INFORME DE ACTIVIDAD

Sistema multi-UAV para inspección de infraestructuras

Motivación

El objetivo es disponer de una formación de UAVs con capacidades de sensado que puedan ser tanto homogéneas como heterogéneas, en orden de que puedan ser organizados en conjunto para inspeccionar una cierta infraestructura, como la base del pilar de un puente, potenciales grietas en zonas de difícil acceso, etc. Para lograr esta organización, se propone un esquema del tipo líder-seguidor en la que un UAV actúa como líder del equipo y los demás adaptan su movimiento a la trayectoria que sigue el líder mientras cumplen una serie de restricciones. El líder, para inspeccionar la infraestructura, debe seguir una trayectoria comandada por el usuario mediante puntos de vista. Mientras tanto, el resto de UAVs siguen al líder y proporcionan vistas complementarias o mediciones alternativas con otros sensores. El usuario puede especificar ángulos complementarios entre los UAVs así como la distancia de inspección respecto a la infraestructura. Hay dos posibles aplicaciones que motivan este sistema de inspección multi-UAV.

Realidad aumentada

Será de gran para la aplicación que el sistema proporcione realidad aumentada a los operarios humanos mientras ejecutan la tarea de inspección. El operario recibirá imágenes de las vistas complementarias que provienen de la formación de los UAVs que están ejecutando la tarea. Además, con las imágenes emitidas en tiempo real por los UAVs se podría construir y proporcionar al operario una vista en 360º de la escena de forma que pueda navegar a través de una representación detallada de la zona que se está inspeccionando. Otra alternativa sería mapear en 3D la zona mediante algún algoritmo que lo permita. La información heterogénea proveniente de los diferentes sensores también puede ser usada para presentar información extra al operario como puede ser por ejemplo la fusión de imágenes térmicas con imágenes RGB. Durante la operación, el operario podría modificar, si lo desea, los ángulos relativos o las distancias entre el UAV líder y los UAVs seguidores, obteniendo así diferentes configuraciones de la vista completa.

Introducción del módulo que se está implementando

El problema planteado se puede dividir en diferentes módulos, de mayor o menor nivel de abstracción, que en conjunto implementaría el total de la propuesta comentada. El objetivo de las actividades llevadas a cabo desde el inicio del contrato es el de desarrollar una arquitectura de software que permita organizar al equipo de UAVs y repartir las tareas que hayan de llevarse a cabo en conjunto. La solución que se proponga para el planificador de alto nivel debe asegurar que las tareas se realizan en todo momento de forma segura, sin colisiones, teniendo en cuenta las limitaciones de recursos, las capacidades sensitivas de cada UAV, y otras restricciones. La arquitectura debe ser capaz de detectar situaciones inesperadas, como puede ser un pronto agotamiento de la batería de uno de los UAV que se encuentren en formación, actuar y re-planificar

sobre la marcha, haciendo que el UAV al cual se le está agotando la batería se dirija hacia la estación de carga para recargar o sustituir su batería, y reasignando todas las tareas teniendo en cuenta esta nueva información disponible. Otra situación similar ante la cual la arquitectura de control debe ser capaz de actuar sería la perdida de conexión con alguno de los UAVs implicados en una tarea. Ante una situación de este estilo se debe asegurar la integridad del equipo desconectado al tiempo que se re-planifican las tareas para cubrir la inesperada desconexión.

Todo esto ha de llevarse de la forma más eficiente posible, realizando el reparto de tareas y la planificación de tareas para cada UAV de forma que la solución se aproxime lo máximo posible a su forma óptima, teniendo en cuenta para ello detalles como los anteriormente mencionados: capacidades de sensado de los diferentes UAV, niveles de batería de cada uno de ellos, posiciones actuales, plan de tareas actualmente asignado, previsión del instante de recarga, etc.

Para la implementación del módulo descrito se establecieron resumidamente los siguientes pasos:

- Desarrollo e implementación del algoritmo de planificación. El algoritmo se desarrollará empleando la librería para robótica ROS, programando enteramente en el lenguaje de programación C++.
- Simulación de la arquitectura. Haciendo uso de las herramientas que brinda ROS y de los diferentes simuladores y emuladores disponibles en la comunidad, se simulará un escenario realista en el que aparezcan los diferentes elementos implicados en el problema y se pondrá a prueba la arquitectura desarrollada.
- Validación de la arquitectura. Finalmente se realizarán los experimentos que sea necesario para observar como si la arquitectura se comporta de la forma esperada ante cada una de las diferentes situaciones en las que se podría ver envuelta durante una misión real. En esta fase se validará la solución planteada o por el contrario, se detectarán y solucionarán problemas que no se habían tenido en cuenta.

