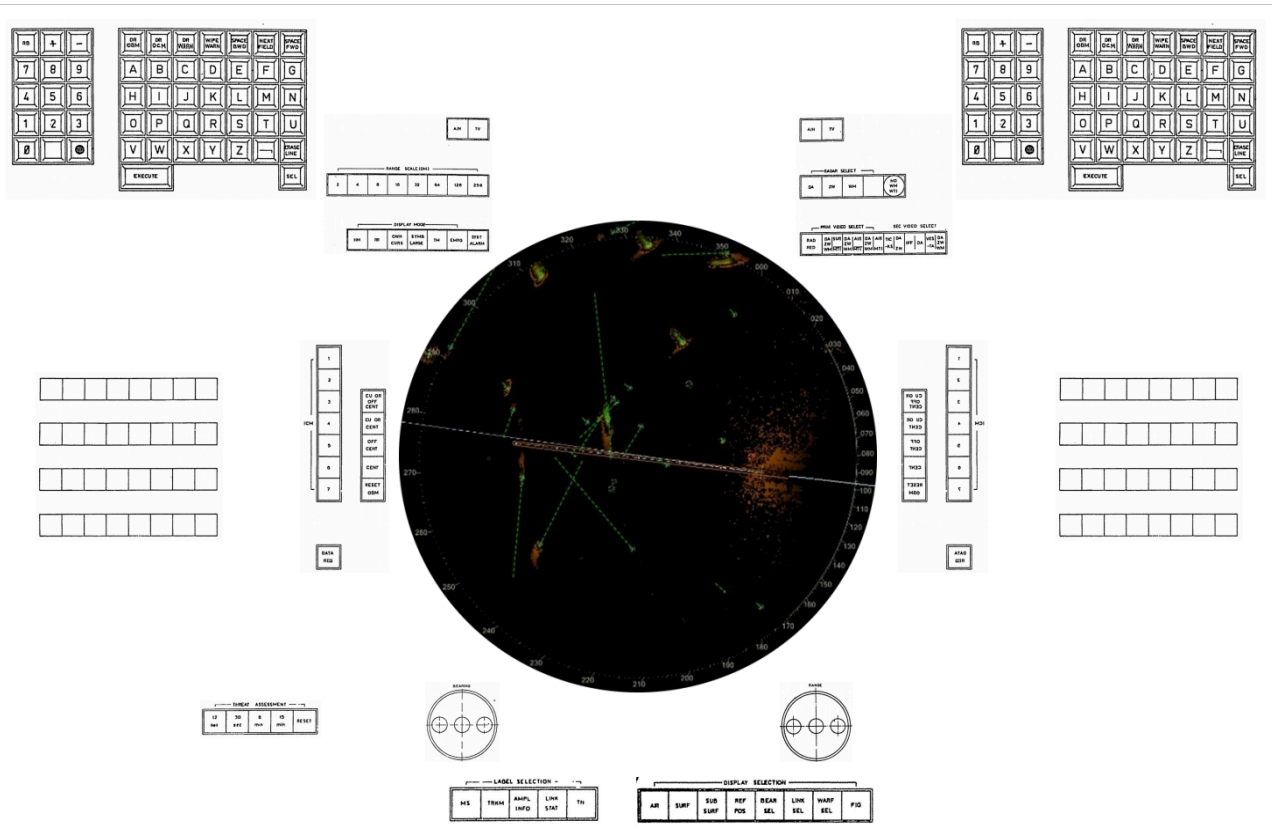


FIGURA (2). VISTA SUPERIOR DE LA CONSOLA ORIGINAL PARA 2 OPERADORES.

En la figura 3 se puede observar como es la representación digital de la primera aplicación desarrollada que reemplaza la consola pero sobre una sola computadora.

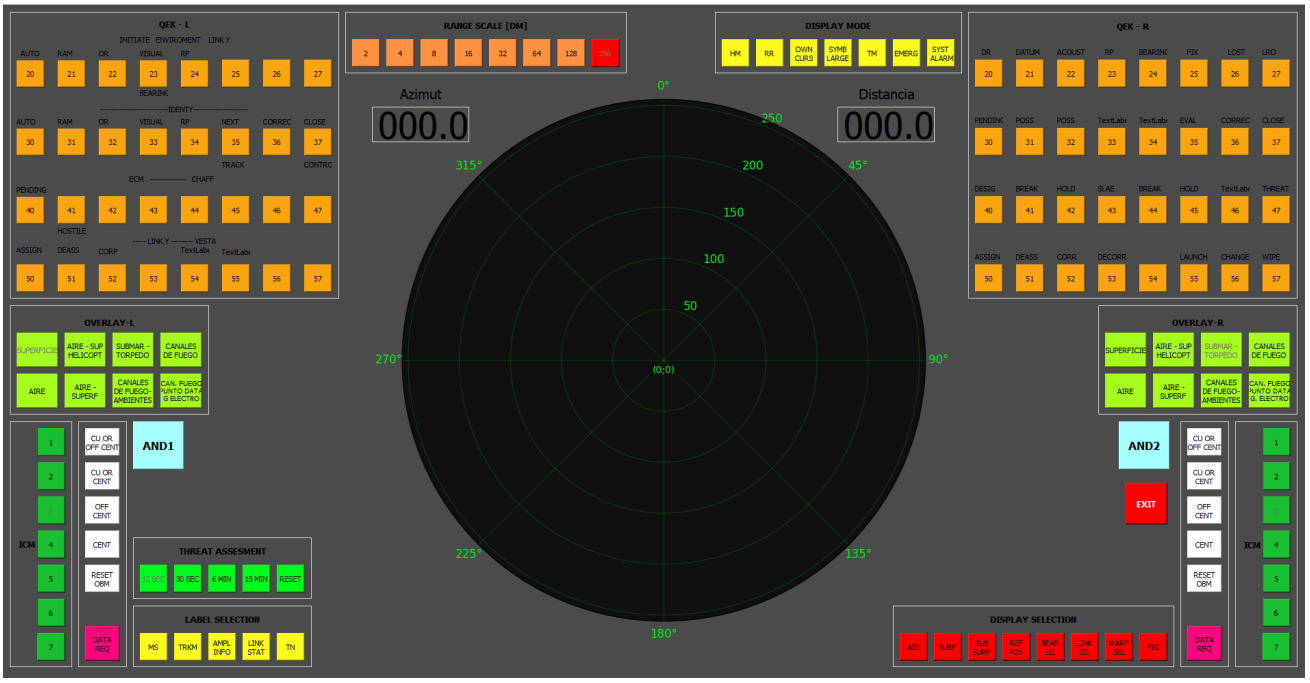


FIGURA (3). VISTA DE LA PANTALLA DE LA PRIMERA APLICACIÓN DESARROLLADA EN PYTHON.

Esta forma de operación es conceptualmente obsoleta y se pretende cambiarla, para conformar el sistema al estado del arte de las centrales de operaciones de los buques actuales. Reemplazando el puesto doble, por dos puestos individuales como se presentó en el resumen y se repite en la figura 4.

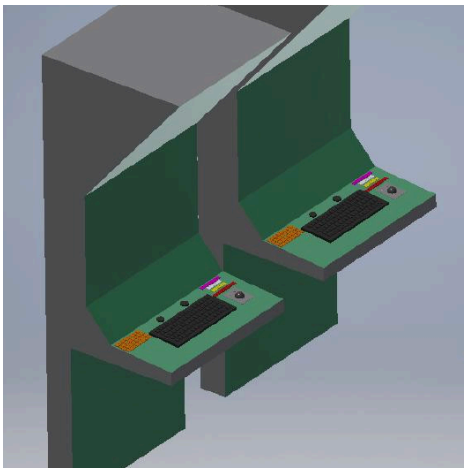


FIGURA (4). VISTA EN PERSPECTIVA DE DOS PUESTOS INDIVIDUALES PARA DOS OPERADORES.

Esta división tiene requerimientos asociados a necesidades de los operadores y a condiciones de borde que vienen impuestas por el hecho que se está “emulando” un sistema de características

particulares. Para la computadora naval original, lo que tiene conectado, es la consola original de operaciones donde trabajan en simultáneo dos usuarios. Por ende:

1) La visualización de la pantalla de información gráfica (En adelante LPD) antes era única y ahora se replicará en dos pantallas, con lo que se debe sincronizar la información en las dos nuevas representaciones de la LPD, para que ambos operadores puedan ver lo mismo todo el tiempo.
2) Existe una sola conexión física entre la computadora naval y la consola, por tanto de los dos nuevos puestos de trabajo, uno será el que se vinculará físicamente, mientras que el otro puesto o computadora deberá usar al primero como gateway o concentrador de información para poder conectarse.
3) Se debe establecer de la aplicación original que posee la funcionalidad desarrollada de todos los botones, que grupo deben dejarse en la primera consola, que grupo en la segunda y cuales son los botones particulares que requieren estar en ambas consolas replicados.
4) Y dado que hay botones que deben replicarse, debe implementarse un método de selección que ante dos estados diferentes de los botones compartidos por ambos puestos, se le dé prioridad a uno.
5) La comunicación entre las PC que conforman los dos nuevos puestos de trabajo debe ser exclusivamente Ethernet. Pudiendo ésta, ser cableada o WiFi.

Estructura del proyecto

En primer lugar se presenta un diagrama general de la arquitectura del sistema sobre el que se realizará el desarrollo para dimensionar la problemática. Luego, se detalla la solución diseñada, con sus correspondientes modificaciones hasta llegar al producto final. Una vez conocida la solución, se descompondrá la misma en los diferentes archivos que lo integran, y una explicación de cada uno y su contenido.

2. Diagrama general

Como se mencionó anteriormente, cada buque posee una computadora central, denominada GMD (Gabinete de Manejo de Datos) que es la encargada del procesamiento de los datos. Este gabinete se comunica con las distintas consolas mediante sus periféricos de entrada y salida. A continuación se ilustra en la figura (1):

Figura (5). Diagrama GMD – Consola

A su vez, cada consola posee una pantalla radar, uno o dos monitores y varios botones, como se muestra a continuación:

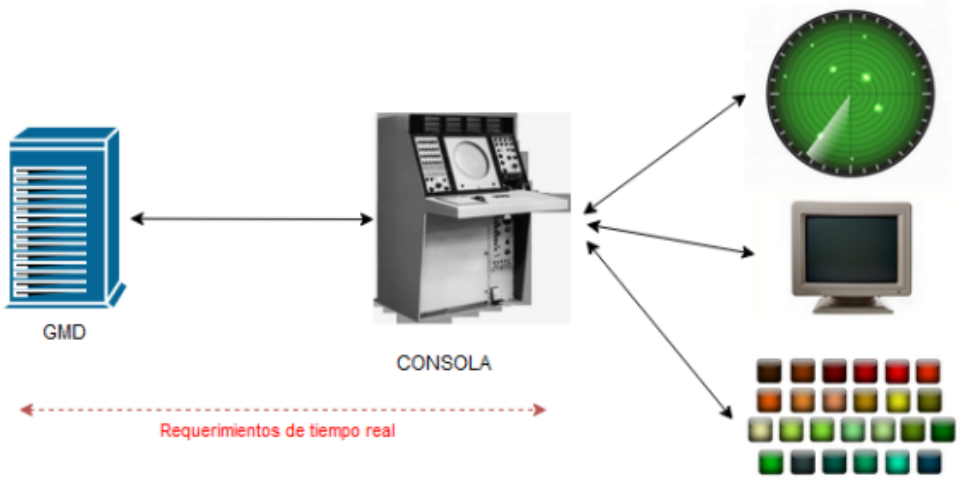


Figura (6). Diagrama GMD - Consola con sus respectivas partes.

