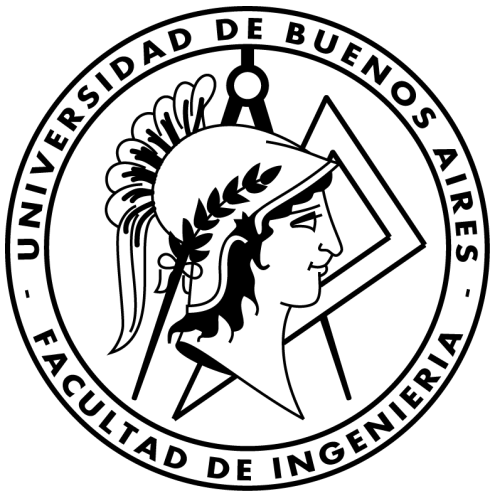


Facultad de Ingeniería

Universidad de Buenos Aires

Tesis de Grado de Ingeniería Electrónica

Diseño y Construcción de una Computadora de Vuelo de Bajo Nivel, con Capacidad de Tolerancia a Fallas a Partir de Redundancias

---

Alumno:

Federico Nuñez Frau (98.211)

fnunezf@fi.uba.ar

Director:

Claudio Pose

cldpose@fi.uba.ar

Co-Director:

Leonardo Garberoglio

COMPLETAR MAIL

COMPLETAR FECHA

Índice

1. Objetivo

El presente trabajo de Tesis tiene por objetivo el diseño y construcción de una computadora de vuelo de bajo nivel, a ser utilizada en un vehíuclo aéreo hexarotor, no tripulado. Como aspecto particular, esta debe contar con la capacidad de tolerar ciertas fallas de hardware que puedan ocurrir en pleno vuelo. Lo que se busca, es que estas fallas no impacten en la misión del vehículo y que puedan ser detectadas lo antes posible para tomar una acción.

En primera medida, se hace un análisis e investigación acerca del estado del arte, para vehículos aéreos no tripulados de caracter comercial, principalmente drones. El objetivo es conocer los mecanismos de seguridad que se implementan en este tipo de vehículos, tanto de hardware como de software. Por otro lado, se busca conocer cuáles son las normas actuales, pertinentes al uso y comercialización de vehículos aéres no tripulados, principalmente drones.

Lo siguiente es el desarrollo de una computadora de vuelo. Esto comprende la definición de los requerimientos de la misma, principalmente de hardware en cuanto a sensores, conectores y funcionalidades deseadas. A partir de estos, se hace una investigación de la variedad de componentes disponibles. Luego, se pasa a una etapa de selección de los componentes a utilizar. Por último, se define un circuito esquemático y se diseña un PCB, el cual será enviado a fabricación.

Para abordar la tolerancia a fallas de hardware de la computadora de vuelo, se plantea utilizar técnicas que involucran la redundancia, tanto de hardware como de software. Para ello, se lleva a cabo una investigación de las técnicas comúnmente utilizadas en el sector aeronáutico para tolerancia de fallas. Finalmente, se define un esquema y una arquitectura a utilizar como mecanismo de tolerancia a fallas.

Para demostrar los resultados, se presentan resultados de pruebas de control de un motor en una arquitectura redundante, sobre la cual se simula la manifestación de distintos tipos de fallas. Se presentan los resultados y la respuesta del sistema.

2. Introducción

COMPLETAR

3. Diseño Tolerante a Fallas de Hardware

3.1. Introducción al Análisis de Tolerancia a Fallas

En los últimos años se ha incrementado mucho la presencia de UAVs en espacio aéreo civil. Debido a esto, se plantea que los UAVs deberían presentar características que permitan un funcionamiento correcto, tolerante a fallas. Como consecuencias posibles, el hecho de volar en espacio aéreo civil puede llegar a causar daño físico a personas, si es que un vehículo presenta una falla y por ejemplo pierde el control. Otra de las posibles consecuencias tiene que ver con los costos que puede ocasionar una falla en una misión relacionada a una actividad laboral. El hecho de tener que repetir la misión puede traer mayores costos para la actividad en cuestión.

El objetivo del diseño tolerante a fallas consiste en mejorar la confianza (*Dependability*) del sistema, apuntando a que este pueda seguir ejecutando su función de manera correcta a pesar de la presencia de una cierta cantidad de fallas [8]. De esta última expresión se puede tomar una definición de lo que es un sistema tolerante a fallas.

Definición 1. ***Sistema Tolerante a Fallas:*** *es aquel donde una falla no implica necesariamente un fracaso en el funcionamiento. Un sistema tolerante a fallas no es aquel donde no ocurren fallas, sino que más bien, se acepta que las fallas pueden ocurrir en el sistema, pero lo que se pretende es que el sistema pueda cumplir con su función de igual manera.*

De manera de introducir la nomenclatura que se encuentra en la bibliografía [8], se definen los siguientes términos:

- Falla (*Fault*): es alguna condición anómala, no esperada.
- Error: ocurre cuando una falla se manifiesta y produce un comportamiento fuera de lo esperado en alguna parte del sistema.
- Fracaso (*Failure*): quiere decir que el sistema no puede cumplir con su función de manera adecuada.

Una de las formas de cuantificar la confianza es a través de la fiabilidad del sistema (*Reliability*). Esta se expresa en la ecuación (1), y se define como la probabilidad de que el sistema pueda cumplir su función de manera correcta en un intervalo de tiempo $[t_0; t]$, dado que en el instante inicial t_0 el sistema podía hacerlo.

$$R(t) = \mathtt{P}\,(\text{funcionamiento correcto en } t | \text{funcionamiento correcto en } t_0) \tag{1}$$

Dado que en el intervalo $[t_0; t]$ puede o no ocurrir una falla, la probabilidad de que el sistema pueda cumplir su función en t puede expresarse como en la ecuación (2). Si no ocurre ninguna falla, luego el sistema podrá seguir cumpliendo su función en t. Además, si llegase a ocurrir una falla, pero el sistema tiene la capacidad de tolerarla, luego el sistema de igual manera podrá seguir cumpliendo su función en el instante t.

$$\begin{aligned} R(t) = {} & \mathtt{P}\,(\text{no ocurrio una falla en } [t_0; t]) \\ & + \mathtt{P}\,(\text{funcionamiento correcto en } t | \text{ocurrió una falla en } [t_0; t])\ \mathtt{P}\,(\text{ocurrió una falla en } [t_0; t]) \end{aligned} \tag{2}$$

En el caso en el que se tuviera un sistema que no comprende ningún mecanismo de tolerancia a fallas, luego la fiabilidad sería exactamente igual a la probabilidad de que no ocurra una falla, ya que la ocurrencia de una falla causaría un funcionamiento incorrecto. Esto no necesariamente representa un problema. Si el sistema en cuestión es tal que puede demostrarse que la probabilidad de que no ocurra una falla es lo suficientemente alta, luego no se requeriría el uso de técnicas de tolerancia a fallas.