Cronología de actividades realizadas hasta la fecha

Al momento del inicio del contrato como becario, se disponía de conocimientos medios de programación en Python, principalmente enfocado al análisis de ficheros y datos; conocimientos algo avanzados en lenguaje C que eran de aplicaciones generales; conocimientos muy básicos de ROS obtenidos durante la formación académica recibida durante el grado de ingeniería electrónica, robótica y mecatrónica; altos conocimientos en el uso y la programación tanto de MATLAB como de Simulink; así como conocimientos en otros lenguajes de programación menos importantes para la tarea encomendada. También se contaba con alta familiaridad utilizando sistemas basados en Linux, con el uso de repositorios y con el uso de Makefile para compilación.

A continuación se intentará reconstruir al cronología de actividades llevadas a cabo desde el momento de inicio de la beca hasta la fecha, explicando el objetivo que se buscaba con cada una de ellas o el motivo por el que se realizaron.

Formación en ROS y C++

Dado que los conocimientos sobre ROS eran muy básicos, y que los conocimientos sobre C++ eran despreciables, primeramente se realizó una labor de formación en lenguaje C++, en el uso de

simuladores y en el uso de librerías como ROS, ActionLib y Rviz y otras herramientas que en su momentos se pensó serían de utilidad. Se desconoce exactamente la duración de esta actividad. Cabe comentar que realizar una labor de formación como esta, sin conocimientos previos, da conocimientos suficientes para comenzar a pensar soluciones de la forma correcta, teniendo en cuenta el objetivo para el cual está diseñado cada lenguaje de programación, el uso que se espera de ellos, y todas las opciones y herramientas disponibles, pero que a la hora de programar, pasado el periodo de diseño, probablemente se necesite repasar lo aprendido así como estudiar problemas y detalles específicos del lenguaje de programación que vayan apareciendo y que se vayan necesitando. Algunas semanas de la actividad hasta la fecha se han tenido que dedicar casi enteramente al estudio de detalles específicos del lenguaje C++ o de la librería ROS para conseguir cierto comportamiento deseado o para solucionar errores de compilación, lo cuales no son para nada triviales y pueden dar largos quebraderos de cabeza cuando se desconocen los detalles de la implementación de cada arquitectura de software.

Lectura de artículos y documentación útil sobre PILOTING

Una vez finalizada la formación en los lenguajes de programación y las librerías que serían base para el diseño del planificador, se pasó a estudiar el problema a solucionar y a leer sobre el contexto en el que este se encontraba. Dado que el módulo a implementar ha de conectarse con otros que son tanto de mayor como de menor nivel, es importante conocer bien qué hacen esos otros módulos, cómo reciben o envían la información, qué información necesitan, en qué situaciones pueden llamar a este módulo, y qué condiciones se deben de cumplir para poder llamar a módulos de menor nivel entre otras cosas. También se desconoce ahora mismo el tiempo que se dedicó a esta tarea.

Máquinas de estados Vs Árboles de comportamiento

Una vez asimilado el contexto y los requisitos del proyecto, y antes de proceder con el diseño de la solución, se dedicó un tiempo a estudiar las diferencias entre las máquinas de estados finitas (FSM), con las cuales se estaba familiarizado en aquel momento, y los árboles de comportamiento (BT), de los cuales se desconocía completamente su existencia y funcionamiento. Durante este tiempo no solo se prestó atención a las ventajas e inconvenientes de cada herramienta, sino que también se estudiaron las diferentes librerías para C++, preferiblemente compatibles con ROS, que estaban disponibles [1][2][3][4] y se valoró la calidad de la documentación, la cantidad de ejemplos, cruciales para aprender rápidamente a usar una librería cuando aún se es un usuario poco experimentado del lenguaje de programación, la facilidad de uso y el soporte de la librería entre otras cosas.