Como se observa en la figura (5), la comunicación entre ambos dispositivos tiene requerimientos de tiempo real. El puerto original es del tipo RS 422 con una implementación por hardware “ad hoc”, y con una latencia del orden de los microsegundos. Se impuso como requisito[5] que la PC se enlazara al GMD mediante un puerto Ethernet de 100Mb (con una latencia significativamente mayor, del orden de las centenas de microsegundos) y que el sistema operativo sobre el que corriera la aplicación fuera de tiempo diferido (incompatible con tiempo real). Se decidió reemplazar la placa original que vincula el GMD con la consola, por una diseñada para resolver las incompatibilidades temporales mediante máquinas de estado implementadas sobre una FPGA.

El resultado final, luego de la implementación del software que describe esta documentación, se representa en la siguiente figura:

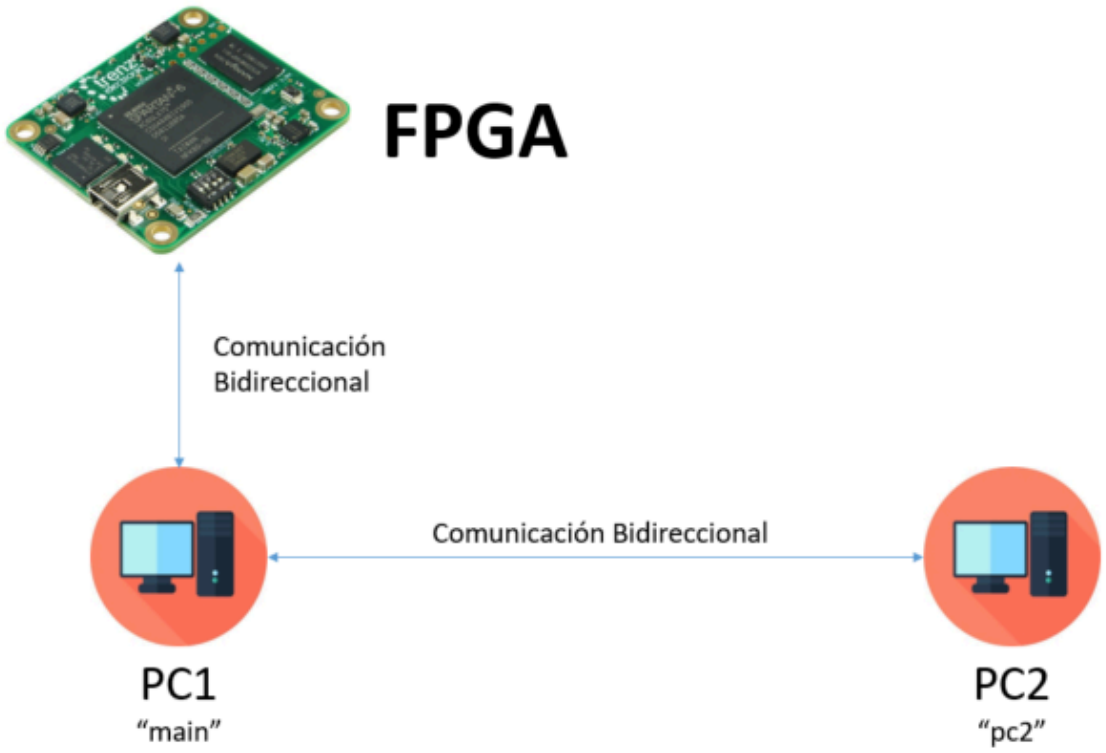


FIGURA (7). ESQUEMA DEL PROTOTIPO. LA FPGA SE ENCUENTRA DENTRO DEL GMD.

[5] Los requisitos fueron impuestos al principio del proyecto por el SIAG y no se detallan en el presente documento.

En la imagen se puede ver lo descrito previamente, donde una de las PCs, (en el software representada como “main”) se comunica con la FPGA mediante un socket. Además, esta también se comunica con una segunda PC (representada como “pc2”), mediante un socket bidireccional. Nos referiremos a las mismas con las notaciones “main” y “pc2” en el resto del documento.

3.Solución

Como se mencionó previamente, en este punto del desarrollo había un problema de rendimiento en la aplicación. La interfaz radar se carga con información propia del entorno actual sobre el cual se está navegando, y el dispositivo dedicado del buque ofrece la posibilidad de agregar un volumen importante de indicadores e imágenes que faciliten a los operadores brindar información y tomar decisiones. Por ese motivo, cuando se cargaba cierto volumen de imágenes, la aplicación saturaba los recursos del equipo comercial, y procedía a cerrarse. Esto denotaba un grave problema de rendimiento, sobre todo en el aspecto gráfico.

Lo primero fue hacer un diagnóstico. La respuesta más intuitiva es analizar los recursos demandados por la aplicación. Se terminó identificando que debido a su implementación, la misma solo hacía uso del procesador para todas las operaciones, lógicas y gráficas. Intuitivamente, esto es un problema, dado que cualquier equipo tiene una unidad dedicada al procesamiento gráfico, denominada GPU (Unidad de procesamiento gráfico, del inglés Graphics Processing Unit). Luego, también se realizó una lectura del código fuente de la aplicación, identificando algunas mejoras a nivel de código, pero que no produjeron grandes mejoras en el funcionamiento.

Conociendo el problema, se buscó una solución. Un primer intento consistió en usar algunas herramientas que permitieran acelerar el funcionamiento del procesamiento gráfico, forzar la delegación de estas tareas a la GPU, o realizando una pre-compilación de fragmentos del código, y así evitar que las tareas repetitivas saturen los recursos. Para la primera opción se estudió la documentación de PyQt5, y para las segunda se hicieron pruebas con dos módulos Numba[6] y JAX[7].

[6] https://numba.pydata.org/

[7] https://jax.readthedocs.io/en/latest/notebooks/quickstart.html

Ambos son compiladores JIT (Justo a tiempo, del inglés Just in Time). Se facilitan algunas fuentes[8] para profundizar en estos conceptos.

Estas herramientas lograron una pequeña ventaja en la aceleración, pero con resultados insuficientes.

Por este motivo, se estudió otra opción, que involucra mudar el proyecto a otra plataforma. Con esta alternativa en mente, la preferencia popular hubiera sido mover el proyecto a un lenguaje de menor nivel. Incluso usando Qt, trasladarlo a C++ podría haber sido una opción exitosa. El gran conflicto en este aspecto, es que ya en este punto el proyecto en su totalidad era grande, y trasladarlo a un lenguaje de menor nivel, con todo lo que esto significa, requería tiempo y personal a disposición, dos variables de las que el equipo carecía.

Con el propósito que conseguir avances en menor tiempo y aún así lograr resolver el problema, se evaluó el uso de la plataforma Electron.JS[9]. Electron es un framework para crear aplicaciones de escritorio usando JavaScript, HTML y CSS. Embebiendo Chromium y Node.js dentro del mismo, Electron permite mantener una base de código JavaScript y crear aplicaciones multiplataforma que funcionan en Windows, macOS y Linux, sin conocimiento o experiencia en desarrollo nativo.

Conceptualmente hablando, la mayor ventaja de esta solución es que el código desarrollado en JavaScript en código en alto nivel, por lo cual, trasladar la aplicación de Python a JavaScript resultaría relativamente sencillo. Respecto al rendimiento, todo el procesamiento gráfico se realiza mediante el navegador integrado, en este caso Chromium[10]. Chromium es el proyecto de fondo detrás del navegador Google Chrome, unos de los navegadores más funcionales, líder del mercado. Los navegadores como tienen sumamente depurado el uso distribuido entre CPUs y GPUs, por lo cual, responden de forma óptima a la necesidad del problema.

[8] http://www.ub.edu/gidea/recursos/casseat/JIT_concepte_carac.pdf

[9] https://www.electronjs.org/es/docs/latest/

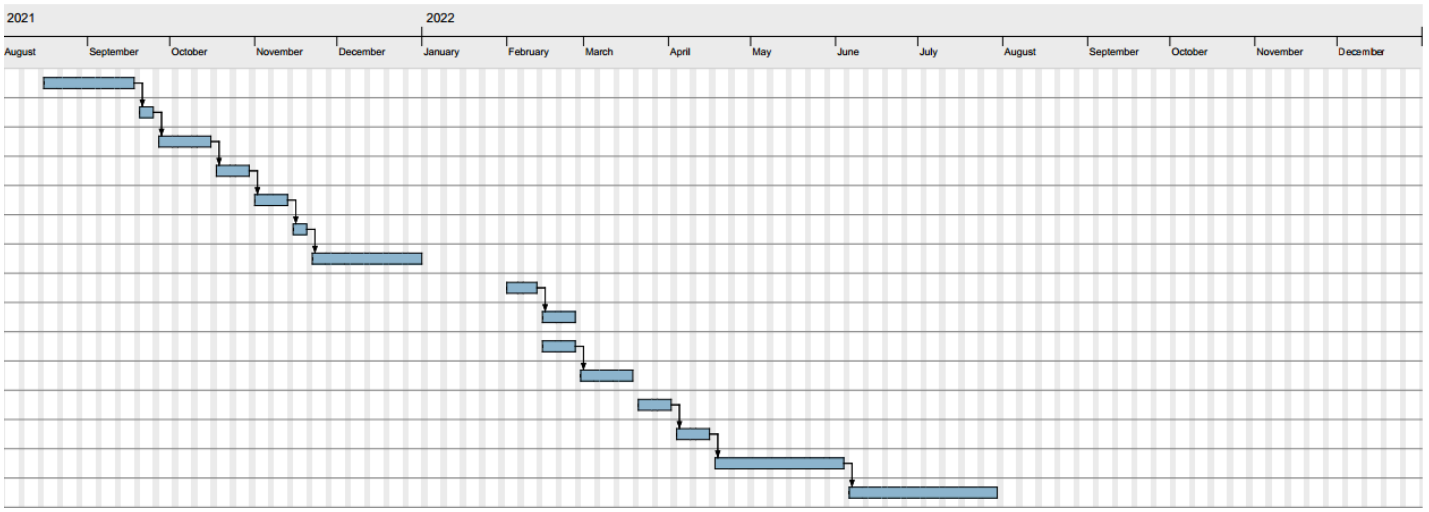
[10] https://www.chromium.org/Home/

Para no perder tiempo en vano, se propuso una implementación parcial de esta solución. Se reemplazó únicamente el componente dedicado a graficar el radar, dejando la comunicación con la placa y el procesamiento de los datos en Python. Para comunicar ambas tecnologías, se utilizaron sockets. De esta manera, luego de verificar que la solución ofrecía alguna ventaja, se trasladó el resto del código a Electron.JS. En este punto del desarrollo, solamente restó trasladar el panel de botones, por una cuestión de tiempo.