En un sistema donde no hay tolerancia a fallas, la fiabilidad quedaría definida como en la ecuación (3) y la única manera de mejorarla sería incrementando la probabilidad de que no ocurra ninguna falla en el intervalo $[t_0; t]$.

$$R(t) = \mathtt{P}\,(\text{no ocurrio una falla en } [t_0; t]) \tag{3}$$

La manera de hacer esto puede ser por ejemplo, utilizando componentes o módulos de muy buena calidad, lo suficientemente confiables como para cumplir con los requerimientos de fiabilidad [8]. Sin embargo, esto puede ser muy costoso, pensando en que un sistema puede tener una enorme cantidad de posibles fallas. No solo eso, sino que esto dificulta la etapa de diseño de un sistema, ya que cualquier error de diseño que no se haya tenido en cuenta puede llegar a causar una falla y por ende un fracaso del sistema. Por el contrario, la tolerancia a fallas plantea permitir que las fallas existan, pero aplicando técnicas para tolerarlas.

Volviendo a la ecuación (2), la probabilidad de que el sistema funcione correctamente a pesar de la falla, está pesada por la probabilidad de ocurrencia de dicha falla. A partir de esto se desprende que aplicar técnicas de tolerancia a fallas para cada una de las posibles fallas puede resultar exhaustivo, principalmente porque deberían conocerse todas las fallas posibles, además de ser algo costoso. Lo que se propone es considerar solo aquellas fallas cuya criticidad es alta.

A modo de ejemplo, una **falla en un sensor de la computadora de vuelo puede generar una lectura incorrecta**. En consecuencia, esto decantará en un **error, es decir, en un cálculo de la ley de control incorrecto**. Finalmente, este error puede llevar al **fracaso de la misión, por ejemplo si el vehículo no es capaz de seguir una trayectoria dada en tiempo y forma**. Esto da a entender que una falla en un sensor es crítica y que por ende requiere la aplicación de técnicas de tolerancia a fallas.

Aquí se habla de falla en un sensor como algo general. Un sensor podría fallar de muchas maneras y debido a muchas razones. Por ejemplo, puede dejar de funcionar por un defecto propio del componente, puede entregar lecturas erróneas debido a interferencias electromagnéticas, por efectos de la temperatura, falta de calibración, etc. Cada uno de estos requeriría la aplicación de un mecanismo tolerante a fallas.

3.2. Causales de Fallas de Hardware y Modelo de Fallas Arbitrarias

Uno de los métodos para aplicar mecanismos de tolerancia a fallas consiste en hacer un análisis de los posibles modos de falla. Un ejemplo es el del análisis *Failure Modes and Effects Analysis* (FMEA). Este consiste en realizar un análisis exhaustivo de los posibles modos de falla más probables y sus posibles efectos en el sistema. En función de este análisis, se toman medidas para tolerar las fallas más críticas. El objetivo de este tipo de análisis suele ser demostrar ante alguna autoridad ceritifcante, que la confianza del sistema se mantiene por encima de cierto valor. Este tipo de análisis suele consumir mucho tiempo y esfuerzo, lo que se traduce en un mayor costo del desarrollo [5].

Una forma de alivianar esta tarea es la de considerar un modelo de falla de hardware más conservador, donde se asume que una falla de hardware consiste en que esta presente un comportamiento anómalo arbitrario, es decir, de cualquier tipo. A este tipo de comportamiento se lo denomina falla bizantina o *Byzantine Fault* en inglés y básicamente consiste en asumir que el elemento que manifiesta la falla presenta un comportamiento arbitrario. Por ejemplo, un sensor puede dejar de funcionar repentinamente y no dar más respuesta, puede dejar de enviar respuesta por un período de tiempo y luego volver a funcionar, podría también enviar datos a un microcontrolador pero que esos datos sean incoherentes, etc. El modelo de falla bizantina no asume modos de falla, sino que el comportamiento es arbitrario [11][4][5]. Se define un sistema tolerante a este tipo de fallas.

Definición 2. ***Sistema Byzantine Resilient****: es aquel capaz de tolerar una cierta cantidad de fallas arbitrarias a la vez.*

Dado que no se asume un modo de falla del hardware, no se requiere un análisis tan exhaustivo como el mencionado FMEA. Considerando el costo y esfuerzo que lleva realizar un análisis de modos de fallas, el hecho de poder contar con un sistema con las características que aquí se mencionan resulta atractivo para alivianar el trabajo relacionado a la validación del sistema tolerante a fallas en cuestión.

A priori, puede parecer que desarrollar un sistema tolerante a fallas arbitrarias representa un trabajo sumamente complejo. La manera de implementar un sistema tolerante a fallas bizantinas es a través del uso de redundancias. Este resultado se toma a partir de un problema teórico denominado *The Byzantine Generals Problem* [6], el cual se presentará más adelante.

3.3. Tolerancia a Fallas a Partir de Redundancias

La principal técnica de tolerancia a fallas es el uso de redundancias [8][10][5]. Esto quiere decir, que se replica el hardware en el sistema y cada réplica realiza la misma tarea en paralelo. De esta forma, si una de las réplicas presenta una falla (arbitraria por ejemplo), esta puede detectarse a partir de la comparación con las demás réplicas, o incluso pasar desapercibida. Utilizando la nomenclatura definida en la sección 3.1, que una falla pase desapercibida quiere decir que no se manifiesta como un error, sino que esta es contenida. A continuación se presentan algunas arquitecturas redundantes para la tolerancia a fallas.

3.3.1. Redundancia Doble

Una arquitectura simple es la redundancia doble. En este tipo de sistemas, dos nodos de un sistema funcionan en paralelo y comparan sus resultados. La comparación permite detectar si los resultados difieren entre sí, lo que se traduce en que ocurrió un error.