Aunque la cantidad de personas que emplean librerías de FSM es mayor que en el caso de los BT, y por tanto, es mucho más probable encontrar información en foros de Internet relativa a algún detalle concreto que se necesite y que no esté bien documentado, las ventajas que ofrecen los árboles de comportamiento frente a las máquinas de estado finitas fueron determinantes para decidir emplear este tipo de arquitecturas. Tras una primera formación empleando una librería para BT que ya había sido usada puntualmente por un miembro del grupo [2], se descartó como librería a emplear por la escasa documentación y la falta de características que se consideraban necesarias. Finalmente se encontró una librería que parecía tener una documentación bastante mejor, soporte y mantenimiento aún activos y ejemplos simples de código que se podían emplear como punto de partida [3], y que además venía acompañada de una herramienta para diseñar BT gráficamente [4]. El punto negativo

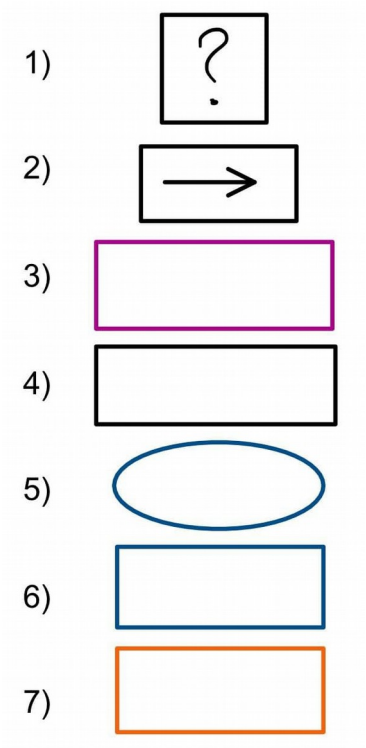
es que no parece tener aún muchos usuarios ni un apartado de preguntas y comentarios, y por tanto, ante cualquier problema o duda con la librería, no se dispondrá de ningún tipo de ayuda.

Diseño del árbol de comportamiento

Diseñar el árbol de comportamiento no fue una labor trivial, ya que se estaba habituado a diseñar pensando en máquinas de estado, pero no pensando en árboles de comportamiento. Además, para cada comportamiento deseado no hay una única implementación posible, lo que hace más complicado el diseño cuando no se tiene la intuición suficiente para saber qué forma es mejor.