Finalmente, se implementó la división de la aplicación en dos computadoras. En este proceso, es meritorio mencionar el aporte del becario del proyecto, Sergio Leoni, que para conseguir acelerar los tiempos de desarrollo, dispuso algunas semanas el traslado de la lógica del proyecto desde Python a Electron.JS. Inicialmente, esto fue un gran aporte, pero dadas algunas dificultades que surgieron durante la depuración, terminaron atrasando el desarrollo. Sin embargo esto era necesario, y en algún momento debía realizarse.

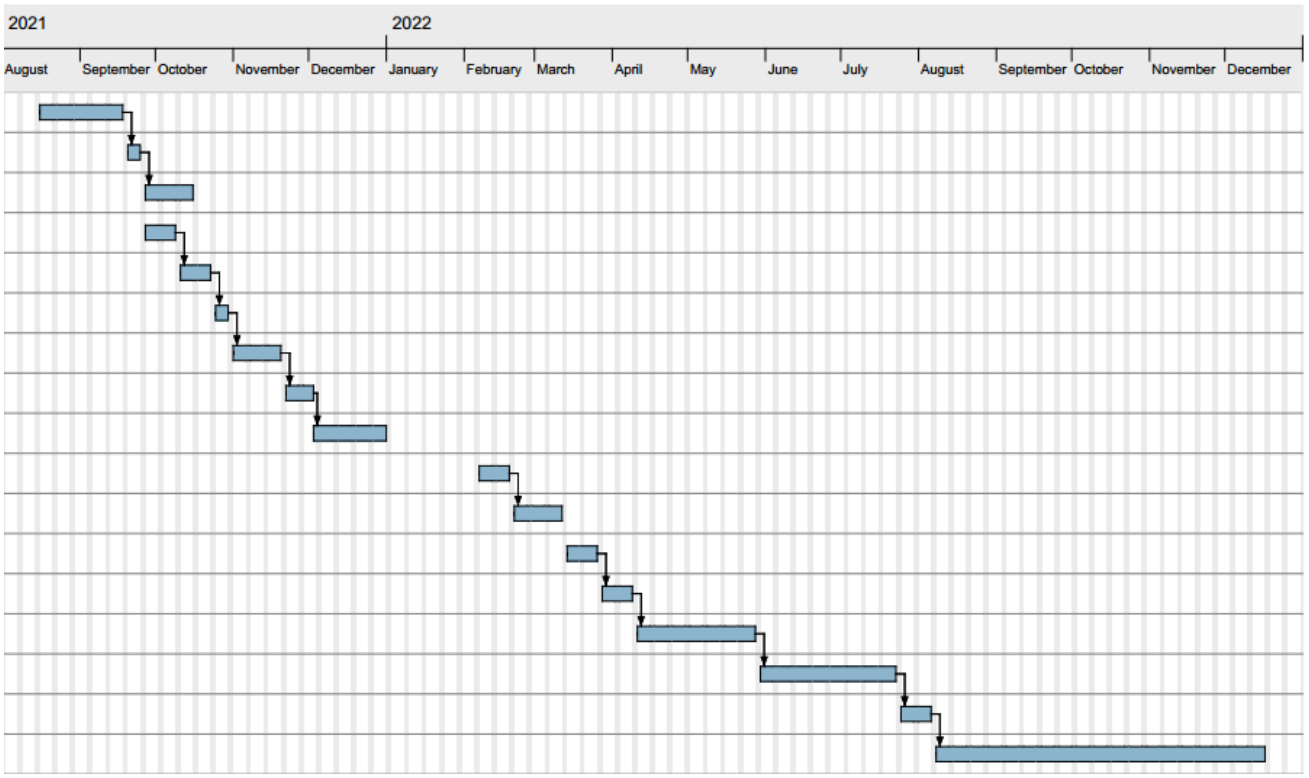
4. Diagrama de tiempos

Basado en la premisa presentada previamente, se estimaron las actividades y tiempos necesarios para cada una. En algunos casos, estos fueron respetados. En otros, se extendieron notoriamente. A continuación, se presenta un diagrama de Gantt, que detalla los tiempos previstos, antes de comenzar el proyecto:



Nombre	Fecha de inicio	Fecha de fin	Duración (días)
Estudio del desarrollo actual	16/8/2021	17/9/2021	25
Optimización del software	20/9/2021	24/9/2021	5
Pruebas con algunas técnicas para acelerar el software	27/9/2021	15/10/2021	15
Investigación de la solución	18/10/2021	29/10/2021	10
Diseño de la solución	1/11/2021	12/11/2021	10
Presentación de la solución	15/11/2021	19/11/2021	5
Desarrollo de la interfaz radar, que reemplaza al módulo RadarWidget.	22/11/2021	31/12/2021	30
Desarrollo de la comunicación con sockets, para probar la interfaz nueva.	1/2/2022	11/2/2022	9
Pruebas sobre la solución implementada	14/2/2022	25/2/2022	10
Pruebas de comunicación entre PCs con diferentes sistemas operativos	14/2/2022	25/2/2022	10
Desarrollo del módulo de comunicación por sockets	28/2/2022	18/3/2022	15
Desarrollo de la división de la interfaz	21/3/2022	1/4/2022	10
Incorporación de la interfaz en el software	4/4/2022	15/4/2022	10
Desarrollo de la funciones pendientes	18/4/2022	3/6/2022	35
Pruebas	6/6/2022	29/7/2022	40

Sin embargo, dadas las dificultades, los tiempos cambiaron. El siguiente diagrama presenta los tiempos reales del proyecto en cuestión:



Nombre	Fecha de inicio	Fecha de fin	Duración (días)
Estudio del desarrollo actual	16/8/2021	17/9/2021	25
Optimización del software	20/9/2021	24/9/2021	5
Pruebas con algunas técnicas para acelerar el software	27/9/2021	15/10/2021	15
Investigación de la solución	27/9/2021	8/10/2021	10
Diseño de la solución	11/10/2021	22/10/2021	10
Presentación de la solución	25/10/2021	29/10/2021	5
Desarrollo de la interfaz radar, que reemplaza al módulo RadarWidget.	1/11/2021	19/11/2021	15
Desarrollo de la comunicación con sockets, para probar la interfaz nueva.	22/11/2021	2/12/2021	9
Pruebas sobre la solución implementada	3/12/2021	31/12/2021	21
Pruebas de comunicación entre PCs con diferentes sistemas operativos	7/2/2022	18/2/2022	10
Desarrollo del módulo de comunicación por sockets	21/2/2022	11/3/2022	15
Desarrollo de la división de la interfaz	14/3/2022	25/3/2022	10

Incorporación de la interfaz en el software	28/3/2022	8/4/2022	10
Desarrollo de la funciones pendientes	11/4/2022	27/5/2022	35
Pruebas	30/5/2022	22/7/2022	40
Correcciones sobre las pruebas	25/7/2022	5/8/2022	10
Pruebas sobre las correcciones	8/8/2022	16/12/2022	95

En el último diagrama se evidencia que la gran extensión de los tiempos se dio debido a que el periodo de pruebas se extendió mucho más de lo previsto. Esto fue un cúmulo de dificultades organizacionales y técnicas, sumada a la inexperiencia en la formulación de este tipo de planificaciones.

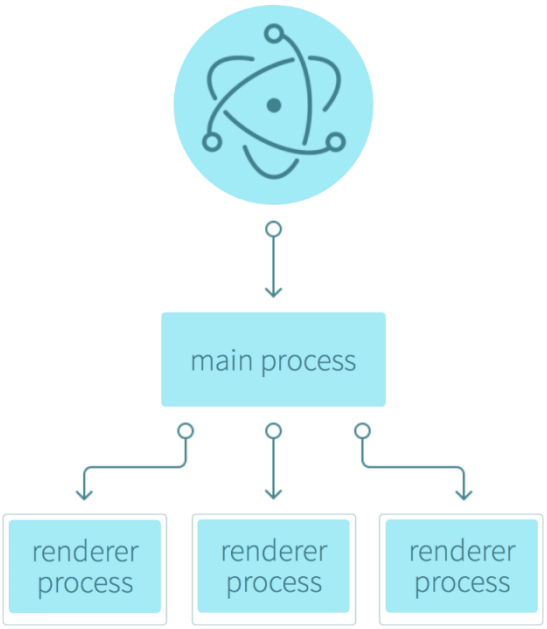
5.Glosario

A lo largo del presente documento se hace referencia a varios términos técnicos. A continuación se presenta un glosario para facilitar la lectura del mismo:

- ACK: Acknowledgement. Mensaje enviado por el receptor confirmando la recepción del mensaje.
- AND: Alpha Numeric Display. Son las pantallas utilizadas para mostrar información.
- DCL: Dispositivo que permite conectar periféricos al GMD.
- GMD: Gabinete de Manejo de Datos. Es la computadora central que contiene un buque.
- GUI: Graphical User Interface. Hace referencia a la interfaz gráfica de usuario.
- LPD: Es la pantalla donde se representa información en el buque, tanto video crudo RADAR como información digital o sintética, y está conformado por un tubo de rayos catódicos.
- TDC: Consola específica donde el usuario interactúa con el GMD.
- PC1 o “main”: Computadora principal, conectada a la FPGA directamente.
- PC2 o “pc2”: Computadora secundaria, conectada indirectamente a la FPGA, mediante la PC1.

6. Estructura

Electron es una plataforma para desarrollar aplicaciones de escritorio usando tecnologías web (HTML, CSS y JavaScript). Funciona creando dos tipos de procesos, el proceso "main" y el proceso "renderer".