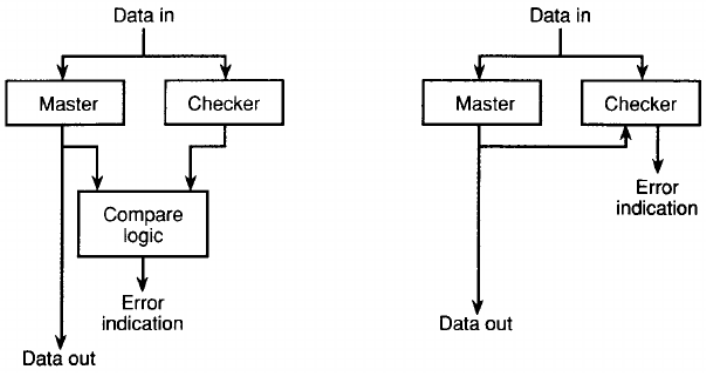
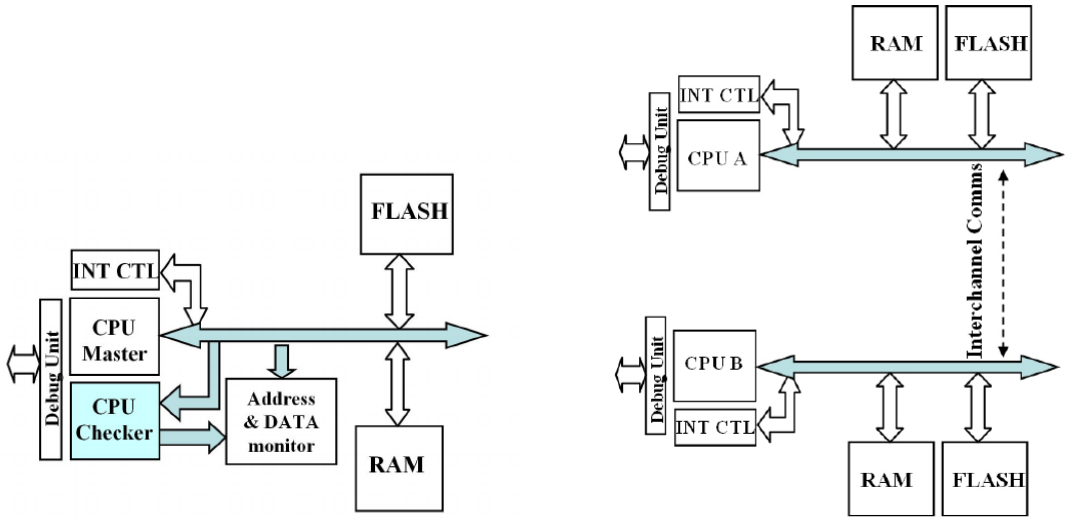


Figura 1: En la figura de la izquierda, dos sistemas ejecutan las mismas operaciones, mientras que otro sistema externo se encarga de comparar las salidas de ambos para detectar errores. En la figura de la derecha, el bloque comparador se encuentra integrado en el sistema *checker*. La imagen fue extraida de [8].

Este tipo de arquitectura permite detectar si ocurrió un error, pero no permite identificar de qué nodo proviene el error. En la figura 1 se muestran dos configuraciones. La configuración de la derecha

puede ser implementada a través de dos CPUs totalmente independientes (a veces denominada *Loosely-Synchronized Dual Processor Architecture*) o a través del uso de un procesador de dos núcleos, donde uno sería el *Master* y otro el *checker*[1]. En esta última, ambos se encuentran sincronizados por estar en el mismo chip y compartir fuente de clock. En la figura 2 se muestra un esquema de ambos casos.



(a) Lockstep dual processor architecture.

(b) Loosely synchronized dual processor architecture.

Figura 2: Se muestran dos casos para un sistema con redundancia doble. La imagen fue extraida de [1].

Debido a que no se puede saber cuál de las dos CPUs cometió el error, esta arquitectura plantea que en el caso en el que la comparación entre ambas CPUs genere una discrepancia en los resultados, cada una de ellas deben ejecutar un algoritmo interno, para detectar si ellas fueron las que cometieron el error o no. En [15] y en [12] se pueden encontrar proyectos de redundancia doble para UAVs.

3.3.2. Redundancia Triple

Esta arquitectura puede encontrarse en la literatura con el nombre *Triple Modular Redundant (TMR) Architecture* [1][8][10][7]. Esta arquitectura consiste en utilizar tres computadoras en paralelo, las cuales computan los mismos resultados. Luego, se comparan los resultados. Se asume que solamente 1 de las 3 presentará una falla a la vez. En dicho caso, los resultados de dos computadoras serán iguales y la de la tercera será distinto, por lo que solamente se descarta el resultado erróneo. En la figura 3 se muestra un diagrama con la arquitectura TMR. Una diferencia de esta arquitectura respecto de la doble redundancia, es el hecho de que puede detectarse cuál de las computadoras falló y además, no es necesario que todas las computadoras ejecuten una rutina para verificar si cometieron el error o no. Esto resulta especialmente útil en sistemas de tiempo real, donde no puede detenerse el sistema para realizar una verificación interna. Esto se denomina *Fault Masking*.

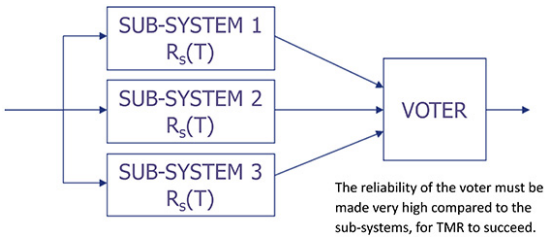


Figura 3: Arquitectura TMR. La imagen fue extraida de [13].

Como indica el texto de la imagen, una cuestión clave de esta arquitectura es el bloque denominado *VOTER*. Debido a que este bloque es el que determina cuál es el resultado correcto, se requiere que la

fiabilidad, $R(t)$, de este sea mucho mayor que la de cada computadora de vuelo. Esto se logra a través del uso de hardware más robusto, lo que resulta en que el bloque *VOTER* sea más costos que cada computadora de vuelo. Por ejemplo, cada computadora de vuelo puede comprender un microcontrolador COTS, mientras que el bloque voter puede estar implementado con un ASIC específico para esa aplicación [4]. Si bien este bloque tiene una fiabilidad mucho mayor, siempre existe la probabilidad de que ocurra un error en este. En cuyo caso, el error puede decantar en un fracaso, por ejemplo si el *VOTER* elige como resultado correcto, aquel que realmente no lo era.

Definición 3. ***Single-Point Failure****: si la arquitectura del sistema es tal que una parte del sistema X fracasa en cumplir su trabajo dentro del sistema, luego el sistema completo fracasará en cumplir su función. En dicho caso, X es un punto único de falla.*

Una forma de combatir esto es replicar los bloques que realizan la votación [8][7]. De esta manera, también pueden enmascararse errores de los bloques que realizan la votación. La arquitectura sería como la que se muestra en la figura 4.

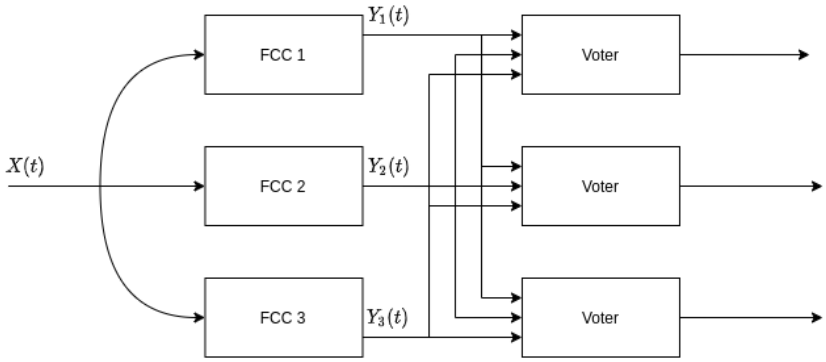


Figura 4: Arquitectura TMR con redundancia en los elementos votantes.