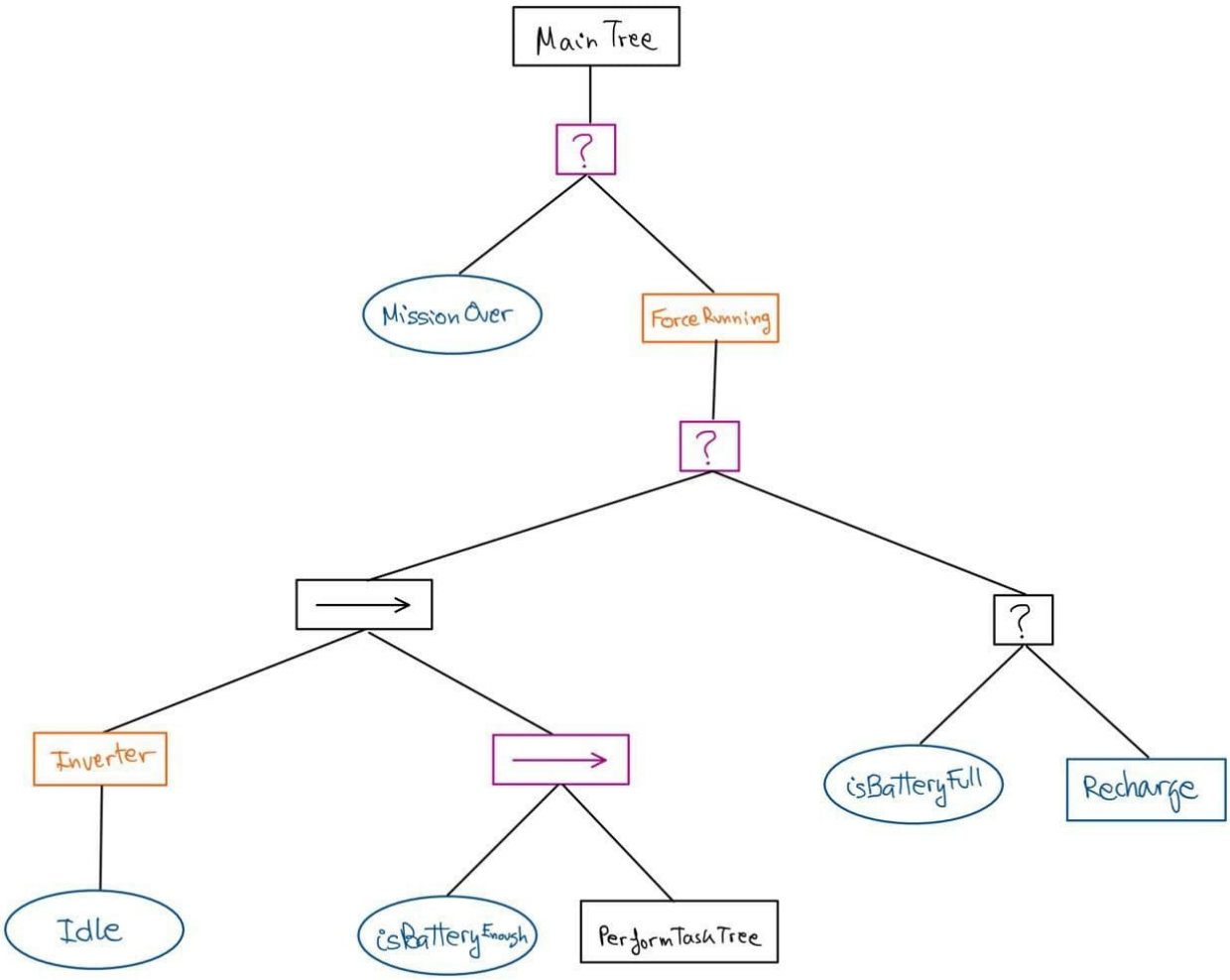
Para tratar de ganar experiencia con la que desarrollar cierta intuición y los conocimientos necesarios para elaborar desde cero un BT, se dedicó un tiempo a reunir y estudiar toda la información posible sobre los árboles de comportamiento que se encontrara por Internet [5][6][7]. Esto fue posible gracias a que los BT sí que son algo más común en el desarrollo de videojuegos [7].

El diseño del BT no ha permanecido invariante desde que se hizo, sino que ha sufrido modificaciones puntuales fruto de reuniones y revisiones del trabajo. Estas modificaciones principalmente buscaban solucionar algún problema detectado ya fuera en algún re-estudio teórico o tras la observación de algún comportamiento indeseado en pruebas de simulación. Antes mostrar y proceder con la explicación del funcionamiento del diseño de BT propuesto, se comentarán brevemente los tipos de nodos disponibles en la librería seleccionada y el funcionamiento de cada uno de ellos.