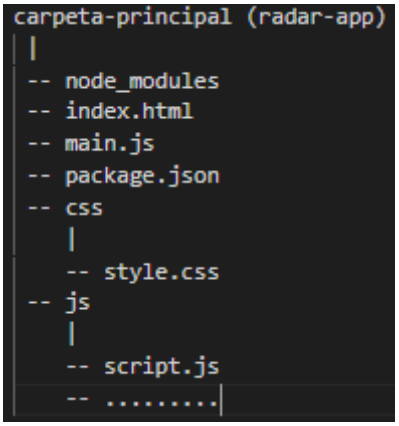
El primero es un proceso de NodeJS. Este es el proceso principal, la aplicación en sí misma. Tiene acceso a varias API de Electron, solo para este proceso que permite interactuar con el sistema operativo y realizar distintas acciones o efectos.

El segundo (renderer) es un proceso de Chromium, pero con una diferencia, este Chromium tiene un NodeJS incorporado y acceso a todos sus módulos y los que se quieran instalar. Por lo que desde nuestro renderer se pueden usar recursos que requieren acceso al sistema operativo, como la lectura y escritura de archivos, listado de directorios, lanzamiento de notificaciones, etc.

Además de acceder a los módulos de Node.js y npm, Electron nos da acceso a algunas APIs al igual que hace con el proceso main.

6. 1. Estructura general del framework Electron.JS

Esta definición de dos procesos, afecta a la estructura de archivos, como se presenta a continuación:



Al generar el proyecto, en la carpeta principal se crea la carpeta “node_modules”, donde se almacenan todos los módulos de NodeJS necesarios. Se crea también el archivo “main.js”, que incluye el código básico fundamental para crear un proceso de renderizado (renderer).

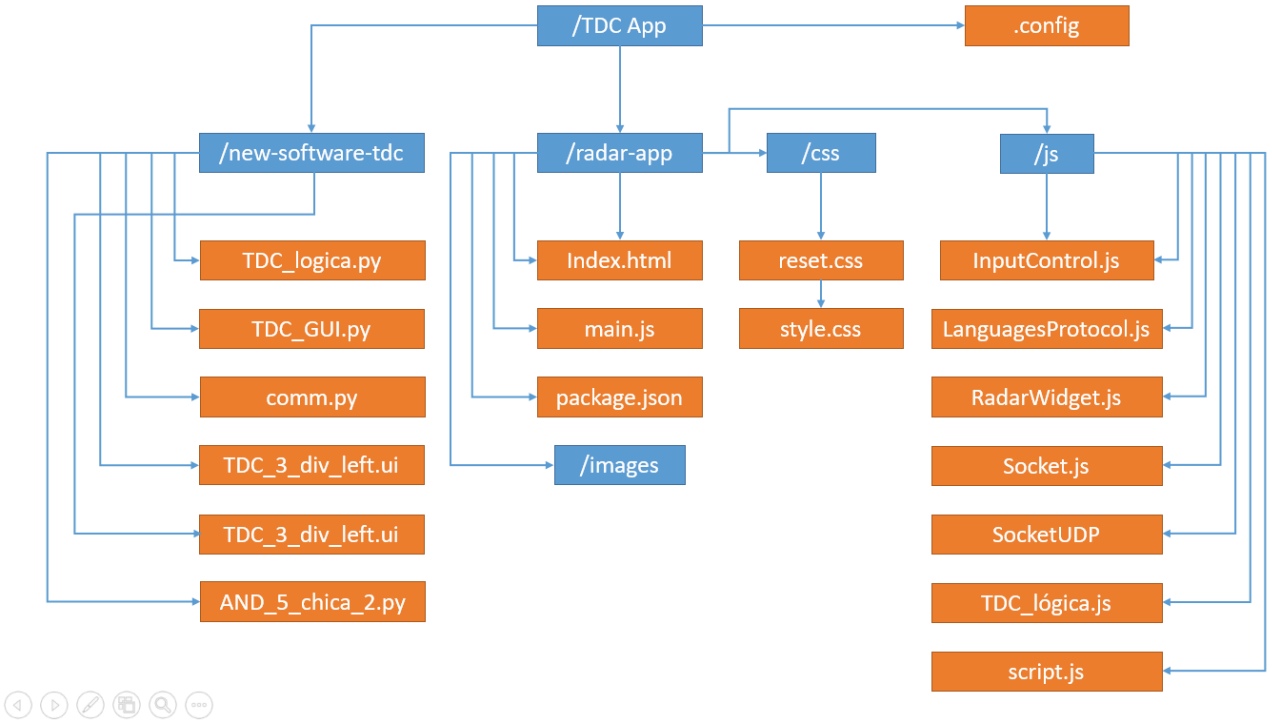
Se genera, además, el archivo “package.json”, que incluye los parámetros de configuración del proyecto. Esto se importa durante la descarga del código fuente, y es necesario para la instalación, dado que la instrucción inspecciona este archivo buscando las dependencias necesarias.

IPC: Inter Process Communication

En ocasiones, es de interés comunicar el proceso principal (main process) con los de renderizado (renderer). Para esto se emplean las herramientas de IPC (comunicación entre procesos), provistos por Electron.

6. 2. Estructura del proyecto

El proyecto en su totalidad, está constituido por dos carpetas, donde una corresponde a la lógica de los controles generales, escrito en Python, y la segunda al framework Electron:



La carpeta que contiene los controles y botoneras, aplicación escrita en Python, tiene de nombre "new-software-tdc". La carpeta que contiene el código fuente vinculado al framework Electron.JS, se nombró "radar-app". Para cada uno, se explica brevemente el contenido de cada archivo:

TDC App:

- .config: Archivo de configuración con la definición de los parámetros básicos de la aplicación. Este archivo permite actualizar la aplicación, sin necesidad de modificar estos parámetros como variables en el código.
- new-software-tdc:
    - TDC_logica.py: Archivo raíz, donde se inicia el socket de comunicación con la aplicación en Electron.JS y se abre la interfaz gráfica.
    - TDC_GUI.py: Contiene el código fuente de la interfaz.
    - comm.py: Define la clase encargada de instanciar el socket.

- TDC_3_div_left.ui: Compone los componentes gráficos de la interfaz, para la PC principal.
- TDC_3_div_right.ui: Compone los componentes gráficos de la interfaz, para la PC secundaria.
- AND_5_chica_2.py: Contiene la lógica del componente que emula la AND.

- radar-app:
  - index.html: Archivo raíz, en HTML, desde el que se describe la estructura de la interfaz.
  - main.js: Incluye el código básico fundamental para crear un proceso de renderizado (renderer).
  - package.json: Archivo de configuración requerido por Electron.JS para la ejecución y el empaquetamiento de la aplicación.
  - images: Directorio con todas las imágenes necesarias.
  - css:
    - reset.css: Reescribe los estilos por defecto definidos por el navegador.
    - style.css: Define todos los estilos de cada objeto en el DOM[11].
  - js:

[11] https://www.w3schools.com/js/js_htmldom.asp

- InputControl.js: Declara la clase para instanciar elementos que permitan ingresar datos en la aplicación. Se emplean para modificar los marcadores de "azimut" y "distancia". Ver figura 13.
- LanguageProtocol.js: Clase que emplea los recursos de alto nivel de Javascript, para procesar las instrucciones que se envían desde Python a Electron a través del socket.
- RadarWidget.js: Clase que crea el widget y elementos necesarios para la interfaz del radar.
- Socket.js: Clase que permite la comunicación entre Python y Electron.JS, y además permite la comunicación entre PCs.
- SocketUDP.js: Clase que comunica la PC principal o "main" y la FPGA.
- TDC_logica.js: Clase que procesa los datos enviados por la FPGA, en forma de cadenas de bits, y devuelve un arreglo de cadenas más sencillas de interpretar.
- script.js: Clase que concentra la lógica principal de la aplicación.

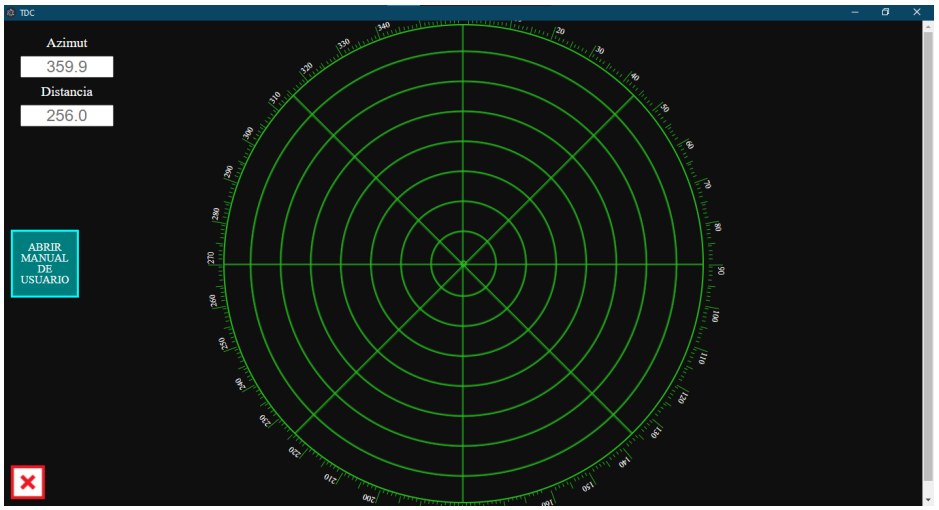


FIGURA (13). PANTALLA DE RADAR, EN LA ESQUINA SUPERIOR IZQUIERDA SE VEN LOS MARCADORES DE AZIMUT Y DISTANCIA LOS CUALES PUEDEN SER MODIFICADOS POR EL OPERADOR.

7. Dependencias

A continuación se listan las dependencias de cada componente. Para la aplicación corriendo sobre Python:

- “pynput”, para los eventos globales del teclado (ANDs).
- “PyQt5”, para la interfaz de botones.
- “python-socketio”, comunicación por sockets con Electron.
- “numpy”, para algunas operaciones matemáticas.

Para la aplicación corriendo sobre Electron:

- “socket-io”, comunicación con Python y entre PCs.
- “socket-io-client”, lo mismo pero con cliente incluido.
- “express”, soporte de socket-io.
- “math”, cálculos.
- “npm” y “electron”, que hacen a la plataforma.

Los métodos de instalación y configuración se detallan en su correspondiente manual.

8. Descripción de archivos

En las siguientes páginas se explicará en detalles las clases y funciones previamente mencionadas, en relación a los archivos que las contemplan. Algunas se mencionan de forma superficial por ser básicas.