Los tres elementos *Voter* reciben las mismas entradas y en el caso de que ninguno de los *voters* cometa un error, dado que las entradas de los *Voters* son exactamente iguales, luego los tres decidirán por el mismo resultado como el valor correcto.

Esta arquitectura es más compleja que las anteriores, ya que requiere una gran cantidad de nodos, 3 FCCs + 3 bloques votantes, dando un total de 6. Además, pensando en que se argumentó que los votantes generalmente son más confiables que las FCCs, la triplicación del bloque *Voter* encarece mucho al UAV.

Como medida para evitar esto último, los bloques votantes pueden integrarse dentro de cada una de las FCC. Esto quiere decir, que en lugar de tener 3 bloques votantes, las mismas FCC sean las encargadas de realizar la votación. En el artículo [4] se propone que los microcontroladores automotivos ofrecen las interfaces necesarias para implementar una red redundante para tolerar fallas. En el artículo [3], los mismos autores presentan resultados para una arquitectura con redundancia cuádruple, donde los mismos microcontroladores de cada FCC son los encargados de realizar la votación. Para el caso de una arquitectura de redundancia triple, puede diagramarse como en la figura 5.

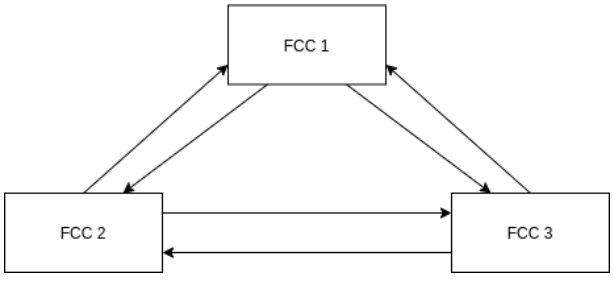


Figura 5: Arquitectura de redundancia trple, donde los bloques votantes son las mismas FCCs. Los votantes se encuentran integrados dentro de cada FCC.

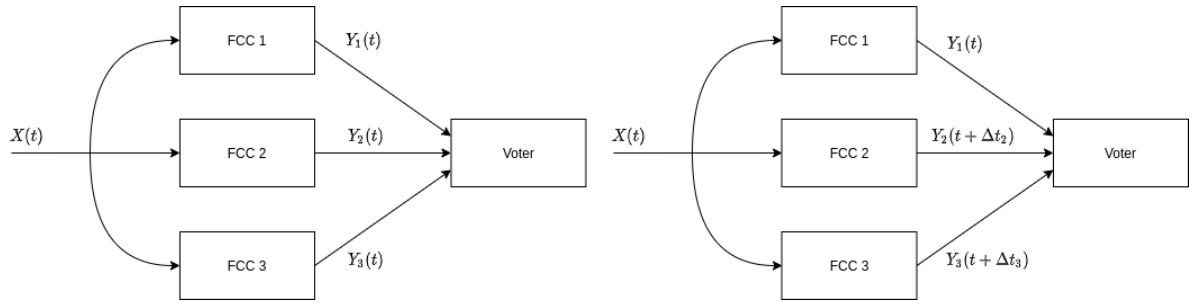
3.4. Algunos Requerimientos de un Sistema Redundante

Si bien el uso de redundancias apunta a incrementar la fiabilidad del sistema y tolerar fallas, es un error pensar que el simple hecho de tener un sistema redundante equivale a un incremento de la fiabilidad [5]. Esto es principalmente por el hecho de que un sistema redundante incluye además las comunicaciones y rutinas necesarias para ejecutar las votaciones. Si estas funcionalidades no son administradas de manera correcta, un sistema redundante solamente traerá más fallas.

3.4.1. Sincronismo de los Nodos

En las arquitecturas antes presentadas, se menciona que se realiza una comparación de los resultados calculados por cada nodo, para detectar/enmascarar errores. Para que el funcionamiento de esta comparación sea adecuado, los nodos deben estar sincronizados. Esto es un requerimiento para sistemas de tiempo real, como el caso de la computadora de vuelo de un UAV.

En la figura 6 se muestra un ejemplo. En el instante t, se presenta una nueva medición de un sensor a las tres computadoras de vuelo. Al comienzo de la misión, todas ellas estarán sincronizadas y generarán un resultado del cálculo de la ley de control que corresponde al mismo intervalo de tiempo. Luego se realiza la votación para elegir el valor correcto. La figura 6b, muestra lo que sucede al cabo de un período de tiempo. Se presenta una nueva medición de un sensor en el instante t. Debido a la desincronización, es posible que las computadoras de vuelo no presenten sus resultados al *Voter* a tiempo, por lo que este asumirá que una de las FCCs no presentó ninguna respuesta. Este caso suele estar contemplado dentro de las posibilidades y correpsonde al caso en el que una computadora de vuelo presentó un error y debido a ello no respondió con ningún valor (por ejemplo, se reinició su procesador debido a un *watchdog*). En esos casos el *Voter* simplemente asume algún valor por defecto.



(a) Computadoras de vuelo al inicio de la misión. (b) Al cabo de un período de tiempo, se desincronizarán.

Figura 6: A medida que transcurra el tiempo, la desincronización entre FCCs impactará en el sistema redundante.

Otra situación que puede presentarse, es que los resultados propuestos por las computadoras de vuelo Y_1, Y_2 e Y_3 correspondan a intervalos de tiempo distintos. Este caso es todavía peor que el anterior, ya que no se encuentra contemplado y los *Voters* simplemente realizarán la votación asumiendo que el dato es válido.

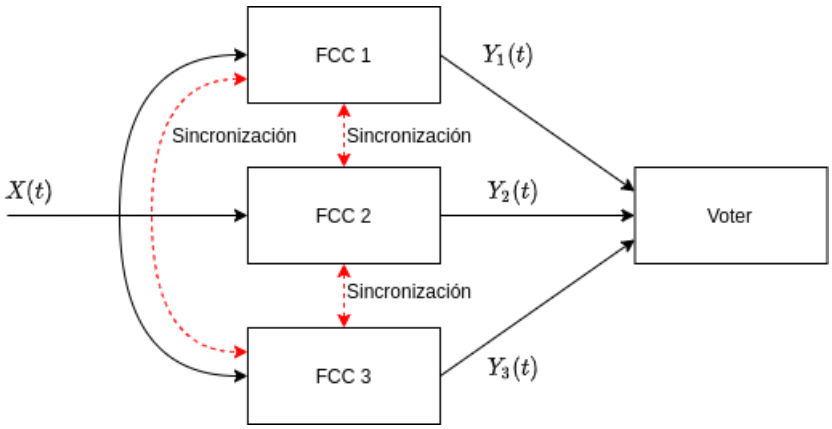


Figura 7: La sincronización entre nodos es necesaria para un correcto funcionamiento de las redundancias.