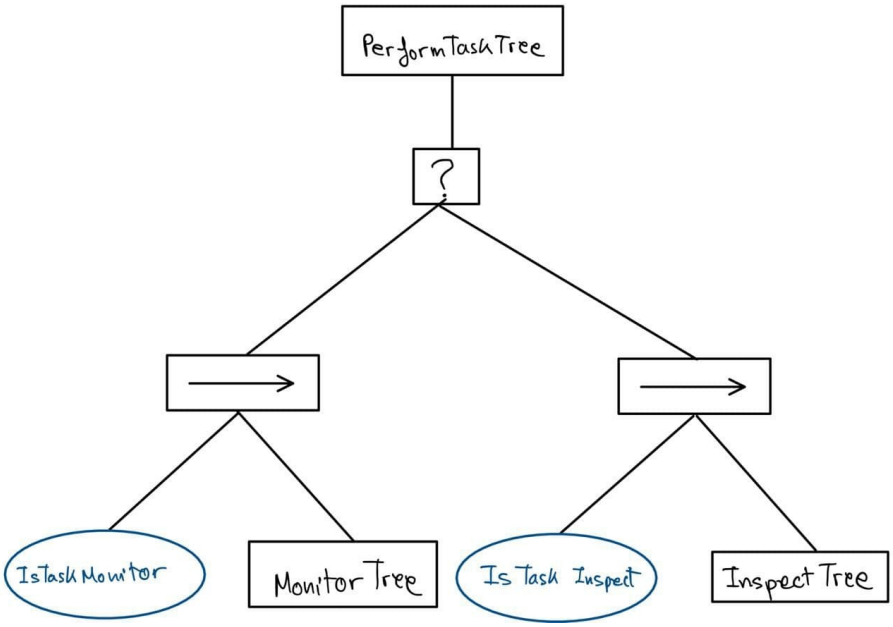


El nodo tipo 1) es un nodo de control de decisión, en inglés llamado “*Fallback*”. Su comportamiento es el de llamar uno a uno a cada uno de sus hijos, de izquierda a derecha, hasta que uno devuelve “*SUCCESS*”, devolviendo él también “*SUCCESS*”, o devolviendo “*FAILURE*” en caso de que todos sus hijos devuelvan “*FAILURE*”. El nodo 2) es un node de control del tipo secuencia. Cuando un hijo devuelve “*SUCCESS*”, llama al siguiente. En caso de que alguno

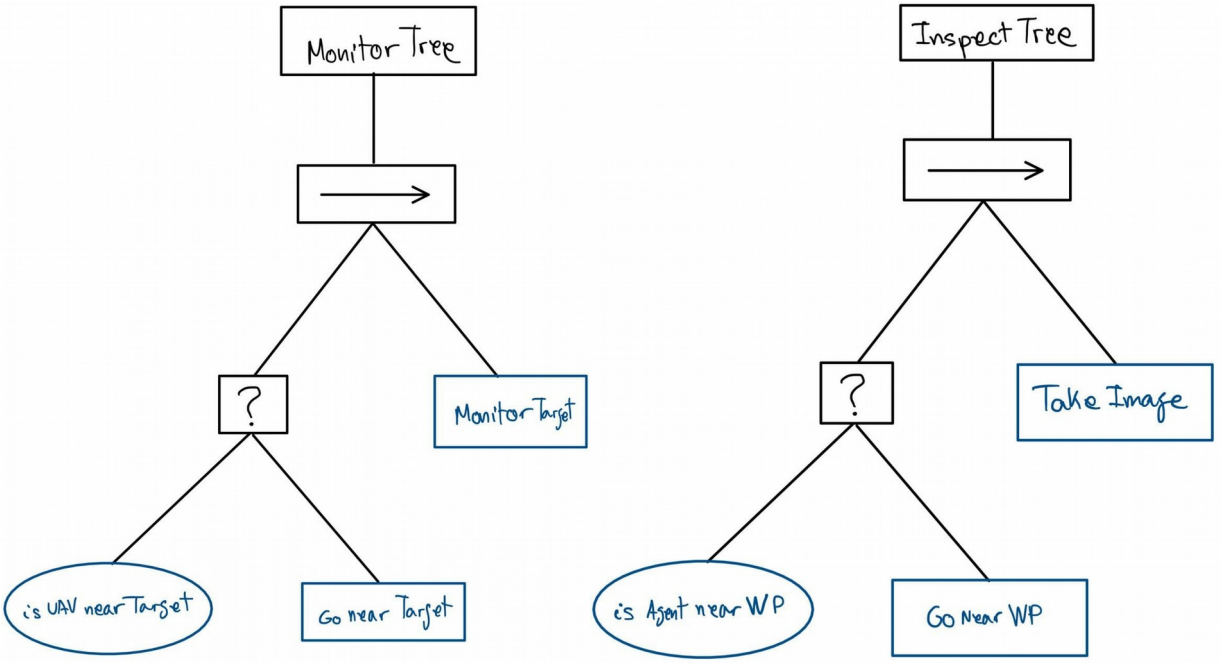
devuelva "*FAILURE*", la secuencia se para y el nodo devuelve también "*FAILURE*". Si todos lo hijos se han ejecutado y han devuelto "*SUCCESS*", el nodo también devuelve "*SUCCESS*". Si el nodo es del color de 3), significa que es un nodo reactivo, lo que significa que, aunque un hijo esté ejecutándose, se re-evalúan los hijos previos. El color 4) es para los nodos normales. Si un hijo devuelve "*RUNNING*", se le vuelve a llamar en la ejecución siguiente. En azul se representan los nodos sin hijos, que pueden ser del tipo 5), que es un nodo para comprobar una condición, o del tipo 6), que ejecuta una acción. Por último, en color 7) se representan los nodos decoradores, cuyo comportamiento dentro del BT es programable.