8. 0. ".config"

Este archivo contiene algunos parámetros de configuración, según el siguiente formato:

{ROLE}|{DEBUG}|{IP-TDC}|{PORT-TDC}|{IP-FPGA}|{PORT-FPGA}|{MAIN-PC-IP}|{PC-PORT}|{PYTHON-PORT}|{MANUAL-FILEPATH}

Siendo:

- ROLE: El rol del equipo, "main" o "pc2" según corresponda.
- DEBUG: Constante booleana para determinar si la aplicación corre en modo de depuración.
- IP-TDC: IP del dispositivo Ethernet configurado estáticamente, de la PC que se conecta a la FPGA. Solo es relevante para la PC "main".
- PORT-TDC: Número de puerto del dispositivo Ethernet configurado estáticamente, de la PC que se conecta a la FPGA. Solo es relevante para "main".
- IP-FPGA: IP de la FPGA. Solo es relevante para "main".
- PORT-FPGA: Número de puerto de la FPGA. Solo es relevante para "main".
- MAIN-PC-IP: IP de la PC con rol "main". Solo es relevante para la pc2.
- PC-PORT: Número de puerto de la PC con rol "main". Solo es relevante para la pc2.
- PYTHON-PORT: Número de puerto del socket que comunica Python con la aplicación sobre Electron.JS.
- MANUAL-FILEPATH: Dirección del manual de uso de la aplicación, en el equipo en el que corre.

Los siguientes son ejemplos de cómo se compondría la cadena completa, para cada caso según el rol:

- Si la computadora que se está configurando es la computadora principal, completar con el siguiente texto:
  main|false|172.16.0.100|8001|172.16.0.99|9000|192.168.1.10|3000|3000|C:/User/Desktop/manual.pdf
- Por otro lado, si la computadora es la PC secundaria, completar con:
  pc2|false|172.16.0.100|8001|172.16.0.99|9000|localhost|3000|3000|C:/User/Desktop/manual.pdf

8. 1. TDC_logica.py

Este archivo es desde donde se ejecuta la aplicación en Python, y cumple algunas funciones muy concretas:

- Abre el archivo de configuración para copiar el rol que se le asignó al equipo.
- Inicia el socket de comunicación con la aplicación en Electron.JS.
- Inicia la interfaz.

Luego de esto, se queda esperando el cierre de la interfaz para terminar su ejecución.

8. 2. TDC_GUI.py

Esta clase carga la interfaz desde su correspondiente archivo “ui”. Para la computadora principal, se abre el panel de controles lateral izquierdo, y para la computadora secundaria, el panel lateral derecho.

Luego se define la lógica disparada para cada control cuando es presionado o modificado, que en general consiste en generar un dato en forma de cadena y enviarlo a la aplicación en Electron.JS a través del socket.

8. 3. AND.py

Corresponde a la interfaz y a la lógica de la componente AND (Alpha Numeric Display). Este componente no se modificó respecto a lo recibido al iniciar las actividades.

8. 4. comm.py

Este archivo contiene la clase instanciada para configurar e iniciar el socket de comunicación con la aplicación en Electron.JS.

Para la comunicación con sockets, se empleó el módulo “socketio”. Este asegura una comunicación por sockets, bidireccional y de baja latencia. Además, le da robustez a la comunicación, dado que emplea una estructura propia para el websocket.

El motivo principal por el que fue elegido, es que está pensado para comunicar distintos lenguajes, y tiene una versión depurada para JavaScript y para Python. Esto facilitó mucho su implementación.

En la implementación de esta clase para Python, solo se desarrolló para que trabaje como servidor, dado que la aplicación en Electron.JS siempre haría de cliente.

Además, dado que se requería un mayor flujo de datos para distintos propósitos, se evaluó el uso de namespaces. socketio implementa el concepto de “namespace” como un canal de comunicación que permite dividir la lógica de la aplicación dentro de la misma conexión, lo cual se puede ver como un tipo de multiplexado. Al emplear este recurso, se podría seccionar la información sin necesidad de añadir manualmente “encabezados” que identifiquen el remitente del mensaje.

Respecto al módulo socketio, es importante que tanto el módulo servidor como el cliente, y el de python-socketio sean compatibles, según la siguiente tabla:

JavaScript Socket.IO version	Socket.IO protocol revision	Engine.IO protocol revision	python-socketio version	python-engineio version
0.9.x	1, 2	1, 2	Not supported	Not supported
1.x and 2.x	3, 4	3	4.x	3.x
3.x and 4.x	5	4	5.x	4.x

Compatibilidad socketio JavaScript - Python[12]

8. 5. main.js

Cuando el ejecutable de Electron es corrido, este comienza abriendo un archivo desde donde se definen los procesos de renderizado. Este archivo es main.js. Esencialmente, define la/s ventana/s (en nuestro caso una sola), con sus respectivas propiedades. Además, define los eventos de cierre de la ventana y la aplicación.

8. 6. index.html

Este es el archivo HTML raíz de la aplicación en Electron.JS. Este tiene definidos los contenedores para las estructuras que se generarán automáticamente por JavaScript, como los puntos, líneas, círculos, etc.

8. 7. package.json

Contiene los parámetros de la aplicación, como son, nombre, versión, descripción, dependencias, etc. Estos son necesarios para la ejecución y el empaquetamiento de la aplicación.

8. 8. reset.css y style.css

Ambos archivos componen los estilos gráficos y visuales de todos los componentes de la aplicación, existentes en el momento de su ejecución, o añadidos después.

8. 9. InputControl.js

Declara la clase para instancia elementos que permitan ingresar datos en la aplicación. Se emplean para modificar los marcadores de “azimut” y “distancia”.

[12] https://python-socketio.readthedocs.io/en/latest/intro.html

8. 10. LanguagesProtocol.js

Clase que emplea los recursos de alto nivel de Javascript, para procesar las instrucciones que se envían desde Python a Electron a través del socket.

8. 11. RadarWidget.js

En esta clase se definen los elementos gráficos y eventos del radar. En este archivo se definen los callbacks para clicks, teclas, marcadores de azimut y distancia, indicadores, íconos, etc.

8. 12. Socket.js

Clase análoga a comm.py, para la componente escrita en JavaScript. Para este caso era necesario que el módulo pudiera instanciar sockets que funcionaran tanto como servidor, como cliente. Este también emplea namespaces. Como los módulos que trabajan como servidor y como cliente son módulos separados de socketio (socketio y socketio-client), es preciso que las versiones de cada uno sean compatibles, según la siguiente tabla:

JavaScript Socket.IO version	Socket.IO protocol revision	Engine.IO protocol revision	python-socketio version	python-engineio version
0.9.x	1, 2	1, 2	Not supported	Not supported
1.x and 2.x	3, 4	3	4.x	3.x
3.x and 4.x	5	4	5.x	4.x

Compatibilidad socketio JavaScript - Python[13]

8. 13. SocketUDP.js

Este módulo tiene una función muy simple, la cual es concretar la comunicación con la FPGA, mediante un socket UDP que hace de cliente. Aunque por diseño solo tenía esta función, se implementó para trabajar como servidor o cliente según la necesidad.

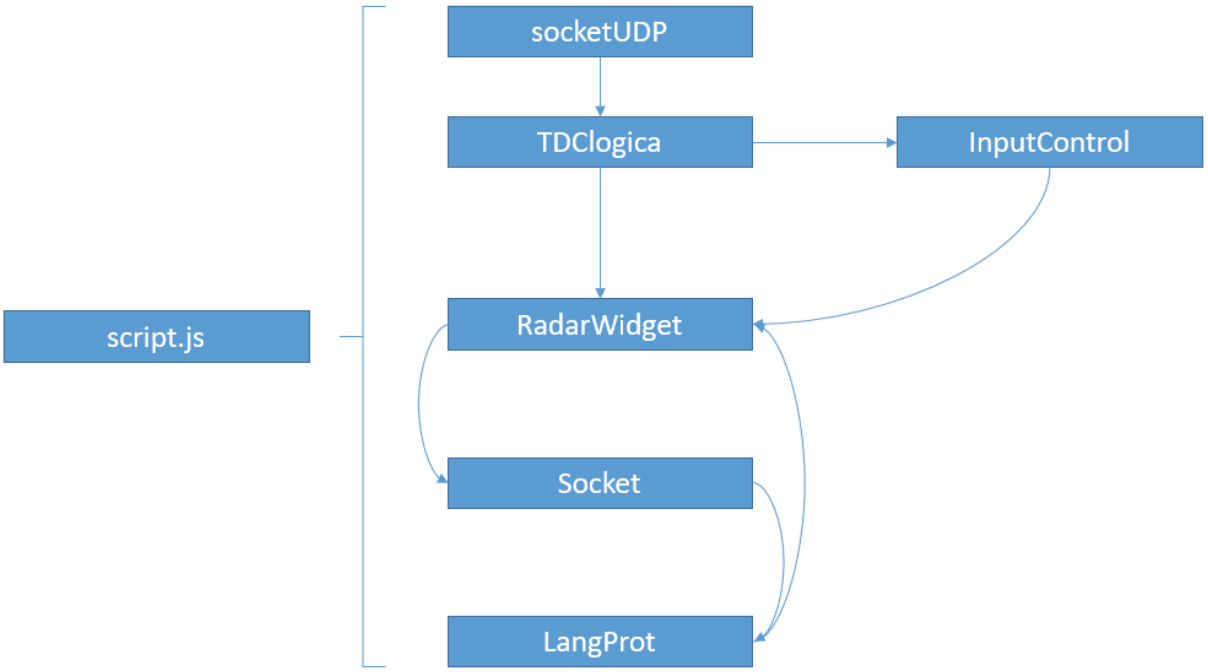
[13] https://python-socketio.readthedocs.io/en/latest/intro.html

8. 14. TDC_logica.js

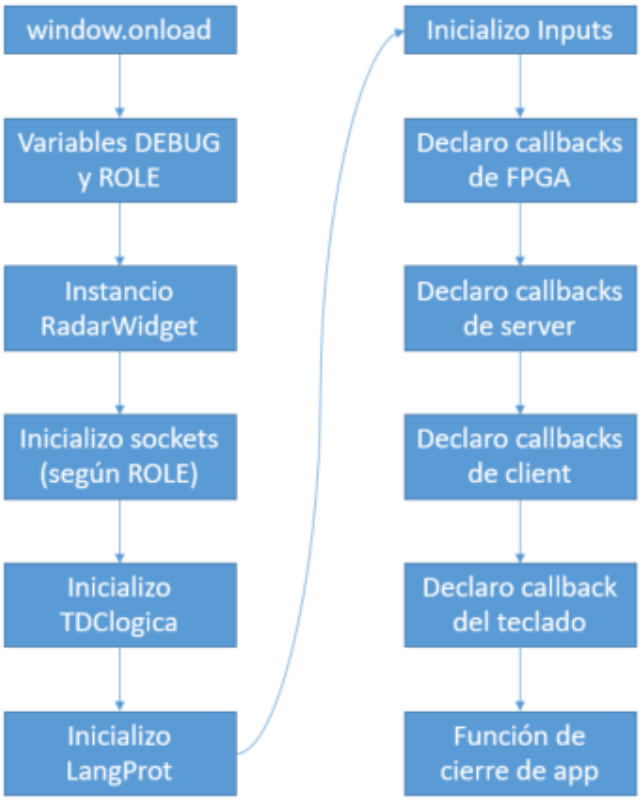
Esta clase es la encargada de decodificar el mensaje recibido desde la FPGA. Esta última envía una secuencia de bits concatenados, que esta clase descompone y transforma en una lista de cadenas, las cuales devuelve para procesar en el archivo script.js. Este tratamiento, aunque lleva tiempo, permite implementar las soluciones para cada caso con mayor facilidad. Los métodos empleados son los mismos que los detallados en el documento del Ing. Manuel Fernandez, en la sección "Métodos complejos detallados".