Se concluye que es mandatorio utilizar alguna técnica de sincronización entre los nodos. Como detalle de la figura 7, se muestra que la sincronización entre nodos presupone otro canal de comuniación más. Otra forma podría ser relegar la tarea de la sincronización al bloque *Voter*, aunque esto nuevamente presenta un punto singular de falla. Como se demostró en esta sección, el sincronismo es un aspecto crítico en el sistema redundante, por lo que se prefiere evitar esto último.

3.4.2. Consenso

Lo que se plantea en esta sección, es que existe la posibilidad de que una de las FCC entregue distintos valores a cada *Voter*. En la figura 8 se muestra una situación en la que la FCC1 entrega dos valores distintos a los demás *Voters*, siendo los valores posibles *True* y *False*. Esto puede deberse por ejemplo a que la FCC1 así lo quiso, debido a una falla muy compleja de analizar y que se manifiesta como un error de esta manera. Otra posibilidad más realista puede ser el hecho de que, debido a que el dato enviado por la FCC1 llegó más tarde al tercer *Voter* que a los demás, este interpretó mal el valor recibido por la línea de comuniación, generando la situación de la figura 8.

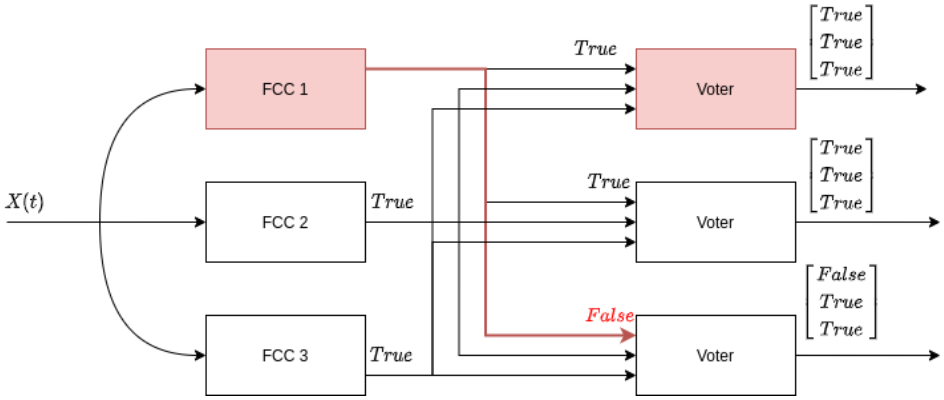


Figura 8: La FCC1 entrega el valor de *True* a un *Voter* y *False* a otro. Los vectores representan los valores sobre los que cada *Voter* debe decidir y votar por un único valor.

Este caso a priori parecería no presentar un problema que la arquitectura TMR no pueda resolver. El primer y segundo *Voter* decidirán por el valor *True*, ya que todas sus entradas son iguales a este valor. El tercer *Voter* también decidirá por el valor *True* ya que 2 de 3 de sus valores son *True*. En definitiva, la arquitectura TMR resuelve el problema del consenso para este caso.

A continuación se plantea un caso diferente. Se analiza la situación en la que cada FCC propone un valor distinto, que fue calculado por su propia observación del escenario en el que se encuentra. Esto podría ser por ejemplo, el valor de algún sensor interno a esta. Las FCC 1, 2 y 3 se encuentran dentro del mismo UAV, por lo que si poseen sensores redundantes, uno esperaría que se obtengan las mismas lecturas (si es que todos los sensores funcionan adecuadamente). Esto puede no ser así, ya que por ejemplo estas pueden presentar pequeñas variaciones por tratarse de lecturas analógicas. De manera de que el algoritmo de control se ejecute de manera consistente en las tres FCCs, ellas deben ponerse de acuerdo en un valor del sensor.

Como se mencionó en la sección anterior, se requiere lograr una sincronización entre computadoras de vuelo redundantes. De manera de ejecutar un algoritmo de sincronización adecuado, las computadoras de vuelo deben compartirle a las demás, un valor asociado a su propio clock interno.

Estos dos últmos escenarios difieren del caso en el que la FCC comparte un valor que puede ser *True* o *False*. Lo que se plantea aquí es un caso en el que cada FCC comparte un valor que corresponde a la propia perspectiva de cada una de ellas. Debido a esto, no existe un valor correcto a transferirle a las demás. Se muestra un ejemplo para la sincronización de FCCs en la figura 9.

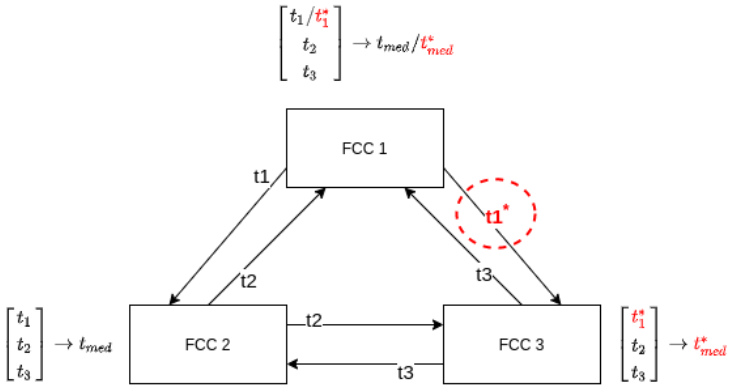


Figura 9: La FCC1 entrega un valor distinto de timing a las demás FCCs

.

En este escenario, la FCC1 entrega dos valores distintos de su *clock* a las demás FCCs. Cada una de ellas luego realiza un promedio para llegar a un único valor. Lo que se observa es que las FCC2 y FCC3 calcularán un valor promedio distinto, es decir, no se sincronizarán. Una posible solución podría ser que las FCCs hagan un nuevo intercambio, con los valores promedio calculados y realicen una votación interna. Esto se muestra en la figura 10.