Esta es la base del árbol de comportamiento implementado. Primeramente se comprueba que la misión no haya terminado, de forma que el BT siga ejecutándose hasta que esta finalice. En caso contrario se ejecuta el resto del árbol. En caso de que el UAV sobre el cual se ejecuta el BT tenga asignada alguna tarea, se comprueba si se tiene batería suficiente y en caso afirmativo se ejecuta el sub-árbol para las tareas. Si el UAV no tiene ninguna tarea asignada, se comprueba el nivel de batería y se recarga. Esta estructura está diseñada de forma que, si el UAV se desconecta, o si el nodo "*isBatteryEnough*" devuelve "*FAILURE*", se vacía la cola de tareas asignada como plan al UAV en cuestión y este, al detectar en la siguiente iteración que se encuentra ocioso, "*Idle*", se dirigirá de forma segura a recargar. En ambos casos, será ejecutada una re-planificación de tareas.



Este es el sub-árbol para comprobar qué tarea es la que se debe ejecutar y llamar consecuentemente a uno u otro sub-árbol. En este punto se podrían llamar directamente a los módulos del nivel inferior, pero se decidió que el control no se le pasaría a los módulos de niveles inferiores hasta que el UAV se encuentre cerca de la zona donde haya de realizar su tarea.



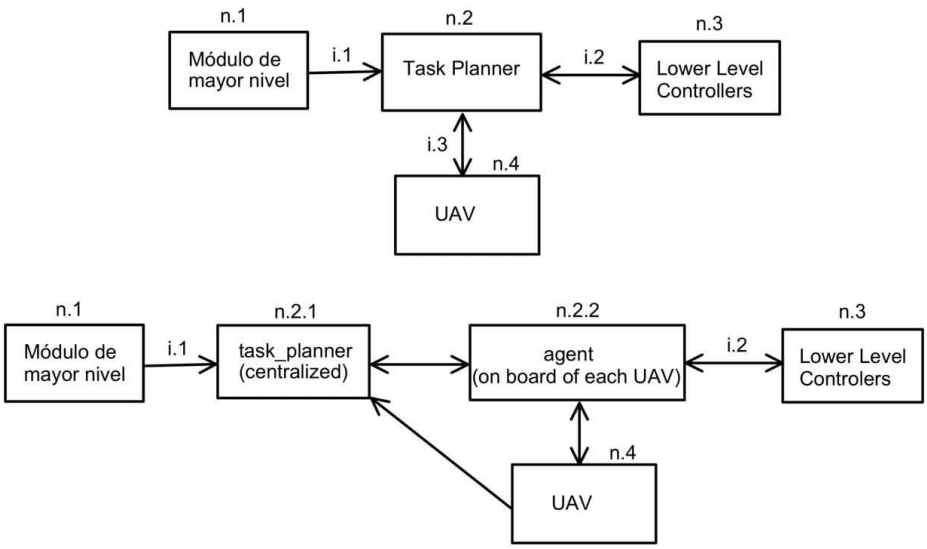
Ahora sí, estos últimos dos sub-árboles son los que llaman a módulos y controladores de niveles inferiores y les pasan el control para que ejecuten la tarea.

Destacar que este árbol de comportamiento no se encarga de realizar el reparto y la planificación de tareas a largo plazo como tal, sino que este se ejecuta sobre cada UAV y se encarga de llevar a cabo la ejecución del plan que se le ha asignado. Este BT es además el responsable de detectar, comunicar y actuar ante cualquier situación inesperada como un pronto agotamiento de la batería o una desconexión. A su vez, este submódulo se encarga de comunicar al planificador todos los detalles y los eventos que vayan sucediendo para que este decida si es necesario re-planificar toda la misión o no. El objetivo de este BT es conseguir que en los UAVs no haya toma de decisiones como

tal, sino que toda la inteligencia se encuentre centralizada y que todos los comportamientos y decisiones que un UAV pueda llegar a tomar estén contenidas y precalculadas dentro de la estructura del propio BT, que está diseñado de forma que asegure el buen funcionamiento y la seguridad del equipo ante cualquier circunstancia.

Diagrama de nodos de ROS

La solución propuesta se compone de una serie de nodos de ROS con diferentes funcionalidades y que se conectan y comunican entre ellas a través de unas interfaces que se explicarán más adelante y que están numeradas en la siguiente imagen de forma esquemática.