8. 15. script.js

La ejecución del software parte del archivo "script.js", invocado por index.html al abrir el navegador integrado. En este archivo es donde se vinculan cada una de las clases escritas en JavaScript, como parte del proyecto que corre sobre la plataforma Electron.JS. Estas se relacionan de la siguiente manera:



La secuencia de instrucciones se describe a continuación:



Cuando el navegador termina de cargar todos los componentes del HTML, se dispara el evento "window.onload". Dentro de este se ejecutan todas las instrucciones de inicialización.

Primero, el script utiliza la IPC para solicitarle información al "main.js" del framework Electron. Este le pasa los parámetros que recibió por consola en su invocación, con las variables DEBUG y ROLE.

Luego, se instancia la clase RadarWidget, que contiene la visualización del radar.

Se inicializan los sockets, según el ROLE actual. El siguiente fragmento de código representa este proceso:

```
if (ROLE == "main") {
    var socketIO = new Socket(
        "0.0.0.0", // Con esto el socket buscara el puerto abierto para todas las IPs
        PYTHON_PORT,
        SERVER,
        ["AND1", "AND2", "DCL", "PC2", "FPGA"],
        on_read_callback_server
    );
    var socketPython = socketIO;

    var socket_FPGA_UDP = new SocketUDP(
        SERVER,
        IP_TDC,
        PORT_TDC,
        IP_FPGA,
        PORT_FPGA,
        on_ready_callback_FPGA_UDP,
        on_read_callback_FPGA_UDP
    );
    // Socket UDP con la FPGA
} else if (ROLE == "pc2") {
    var socketIO = new Socket(
        MAIN_PC_IP,
        PYTHON_PORT,
        CLIENT,
        ["PC2", "FPGA"],
        on_read_callback_client
    );

    var socketPython = new Socket(
        "localhost",
        PYTHON_PORT,
        SERVER,
        ["AND1", "AND2", "DCL"],
        on_read_callback_server
    );

    var socket_FPGA_UDP = socketIO;
```

Para ROLE igual “main”, se crea un socket como servidor TCP, usando el módulo socketio. La única forma en la que se pueden comunicar dos computadoras con este módulo, es declarando la IP de este servidor como “0.0.0.0”. De esta forma, le asignamos un puerto, y el servidor estará escuchando ese puerto para todas las direcciones IPs disponibles. Por este motivo, empleamos el mismo puerto para la comunicación con la aplicación en Python. Los namespaces nos permitirán diferenciar los emisores y remitentes del mensaje:

- “DCL” se usa para la comunicación de los mensajes del panel a la FPGA.
- “AND1” y “AND2” para los mensajes con la AND.
- “PC2” es el namespace con el que “pc2” envía mensajes a “main”, y viceversa.
- “FPGA” es el namespace con el que “main” envía lo recibido desde la FPGA, a “pc2”.

Luego, se crea el socket UDP para la comunicación con la FPGA.

Para ROLE igual a “pc2”, se crea el socket cliente TCP para conectarse a “main”, con los únicos dos namespaces necesarios PC2 y FPGA. Luego, se crea el socket servidor para la comunicación con Python, y se asigna la variable de comunicación UDP al socket TCP, dado que las operaciones de comunicación son las mismas y esa simple asignación ya basta para la correcta recepción de datos de la FPGA desde “main”.

Siguiendo la inicialización global del software, instancio TDClogica.

1. Instancio LangProt.
2. Instanció e inicializó “InputControl”.
3. Declaró los callbacks de la FPGA.
4. Declaró el callback del servidor.
5. Declaró el callback del cliente.
6. Declaró un callback adicional para responder al teclado desde el radar, necesario para algunas funciones de la aplicación en Python.
7. Declaró una función para el correcto cierre de la aplicación.

Los siguientes son algunos aspectos relevantes respecto al flujo de ejecución del programa.

La función “on_ready_callback_FPGA_UDP” incorpora la ejecución del módulo de Python. Al ejecutarse, evalúa el sistema operativo sobre el cual está corriendo, y ejecuta la aplicación en Python. Además, define los callbacks para cualquier salida, error o fin de ejecución del software.

En la función “on_read_callback_FPGA_UDP” se reciben los datos enviados por la FPGA. Empleando la clase TDClogica se realiza un parseo de estos datos, y según corresponda, se ejecutan algunas funciones que le retornen a la FPGA la información necesaria.

```
if (msgDecode[0] == "AND1" && ROLE == "main") { // envio la informacion de la AND1 a la app de Python

    socketPython.send_message(msgDecode[1], "AND1");
    msgSend = msgDecode[2];

} else if (msgDecode[0] == "AND2") {

    if (ROLE == "main") {
        msgSend = msgDecode[2];
    } else {
        socketPython.send_message(msgDecode[1], "AND2");
    }

} else if (msgDecode[0] == "LPD" && msgDecode.length > 1) { // Envio hacia la clase radar la informacion de la LPD

    radar.borrarPuntos();
    if(msgDecode[1].length > 0) { // msg AB1
        radar.set_origen_x_y(msgDecode[1]);
    }
    if(msgDecode[2].length > 0) { // msg AB2
        radar.graficar_markers(msgDecode[2]);
    }
    if(msgDecode[3].length > 0) { // msg AB3
        radar.graficar_cursores(msgDecode[3]);
    }
```

Dentro de esta misma función, también se modifican los registros enviados a la FPGA, para que agrupen tantos los mensajes correspondientes a la PC principal (“main”), como a la PC complementaria (“pc2”).

```
msgSend = msgDecode[1];

if (ROLE == "main" && last_msg["DCL"] != undefined) {
    let pc2_msg = last_msg["DCL"][0];
    // Tomo solamente el mensaje decodificado

    msgSend[7] = (pc2_msg[7] & 254) | (msgSend[7] & 3);
    // Superpongo los primeros 5 bits del registro 7 de PC2
    // (corresponde a los botones de centrado). Los bits 1 y 2 no
    // los modifico, mientras que el bit 3 corresponde al data request
    // por lo que mantengo el ultimo estado
    msgSend[10] = pc2_msg[10];
    // Superpongo los valores del elemento 10 que corresponden a la MIK
    // derecha (AND2)
    msgSend[13] = pc2_msg[13];
    // Superpongo los valores del elemento 13 que corresponden al QEK
    // derecho
    msgSend[16] = pc2_msg[16];
    // Superpongo los valores del elemento 16 que corresponde al ICM
    // y Overlay derecho
    msgSend[18] = msgSend[18] & pc2_msg[18];
    msgSend[19] = msgSend[19] & pc2_msg[19];
    // Los elemento 18 y 19 corresponde a la HW.
    // Mantengo el estado de la handwheel segun el
    // ultimo que lo modifico
    msgSend[24] = pc2_msg[24];
    msgSend[25] = pc2_msg[25];
    // Corrijo las coordenadas actuales del rolling derecho con las
    // que recibo de la PC2

    last_msg_DCL = pc2_msg;
}
```

Para entender cuáles son los elementos relevantes del arreglo recibido desde “pc2”, referirse a la función “return_estado_CONC” en el archivo “TDC_GUI.py”, que describe la asignación de cada palabra como variables, y en donde se puede ver cómo está compuesto cada grupo de 24 bits.

Conociendo la relación entre cada control y su inferencia en el arreglo enviado a la FPGA, podemos determinar cuáles son los elementos vinculados a los controles de “pc2”, y reemplazarlos en su correspondiente posición, en el arreglo que enviaremos. Estos serán:

- Los primeros 5 bits del elemento 7, correspondiente a los botones de centrado.
- El bit 3 del elemento 7, que corresponde al data request derecho.
- El elemento 13, sus 8 bits, que corresponde al QEK-R.
- El elemento 16, que corresponde al Overlay-R y al ICM-R.

- Los elementos 18 y 19, que corresponden al estado de la handwheel. En este caso, se hace una operación “AND”, dado que por defecto su valor es 255. Si se modifica en alguna de las dos PCs, el valor enviado es distinto a este valor, y prevalece por encima del 255 de la computadora que no fue modificada.
- Los elementos 24 y 25, que son los que almacenan la posición de la OBM derecha.

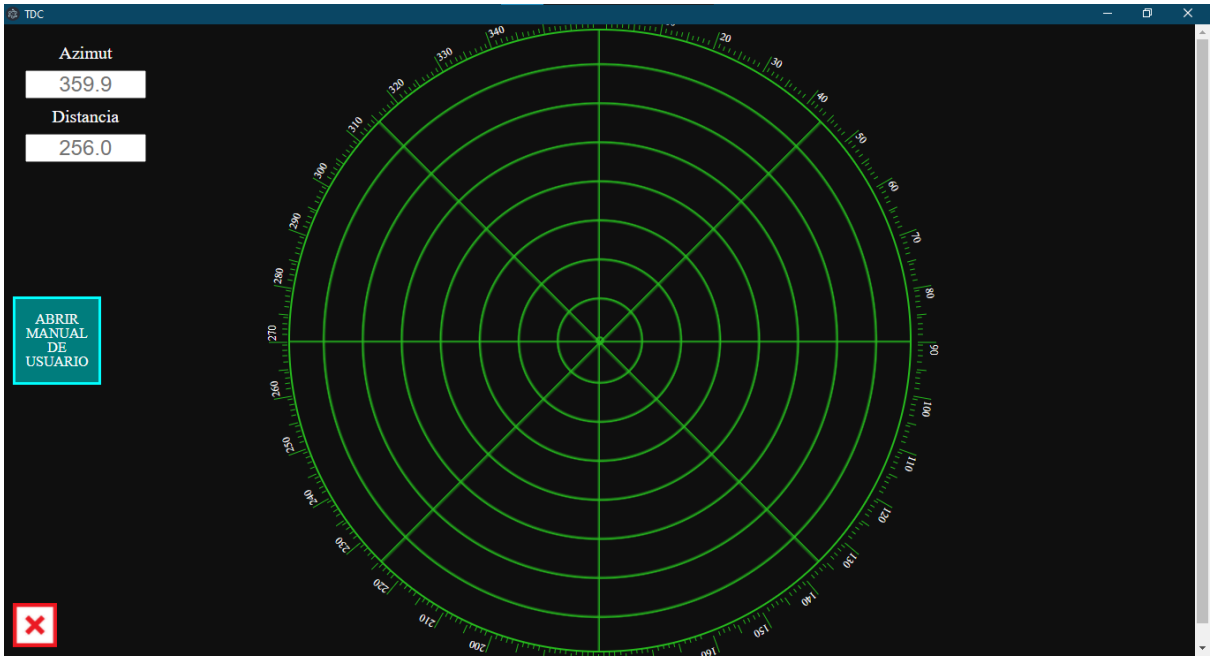
Para el caso de los botones compartidos, cuando alguno de estos es presionado, se envía un mensaje a la computadora opuesta para que presione este botón forzosamente, y por consecuencia, actualice los registros. Esta es la forma más simple y a su vez funcional que se encontró, para hacer coincidir estos controles porque además de alterar los registros, actualiza la condición de los mismos en la interfaz. A estos mensajes, se le antecede el prefijo “panel|”, al índice del botón presionado. Luego, en el callback correspondiente al servidor y al cliente, se recibe el dato, y se pulsa el botón recibido.