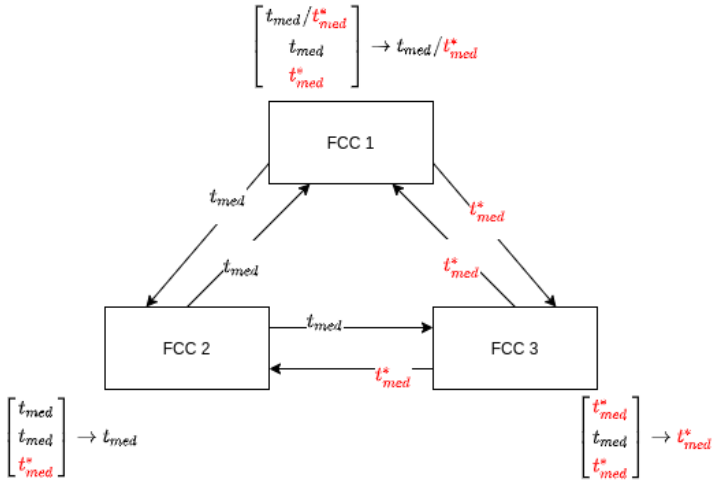


Figura 10: Luego de calcular los promedios, las FCCs intercambian sus resultados. Nuevamente, la FCC1 comete una falla en el envío del dato.

Esta última situación, donde la FCC1 nuevamente comparte dos valores distintos a las demás, puede llevar a que las computadoras de vuelo no se sincronicen, algo que como ya se mencionó, es crítico para la correcta ejecución del algoritmo de tolerancia a fallas.

Podría argumentarse que es demasiado pesimista pensar que la FCC1 puede producir la misma falla 2 veces de manera consecutiva, ya que existe una baja probabilidad de que ello suceda. Sin embargo, la situación planteada en esta sección puede tratarse como un tipo de falla antes mencionado, la falla bizantina, ya que contempla fallas de hardware que se manifiestan como comportamientos arbitrarios. El ejemplo presentado en esta sección, se corresponde con un comportamiento arbitrario.

3.5. Redundancia Cuádruple: *The Byzantine Generals Problem*

En las secciones anteriores se habla de un modelo de falla de hardware arbitraria, denominada falla bizantina. El nombre proviene de un problema denominado *The Byzantine Generals Problem*, formalizado en [6]. Este paper plantea un escenario que sirve como base para el análisis de fallas arbitrarias. En esta sección, se presenta brevemente el problema y su relación con la tolerancia a fallas. El análisis completo puede encontrarse en el trabajo original [6]. Otros trabajos que tratan el mismo problema son [9] y [14].

Este último, presenta el diseño de una computadora de vuelo tolerante a fallas que utiliza los resultados del *Byzantine Generals Problem* para realizar distintas tareas de redundancia.

3.5.1. Presentación del Problema

El secenario que se plantea es el siguiente: un grupo de generales, cada uno liderando su respectivo ejército, se encuentran rodeando una ciudad enemiga. Todos los generales deben ponerse de acuerdo, respecto de si la mejor decisión es atacar la ciudad o retirarse. Independientemente de cuál sea la decisión, todos deben tomar la misma decisión.

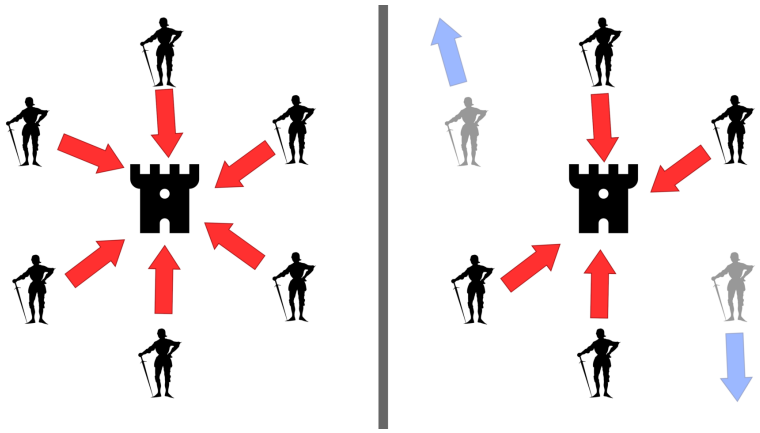


Figura 11: La situación que se presenta, donde los generales deben tomar una decisión común. La figura de la derecha muestra la situación donde algunos generales atacan mientras que los generales traidores no lo hacen. La imagen se extrajo de [2].

Debido a que los generales se encuentran alejados unos de otros, estos solo pueden comunicarse con mensajes uno a uno, por ejemplo con un soldado que lleve un mensaje a caballo, desde un ejército a otro ejército. Por ejemplo, si el general 1 decide que lo mejor es atacar, este enviará un mensaje a cada uno de los otros generales para informarlse que su voto es por atacar la ciudad.

Además, el problema plantea la posibilidad de que algunos de los generales sean traidores. Esto quiere decir que ellos pueden actuar de manera independiente a la decisión común.

Cada general vota por atacar o por retirarse, y la decisión final será la que tenga más votos. **Esto quiere decir que cada general debe conocer la opinión de los demás generales, para así poder coincidir en el resultado final, es decir, atacar o retirarse**. El problema, es que los generales traidores pueden mentir o enviar información diferente a cada general. Esto último se refiere a que un traidor puede decirle a un general que su opinión es "atacar" y a otro general decirle que su opinón es "retirarse". **Esto último implica que todos los generales deben disponer de la misma información para así poder tomar la misma decisión y que los traidores no perjudiquen el consenso al que deben llegar los generales**. Por ejemplo, si se tienen 3 generales y los generales 1 y 2 reciben los votos:

$$\text{General 1} = \begin{bmatrix} Atacar \\ Retirarse \\ Atacar \end{bmatrix}$$

$$\text{General 2} = \begin{bmatrix} Atacar \\ Retirarse \\ Retirarse \end{bmatrix}$$

Esto llevará a que el General 1 ataque mientras que el General 2 se retire. El error fue causado por la presencia del traidor, el General 3.

3.5.2. Solución al Problema

El paper plantea una solución para este problema, pero que solamente es válida en el caso en el que se tienen m traidores y al menos $3m + 1$ generales en total. En la figura 12 se muestra un caso para 4 generales y 1 traidor. El General 1 es el traidor y le envía información diferente a cada general.

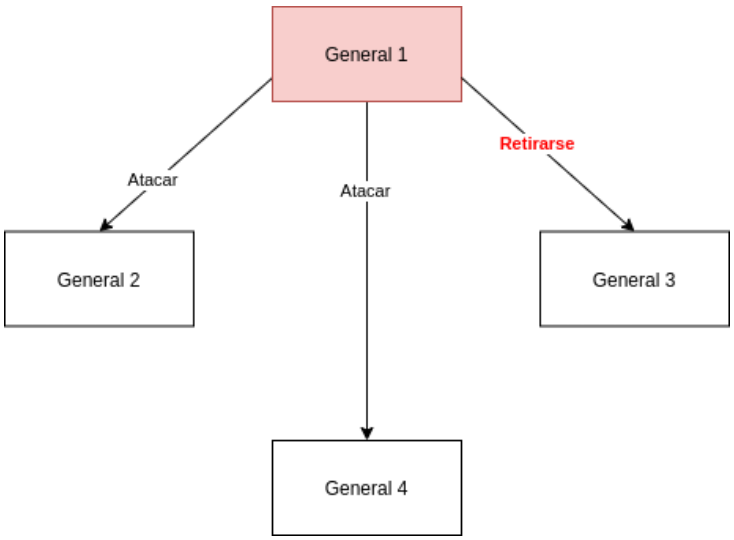


Figura 12: El general 1 es un traidor y le envía información conflictiva a los demás generales.