Desde la perspectiva del módulo implementado, la estructura de nodos se ve como está representado en la parte superior de la imagen. Se diferenciarían 3 conexiones distintas: la conexión con los módulos de nivel superior, la conexión con los controladores de bajo nivel, y la comunicación con los UAV. A su vez, el módulo marcado como n.2 se compone de dos partes, el planificador de tareas, que es un nodo centralizado, y que se encargará principalmente del reparto óptimo de tareas; y los nodos agente, de los cuales habrá una instancia por cada UAV y se ejecutarán sobre cada uno de ellos.

La idea es que a través de la interfaz i.1, el n.2.1 reciba asíncronamente tareas provenientes de los módulos de mayor nivel, n.1. La información se propaga también de forma asíncrona hacia n.2.2 y los controladores de bajo nivel. Se consideran dos tareas posibles, "*Monitor*" e "*Inspect*". Dependiendo de la tarea, a través de i.1, i.2 e i.3 viajarán unos datos u otros.

Las flechas que conectan los nodos en el diagrama indican la direccionalidad de la comunicación, pudiendo haber comunicaciones unidireccionales o bidireccionales entre nodos.

Nodo Planner

Este nodo está pensado para que se ejecute de forma centralizada y se comunique con cada uno de los UAVs disponibles. Se encarga principalmente de organizar el reparto de tareas, pero también tiene otras labores secundarias como son realizar la escucha y seguimiento de las balizas que envían los UAVs a modo de latidos o "*heartbeat*", subscribirse a la información necesaria de los sensores de cada uno de los UAVs, comprobar si la batería de cada UAV es suficiente para completar el plan

completo de tareas asignado y atender a la información que le envía el nodo "*Agent*"con información sobre el estado del BT.

En cuanto al planificador como tal, para problemas de este tipo se pueden aplicar técnicas de planificación con incertidumbres, ya que los UAVs tienen que razonar ante situaciones inesperadas. Estos planificadores son computacionalmente complejos, ya que tienen que razonar sobre todas las posibles acciones y estados posibles con sus probabilidades. Existen planificadores en línea (online) que intentan paliar esa complejidad tomando el estado actual del UAV y construyendo un árbol con posibles acciones y resultados futuros. Por lo tanto, sólo se evalúan los planes más probables. Existen planificadores que utilizan simulaciones de Monte-Carlo paralelizadas para poder abordar problemas complejos de manera eficiente [8][9]. Cuando hay que planificar para múltiples UAVs, también hay algoritmos descentralizados que pueden planificar en línea logrando la cooperación multi-UAV [10][11][12][13]. Se proponen dos enfoques a comparar para resolver el problema de planificación planteado:

Enfoque 1: Máquina de estados tradicional. Cada vez que aparece una nueva tarea se asigna teniendo en cuenta las capacidades y la batería de cada UAV. Este enfoque tiene la desventaja de que puede comportarse mal cuando haya incertidumbres. Además, la máquina de estados necesaria para resolver la planificación puede ser compleja.

Enfoque 2: Planificador en línea que razone con incertidumbres, partiendo de los mencionados del estado del arte. Las acciones de cada UAV serían las tareas de alto nivel. El estado de cada UAV incluiría su batería y posición, y por otra parte se tendría una lista con las tareas y sus especificaciones. Aplicando modelos de probabilidad, el planificador podría razonar incluso sobre la intención futura de los operarios y responder mejor ante cambios de especificaciones, ángulos y distancias de formación o la entrada de nuevas tareas.

Decir que por el momento, el planificador de tareas implementado asigna las tareas por orden tal y como van llegando. Queda pendiente sustituir este planificador por uno con las características que se han comentado, mientras tanto, con este planificador se pueden ejecutar simulaciones con el objetivo de probar el buen funcionamiento del árbol de comportamiento y determinar si su diseño es correcto o necesita ser cambiado.

Nodo Agent

El nodo "*Agent*" se ejecuta sobre cada UAV, siendo su función principal la de ejecutar el árbol de comportamiento. En el código que implementa este nodo se encuentran definidos todos los nodos del BT, el código interno de cada uno de ellos. Además, este nodo se encarga de subscribirse a la información necesaria de los UAV, de mandar las balizas para comunicar al nodo "*Planner*" que se ha conectado, o que sigue conectado, de recibir balizas a su vez de este nodo para saber de esta forma cuándo él mismo se ha desconectado, de recibir la lista de tareas que asigna el nodo "*Planner*" al UAV sobre el cual se está ejecutando este nodo, y de llamar a los controladores de bajo nivel.