Otra función del programa a tener en cuenta, es un pequeño fragmento que evalúa la calidad de la conexión, cada 500ms. Esto se hace enviando un mensaje (“CON-ACK”) periódicamente desde la computadora “main”, que al ser recibido por “pc2”, es reenviado a la primera. Al recibir “main” la respuesta de este mensaje, entiende que la comunicación es estable, y en ambos equipos se actualiza el indicador en pantalla.

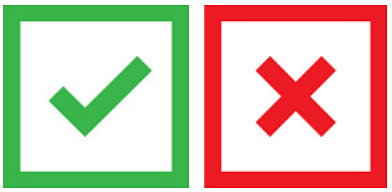
En ocasiones, este indicador alterna su estado recurrentemente cada cierto tiempo, como una pulsación. Esto se da porque una de las dos aplicaciones fue minimizada. Al minimizar, el sistema operativo frena algunos procesos, y los mensajes dejan de enviarse o recibirse automáticamente, produciendo este comportamiento.

```
// Funciones de verificacion de conexion
setInterval(function() {
    /*
    Verifica la correcta conexion de las dos computadoras periodicamente.
    */
    if (!check_state) {
        if (!checker_element.classList.contains('false')) {
            checker_element.classList.add('false');
            checker_element.classList.remove('true');
        }
    } else {
        if (!checker_element.classList.contains('true')) {
            checker_element.classList.add('true');
            checker_element.classList.remove('false');
        }
    }

    check_state = false;

    if (ROLE == "main") {
        socketIO.send_message("CON-ACK", "PC2");
    }
}, 500);
```

La siguiente es una captura de la aplicación, donde se puede ver este indicador en la zona inferior izquierda:



De las siguientes dos imágenes, cuando la conexión es estable, el indicador muestra el ícono de la izquierda, mientras para el caso contrario presenta el indicador de la derecha.



A continuación, se indican los fragmentos correspondientes a cada función, en el callback del servidor:

```
function on_read_callback_server(msg, namespace) {
    if (namespace == "DCL") {
        TDC.setEstadoCONC(msg);
    } else if (namespace == "PC2") {
        // Recibo desde Python de PC2
        if (msg.split("|")[0] == "panel") {
            socketPython.send_message(msg, "DCL");
        } else if (msg == "CON-ACK") {
            // Conexion verificada
            check_state = true;
        }
    } else if (namespace == "/") {
        if (msg.split("|")[0] == "panel") {
            socketIO.send_message(msg, "PC2");
        } else {
            if (msg == "role") {
                socketPython.send_message(ROLE);
            }
            else {
                lp.take_action(msg);
            }
        }
        /*
        Los mensajes de un boton presionado llegan con el
        prefijo "panel" (haciendo referencia al panel de
        botones)
        */
    }

    if (namespace == "FPGA") {
        // Mensajes recibidos desde la PC2, para FPGA
        last_msg = msg;
    }
}
// Funciones server
```

Cada condicional de esta función representa una utilidad relevante. Tomar en cuenta que este callback es el mismo para el servidor de “main”, y el servidor que se conecta con la aplicación en Python. Por lo que habrá condicionales que respondan a callbacks distintos, dentro de la misma función.

- En rojo: Configura los registros del concentrator antes de enviarlo a la FPGA.
- En azul: Estando en “main”, recibe el botón presionado, y se lo envía a la interfaz en Python para que modifique su estado.
- En verde: Estando en “main”, recibe el mensaje “CON-ACK” de vuelta, y confirma el estado de la conexión.
- En violeta: Tanto en “main” como en “pc2”, es el callback del servidor para la comunicación en Python. Recibe el botón presionado, y se envía a la PC opuesta.
- En blanco: Función invocada cuando se recibe un mensaje de Python, y se aplica al RadarWidget. Además, se agrega un condicional dedicado a pasarle a la aplicación en Python, el ROLE actual. Esto se hace al principio del programa, al iniciar la ejecución.
- En amarillo: Estando en “main”, cuando se recibe desde “pc2” el arreglo a enviar a la FPGA, se almacena en una variable auxiliar para después tomar los registros necesarios, como se explicó previamente.

Ahora, se indican los fragmentos correspondientes a cada función, en el callback del cliente:

```
function on_read_callback_client(msg, namespace) {
    if (namespace == "FPGA") {
        on_read_callback_FPGA_UDP(msg);
        // Recibo msgDecode desde PC1
    } else if (namespace == "PC2") {
        // Recibo desde Python de PC2
        if (msg.split("|")[0] == "panel") {
            socketPython.send_message(msg, "DCL");
        } else if (msg == "CON-ACK") {
            // Verificacion de conexion, envio devuelta el ACK
            socketIO.send_message("CON-ACK", "PC2");
            check_state = true;
        }
    }
    // Convoco la funcion con el mismo contenido que le llego
    // a la PC main desde la FPGA
}
// Funciones cliente
```

- En rojo: Estando en “pc2”, cuando desde “main” se retransmite lo recibido desde la FPGA, se recibe en este callback para procesarlo con la misma función.
- En azul: Estando en “pc2”, recibe el botón presionado, y se lo envía a la interfaz en Python para que modifique su estado.
- En verde: Cuando recibe el mensaje de verificación “CON-ACK”, lo reenvía, y confirma la estabilidad de la conexión.

Con respecto al callback del teclado, su uso es muy concreto:

```
on_key = function(event, $this) {
    let keycode = [32, 17, 16, 112, 113, 114, 115, 116, 117, 118, 119, 67, 68, 88, 87];

    let keychar = [
        "space",
        "ctrl_l",
        "shift_l",
        "f1",
        "f2",
        "f3",
        "f4",
        "f5",
        "f6",
        "f7",
        "f8",
        "c",
        "d",
        "x",
        "w"
    ];

    let tecla;

    if (event.which >= 49 && event.which <= 56) { // Numeros del 1 al 8
        if (!azimutInputControl.focused() && !distanciaInputControl.focused()) {
            // Si no se esta escribiendo en los inputs, tomo los numeros
            // como atajos de teclado para los rangos de escala
            tecla = (event.which - 48).toString();
        }
    } else {
        tecla = keychar[keycode.indexOf(event.which)];
    }

    if (tecla) socketPython.send_message("tecla_apretada_mik|" + tecla);
}
```

```
document.onkeydown = function(event) {
    // Funcion invocada cuando una tecla es presionada
    if (event.which >= 112 && event.which <= 123) // F1 - F12
        event.preventDefault();
    // Para evitar que Electron ejecute los atajos de teclado por defecto

    on_key(event, $this);
}
```

Esta función fue agregada únicamente para escuchar a ciertas teclas y combinaciones necesarias del lado de Python. Por eso, solo te toman algunas teclas, y se las envía a la aplicación en Python, al ser presionadas. Las teclas en escucha son:

- Espacio

- Control izquierda
- Shift izquierda
- Algunas teclas “función”.
- Las letras “c”, “d”, “x”, y “w”.
- Los números, del 1 al 8.

La misma sugiere una complicación menor, que es cuando se presionan algunos de los números. Estos tienen doble función, pueden ser usados como atajos de teclado para los rangos de escala, sino además para escribir un valor en los campos de entrada de azimut y distancia. Por lo cual, se desarrolló la excepción, para que no actúen como atajos cuando se está escribiendo en estos campos.

Para algunas funciones se requiere también, que se detecta cuando se suelta una tecla. Usando la misma lógica, se agregó el evento y su callback correspondientes:

```
up_key = function(event, $this) {
    let keycode = [17];
    let keychar = [
        "ctrl_l"
    ];
    let tecla = keychar[keycode.indexOf(event.which)];

    if (tecla) socketPython.send_message("tecla_liberada_mik|" + tecla);
}
```

```
document.onkeyup = function(event) {
    // Funcion invocada cuando la tecla presionada, se suelta
    if (event.which >= 112 && event.which <= 123) // F1 - F12
        event.preventDefault();

    up_key(event, $this);
}
```

Apertura y cierre de la aplicación

Cuando se ejecuta Electron, y este arranca los procesos de main.js, se genera una ventana empleando “index.html” como plantilla. Dentro de index.html, se referencia “script.js”, que se ejecuta automáticamente. Al terminar de cargar el contenido HTML de la plantilla, se dispara el evento

"window.onload". En este evento, se inicia la comunicación UDP. Cuando esta comunicación se da exitosamente, se dispara el evento "on_ready_callback_FPGA_UDP".

En este evento, es cuando se ejecuta la aplicación en Python. Lo primero que se hace, es verificar el sistema sobre el cual se está ejecutando la aplicación, dado que los procesos necesarios son diferentes:

La constante "SYSTEM" se asigna automáticamente al inicio del script, desde la variable process del sistema.

```
const MAIN_PC_IP = "192.168.1.10";
const PC2_IP = "192.168.1.20";

const PYTHON_PORT = "3000";

const IP_TDC = "192.168.1.5";
const PORT_TDC = 8000;
const IP_FPGA = "192.168.1.11";
const PORT_FPGA = 8001;

const SERVER = 1;
const CLIENT = 0;

const SYSTEM = process.platform;

const checker_element = document.getElementsByClassName("checker")[0];
// CONSTANTES
```

Según el sistema, se ejecuta un script diferente. Como herramienta de ejecución, para Windows se emplea "cmd.exe", y para Linux, "sh".