Como fue mencionado, para llegar a una decisión común, todos los generales deben conocer la opinión de los demás. El problema en este caso es que el General 1 envió una información difernte a sus pares. Para resolver esto, el algoritmo plantea realizar un segundo intercambio de mensajes como el de la figura 13.

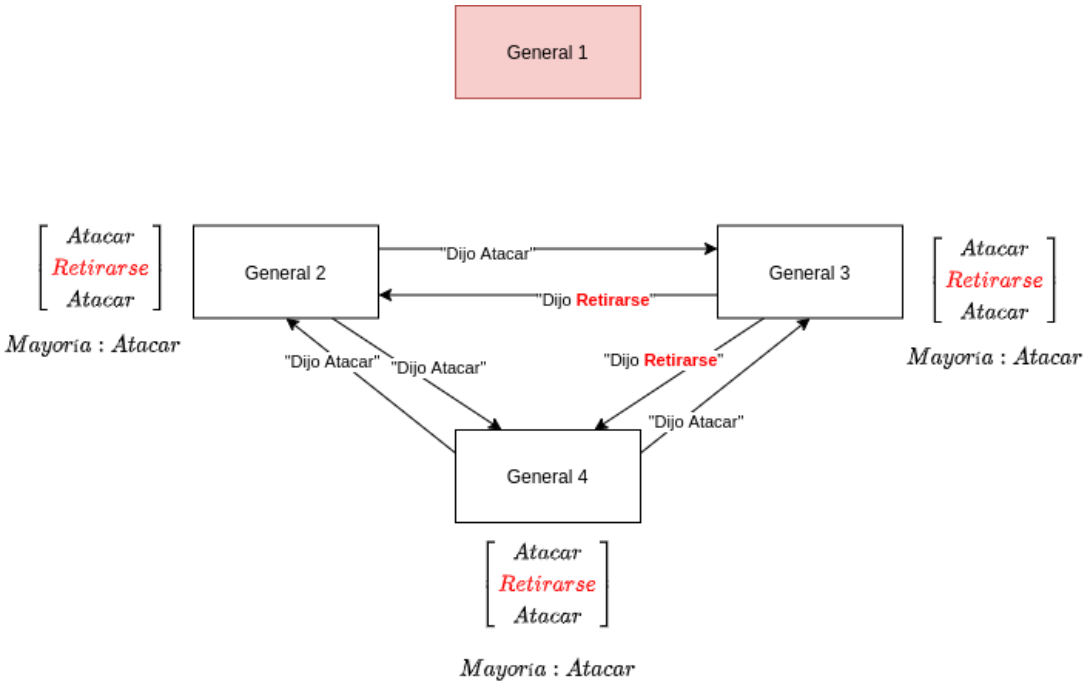


Figura 13: Se produce un intercambio entre los demás generales, para ponerse de acuerdo respecto de si el General 1 dijo “Atacar” o “Retirarse”.

Al lado de cada General, se muestra un vector que contiene los mensajes informados por los otros Generales, respecto del voto del General 1. Lo que se muestra es que en este caso, los Generales leales logran ponerse de acuerdo en que el General 1 dijo “Atacar”, es decir, llegan a un consenso. Para continuar con el algoritmo, se debe repetir el mismo procedimiento de intercambio de mensajes para los otros tres generales. Al finalizar todos los intercambios de mensajes, los Generales leales tendrán la misma información respecto a los votos de sus pares y llegarán a la misma decisión final.

3.5.3. Relación del Problema con la Tolerancia a Fallas

Si bien el análisis del problema se plantea como un juego, la motivación surge de realizar un análisis de tolerancia a fallas a partir de redundancias. En [14], los mismos autores de *The Byzantine Generals Problem* presentan un trabajo de diseño y análisis de una computadora de vuelo tolerante a fallas. Este es anterior a la formalización del problema, pero menciona que la necesidad del consenso entre cada nodo de la red redundante, es un requerimiento para aplicar los mecanismos de tolerancia a fallas correctamente.

Se traza un paralelismo entre los generales que deben llegar a un consenso con una serie de computadoras interconectadas, cuyo objetivo es también generar consenso respecto de alguna variable.

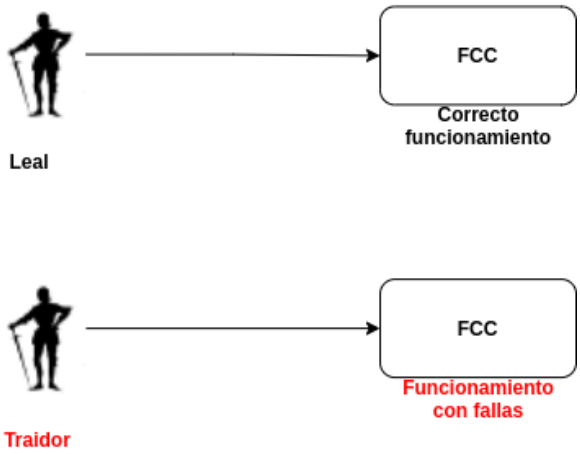


Figura 14: En el problema, un general leal representa un nodo, en este caso una computadora de vuelo, que funciona correctamente. Un General traidor es equivalente a una computadora de vuelo que presenta fallas.

Los generales traidores representan a las computadoras de vuelo que presentan fallas. En [14] se presenta un ejemplo de la aplicación del algoritmo de *The Byzantine Generals Problem* para lograr sincronizar a los nodos. A continuación, se analiza brevemente este problema, con motivo de demostrar su importancia en los sistemas redundantes tolerantes a fallas.

Se plantea una situación como la de la figura 12, pero se reemplazan a los generales por computadoras de vuelo. En este caso, las computadoras de vuelo deben sincronizarse. Para lograrlo, ellas comparten un valor de *timestamp*, que pueden utilizar para ajustar sus clocks. En la figura 15 se muestra un escenario en el que una de las computadoras de vuelo presenta una falla tal que le informa un valor distinto a cada una de las computadoras de vuelo.