Para el cierre de la aplicación, se emplea la comunicación IPC mencionada previamente. Cuando se intenta cerrar la aplicación, se dispara el evento "close", en main process, para la ventana correspondiente:

```
mainWindow.on('close', function(event) {

  if (status == 0) {
    if (mainWindow) {
      event.preventDefault();
      mainWindow.webContents.send("app-close");
    }
  }

});
```

Entonces, se envía el mensaje “app-close” al renderer, que lo recibe como se muestra a continuación:

```
ipcRenderer.on("app-close", _ => {
    if (children) {
        if (SYSTEM == "win32") exec("taskkill /IM 'python.exe' /F");
        else if (SYSTEM == "linux") exec("killall python3");
    }
    ipcRenderer.send("closed");
});
```

En este callback, se comprueba nuevamente el sistema, y se ejecuta un comando de consola que cierra forzosamente la aplicación en Python. Finalmente, el renderer envía un mensaje al main, para que cierre la aplicación como intentaba hacer previo a este proceso:

```
ipcMain.on("closed", _ => {
  status = 1;
  mainWindow = null;
  app.quit();
});
```

Además, la aplicación en su totalidad se cierra al cerrar cualquiera de las dos ventanas, tanto la de Electron como la de Python. Esto se hace en el callback de la invocación del software en Python, donde al cerrarse esta última, cierra automáticamente Electron.

Archivos adicionales

Para facilitar la ejecución del software, y que no se requiriera utilizar la consola en ninguno de los sistemas operativos disponibles, se crearon archivos para automatizar su ejecución. En Microsoft

Windows, se emplea el archivo de nombre “TDC(last).bat”. Este está escrito en Batch, compuesto por las siguientes instrucciones:

```
set OLDDIR=%CD%

cd %OLDDIR%\radar-app
node_modules\.bin\electron main.js

chdir /d %OLDDIR% &rem restore current directory
```

En Linux, se usa “TDC(last).sh”, con lo siguiente:

```
$PWD=$(pwd)

"$PWD/radar-app/node_modules/.bin/electron" "$PWD/radar-app/main.js"
```

En ambos, se ubica en el directorio actual, y usando el archivo raíz de la dirección de instalación de Electron, ejecuta el archivo main.js, que es raíz de la interfaz.

9. Técnicas de depuración

Para depurar el funcionamiento de la aplicación, desde el equipo se desarrolló un firmware corriendo sobre una placa STM Discovery STM32F407VG. Esto permitió acelerar el proceso dado que se podían descartar rápidamente problemas menores, sin necesidad de realizar todas las pruebas a bordo del buque.

Otro conflicto que en cierto punto ralentizaba el proceso, era el uso de dos computadoras. Por tal motivo, se buscó una alternativa, y se encontró que por el diseño modular utilizado en la implementación de la aplicación, se podían correr las dos aplicaciones sobre un mismo equipo. Para esto, era necesario utilizar el puerto local (localhost, 127.0.0.1) para comunicar las dos aplicaciones. A su vez, como siempre, cada aplicación se comunicaría con su correspondiente servidor en Python. Por tal motivo, se tuvieron que cambiar los puertos de los sockets, para que no se mezclaran los mensajes de cada una. Se debe configurar ambas aplicaciones para PC-PORT sea el mismo para las dos. Mientras que en una, PYTHON-PORT debe ser diferente de la otra. Un ejemplo para cada una puede ser:

- Si la computadora que se está configurando es la computadora principal, completar con el siguiente texto:
  main|false|172.16.0.100|8001|172.16.0.99|9000|localhost|3000|3000| C:/User/Desktop/manual.pdf
- Por otro lado, si la computadora es la PC secundaria, completar con:
  pc2|false|172.16.0.100|8001|172.16.0.99|9000|localhost|3000|4000|C:/User/Desktop/manual.pdf

Otro conflicto que surgió en cierto punto del desarrollo, fue la necesidad de reemplazar uno de los equipos por una notebook, que tenía mejores especificaciones. La dificultad fue que la notebook tenía un solo puerto Ethernet, y era necesario que trabajara como computadora principal, por lo cual, debía tener dos puertos. Para solucionar esto, se decidió conectar a las dos PCs mediante WiFi. Para esto, se fija la IP de la PC “main”, con la IP del router WiFi. Debido a esto, la configuración de esta PC quedaría según:
main|false|192.168.0.100|8001|172.16.0.99|9000|localhost|3000|3000| C:/User/Desktop/manual.pdf

Siendo 192.168.0.100, la IP de router.

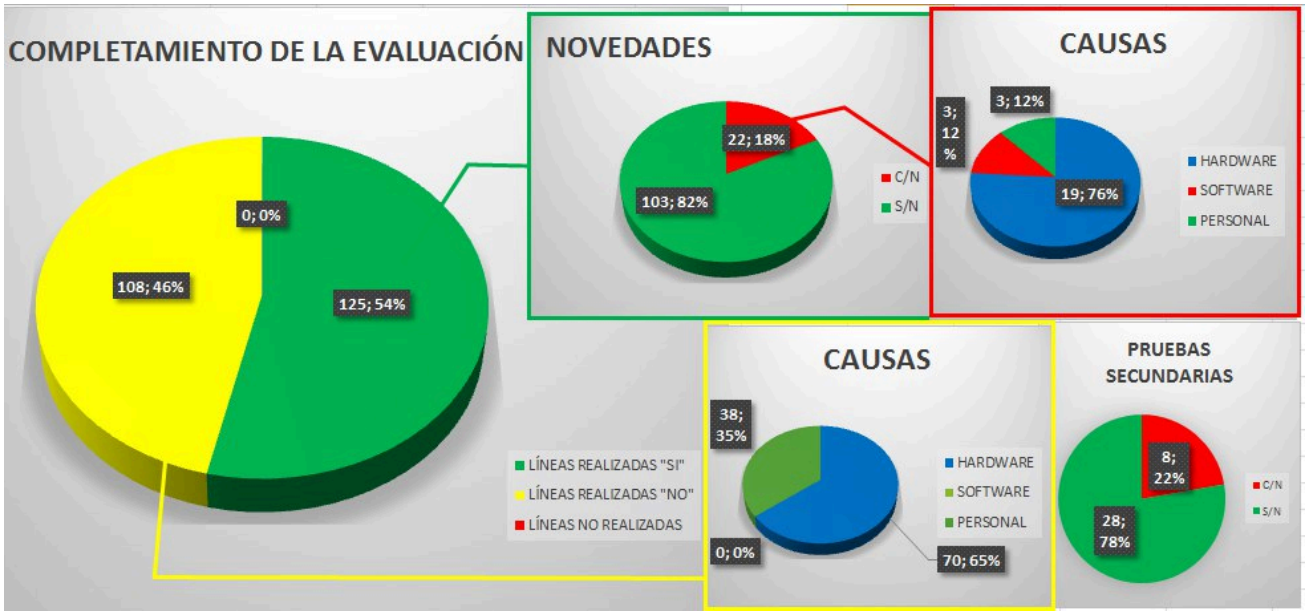
10. Pruebas realizadas

Las primeras pruebas realizadas fueron luego de implementada la interfaz del radar sobre Electron.JS. Como se mencionó, estás pruebas presentaron una ventaja de 10 a 12 veces. Es decir, la aplicación corriendo la interfaz radar en Electron.JS, requería de 10 a 12 veces menos recursos del procesador que la misma aplicación corriendo en Python. Con esta métrica en mente, se procedió a implementar la división de la aplicación, mientras el becario del equipo realizaba la traducción de la comunicación con la FPGA y la decodificación de los mensajes.

Luego de realizar gran parte de esta implementación, se continuaron con algunas pruebas sobre el buque. Los siguientes meses consistieron en prueba y depuración simultáneamente, en busca de avances importantes.

Las pruebas consisten en ejecutar la aplicación, y dar lugar a los operadores para que realicen las distintas operaciones características. En las primeras, lo importante fue evaluar la fluidez de la comunicación con la interfaz radar, los indicadores, cursores y símbolos graficados. Son los mismos operadores los que indican un funcionamiento irregular o mal funcionamiento de los distintos componentes que hacen a la interfaz. A partir de ahí, se intenta evaluar el origen del problema, para poder corregirlo.

Este proceso resultó exitoso después de un periodo de aproximadamente 6 meses de depuración. Meses después, otro equipo se dedicó intensamente a la búsqueda de errores o fallas, resultando en las conclusiones presentadas en el siguiente gráfico.



Del mismo se puede deducir, que se realizaron 125 pruebas. De estas pruebas, 22 resultaron en novedades (hay otras 3 que corresponden a los operadores, pero estás no se toman en cuenta por no ser parte del sistema). De estas 22 novedades, solo 3 responden al software, y se deben a dos motivos:

- Una falla asociada a la representación ausente del símbolo “+” en la AND, que se debe al análisis del paquete de información enviado por el DHC y es previo a la optimización del software.
- Una característica gráfica, donde el botón “sys_alarm” debe cambiar de color en distintas circunstancias, y como no fue tomado en cuenta durante el diseño, generó errores durante las pruebas en dos ocasiones.

Es decir, respecto al desarrollo del software en este contexto, el mismo no presentó falla alguna, denotando una eficiencia de 100%.

Esto se considera muy meritorio, dado que las condiciones de desarrollo no facilitaban la depuración del software.

12. Bibliografía

1. Fernández, M. (2021). *Diseño, desarrollo y verificación de aplicación de software para el envío, recepción y representación de datos por puerto Ethernet* [Proyecto]. UTN FRBB.
2. Lutri, J., & Loidi, M. (2021). *Desarrollo de una interfaz electrónica para posibilitar la comunicación entre una computadora naval y una comercial con requerimientos temporales incompatibles* [Proyecto]. UTN FRBB.
3. Electron.js Documentation. (s.f.). *Electron API documentation*. Recuperado de https://devdocs.io/electron/c
4. Node.js Documentation. (s.f.). *Node.js documentation*. Recuperado de https://www.electronjs.org/docs/latest
5. Socket.IO Documentation. (s.f.). *Socket.IO documentation*. Recuperado de https://socket.io/docs/v4/
6. Python Documentation. (s.f.). *Python 3.9 documentation*. Recuperado de https://docs.python.org/es/3.9/index.html
7. Qt for Python. (s.f.). documentation. Recuperado de https://doc.qt.io/qtforpython-6/.
8. Numba documentación. (s. f.) Numba makes Python code fast. Recuperado de https://numba.pydata.org/
9. JAX Documentation (s.f.). JAX documentation. Recuperado de https://jax.readthedocs.io/en/latest/
10. Chromium documentación. (s.f.). The Chromium Project. Recuperado de https://www.chromium.org/Home/
11. Introducción Electron.js. (s.f.). *Building cross-platform desktop apps with Electron.js*. Disponible en https://www.toptal.com/javascript/electron-cross-platform-desktop-apps-easy