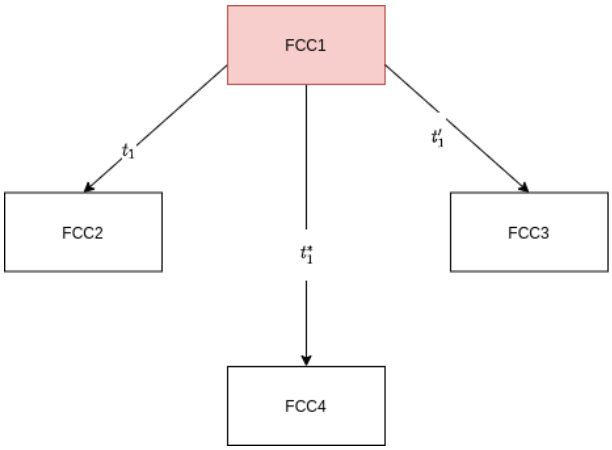


Figura 15: Debido a una falla, la computadora de vuelo 1 le entrega valores distintos de timestamp a las demás.

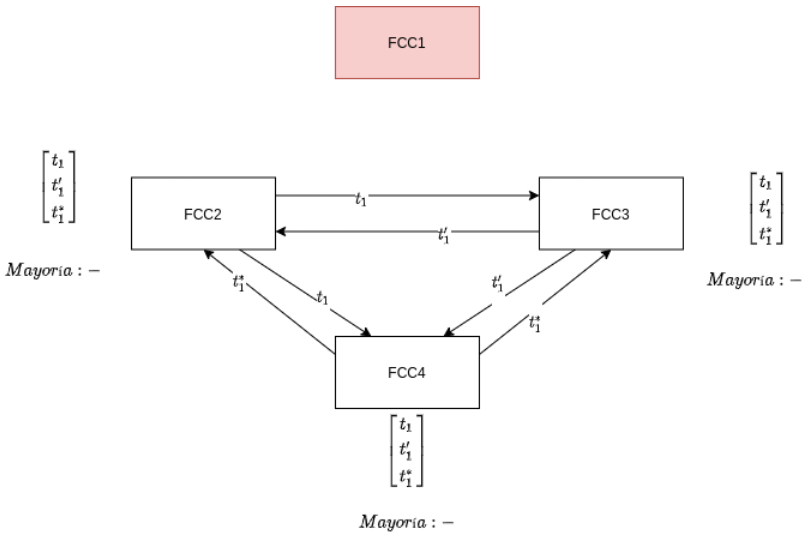


Figura 16: Debido a una falla, la computadora de vuelo 1 le entrega valores distintos de timestamp a las demás.

A través de un segundo intercambio, las FCC 2, 3 y 4 llegan a la conclusión de que el *timestamp* de la FCC1 no es claro. En este caso, descartan el valor. Luego de hacer todos los intercambios de *timestamp*, las FCCs podrán aplicar internamente la sincronización, por ejemplo, calculando un promedio de todos los timestamp. **Dado que todas las FCCs tendrán la misma información de *timestamp* entregado por las demás FCCs, luego todas llegarán al mismo promedio y se sincronizarán**.

Un aspecto interesante es el hecho de que en el paper original, se compara a un general traidor con una computadora con fallas. Como se mencionó, los generales traidores pueden tener cualquier comportamiento. Esto lo que quiere decir es que las fallas presentadas por las computadoras de vuelo pueden ser justamente de cualquier característica, incluso al extremo de presentar un comportamiento malicioso, con el objetivo de perjudicar al sistema [5]. Esto sienta las bases para la tolerancia a fallas de hardware arbitrarias.

La implementación del algoritmo tolerante a fallas arbitrarias resulta costoso. Para poder tolerar fallas provenientes de una FCC se requiere un total de 4 computadoras de vuelo. Además, debe haber una interconexión entre las 4 computadoras y ellas deben intercambiar información continuamente para poder detectar y enmascarar la falla.

En las próximas secciones, se analizará el caso propuesto en este trabajo y se mostrará que bajo ciertas condiciones, puede no ser necesario utilizar 4 computadoras de vuelo para tolerar fallas provenientes de 1 de ellas.

4. Arquitectura de Redundancia Propuesta

5. Conclusiones

Referencias

[1] Massimo Baleani y col. «Fault-tolerant platforms for automotive safety-critical applications». En: *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*. 2003, págs. 170-177.

[2] Wikipedia contributors. «Byzantine fault». En: *Wikipedia* (jul. de 2023). URL: `https://en.wikipedia.org/wiki/Byzantine_fault#`.

[3] Sebastian Hiergeist y Georg Seifert. «Implementation of a SPI based redundancy network for SoC based UAV FCCs and achieving synchronization». En: *2018 IEEE/AIAA 37th Digital Avionics Systems Conference (DASC)*. IEEE. 2018, págs. 1-10.

[4] Sebastian Hiergeist y Georg Seifert. «Internal redundancy in future UAV FCCs and the challenge of synchronization». En: *2017 IEEE/AIAA 36th Digital Avionics Systems Conference (DASC)*. IEEE. 2017, págs. 1-9.

[5] Jaynarayan H Lala y Richard E Harper. «Architectural principles for safety-critical real-time applications». En: *Proceedings of the IEEE* 82.1 (1994), págs. 25-40.

[6] Leslie Lamport, Robert Shostak y Marshall Pease. «The Byzantine generals problem». En: *Concurrency: the works of leslie lamport*. 2019, págs. 203-226.

[7] Robert E Lyons y Wouter Vanderkulk. «The use of triple-modular redundancy to improve computer reliability». En: *IBM journal of research and development* 6.2 (1962), págs. 200-209.

[8] Victor P. Nelson. «Fault-tolerant computing: Fundamental concepts». En: *Computer* 23.7 (1990), págs. 19-25.

[9] Marshall Pease, Robert Shostak y Leslie Lamport. «Reaching agreement in the presence of faults». En: *Journal of the ACM (JACM)* 27.2 (1980), págs. 228-234.

[10] Vinod B Prasad. «Fault tolerant digital systems». En: *IEEE Potentials* 8.1 (1989), págs. 17-21.

[11] Edo Roth y Andreas Haeberlen. «Do Not Overpay for Fault Tolerance!» En: *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2021, págs. 374-386.

[12] Federico Fidencio Solano Pérez. «Development of a Redundancy System for Autopilots». 2019.

[13] *Triple Modular Redundancy*. URL: `https://www.layerzero.com/Innovations/Industry-Firsts/Triple-Modular-Redundancy.html`.

[14] John H Wensley y col. «SIFT: Design and analysis of a fault-tolerant computer for aircraft control». En: *Proceedings of the IEEE* 66.10 (1978), págs. 1240-1255.

[15] Xiaolin Zhang, Haisheng Li y Dandan Yuan. «Dual redundant flight control system design for microminiature UAV». En: *2015 2nd International Conference on Electrical, Computer Engineering and Electronics*. Atlantis Press. 2015, págs. 785-791.