Decir que hasta el momento se han implementado completamente una parte de los nodos, así como versiones falseadoras de los módulos de bajo nivel para simular como que se les ha llamado durante las pruebas, ya que no se dispone de las implementaciones reales para probarlo todo de forma real. Además, realizar las pruebas con módulos falsos puede acelerar el proceso. En lo que no respecta al BT, falta por implementar la baliza que emite el nodo "P*lanner*", sus escucha y temporización desde

el nodo “*Agent*”, y temporizar la llegada del resto de balizas en el nodo “*Planner*” para determinar la desconexión de los UAVs.

Nodo falseador de batería

Otra de las labores llevadas a cabo, esta de forma reciente, ha sido la de la elaboración de un nodo que simule ser la batería del UAV y comunique el nivel de batería falso a través de un tópico de ROS llamado “*/mavros/battery_faker*”. Esta implementación ha sido necesaria debido a que ni *MAVROS* ni *UAL* proporcionaban métodos para el control de la batería. Gracias a este nodo se pueden realizar ahora simulaciones en las que se controle la velocidad de carga y descarga de la batería, se fije la batería a un valor determinado y se controle en qué momentos se carga y se descarga la batería. Este nodo se está empleando en las simulaciones para probar el funcionamiento de la solución planteada. Concretamente, es de vital importancia asegurar que el BT detecta el estado de la batería correctamente y actúa en consecuencia, deteniendo la ejecución de la tarea actual de ser necesario y dirigiendo al UAV hacia la estación de recarga más cercana.

Acompañando al nodo falseador de batería se ha creado un “*Action*” de ROS para controlar los modos de funcionamiento de este nodo, el nivel de la batería y las velocidades de carga y descarga de la misma.

Nivel de batería

Se ha diseñado el sistema de forma que haya dos comprobaciones de batería suficiente diferentes:

- Batería suficiente para completar la tarea actual: esta condición se calcula dentro del nodo “*Agen*t”, y en caso de que su resultado dicte que no se dispone de batería suficiente para completar la tarea actual, se vaciará la cola de tareas, y se comunicará el resultado al nodo “*Planner*” para que actúe en consecuencia y decida qué hacer con esas tareas. En este momento se ejecutará una nueva planificación de tareas. La razón por la que esta condición se calcula en el nodo “*Agente*”, aún a sabiendas de que puede tener un coste computacional no despreciable, es que el UAV debe ser seguro ante todo tipo de situaciones, y esto incluye la posible desconexión del nodo central. Condiciones tan importantes como que exista o no batería suficiente no pueden ser calculadas y comunicadas de forma centralizada porque comprometería la robustez del sistema.
- Batería insuficiente para cumplir la lista completa de tareas: esta condición se calcula en el nodo “*Planner*” para cada agente y decide si ha de re-planificar o no en caso de que no la haya. Al contrario que el anterior, este cálculo se lleva a cabo de forma centralizada porque busca anticiparse a imprevistos futuros, pero que no es necesario para asegurar la seguridad y la robustez del sistema. Además, este cálculo tendrá un coste computacional superior al del anterior.

Análisis de escenario: todos los “*Agent*” sin batería.

Ante una situación como la planteada, debe existir algo en el algoritmo de asignación de tareas definitivo que haga al código no bloquearse al no encontrar ningún UAV al que asignar las tareas. Esto lleva a pensar soluciones en las que no se ignore a los UAVs que estén sin batería durante

repartos en los que hay presentes UAVs aún con batería, sino que por el contrario las tareas se repartan teniendo en cuenta qué UAVs tienen batería y cuales no, decidiendo si es necesario detener una recarga a la mitad con el fin de optimizar el reparto y la ejecución de las tareas, y decidiendo si se les asignan tareas a UAVs que no tengan aún un nivel de batería mínimo a sabiendas de que terminarán de recargar en un tiempo X y que por tanto la tarea será ejecutada entonces.

Re-asignación de tareas

Un breve listado de las situaciones en las que se realizarían a priori re-planificaciones de tarea:

- Llegada de una nueva tarea.
- Conexión de un nuevo UAV.
- Desconexión de un UAV.
- Comunicación de batería insuficiente por parte de alguno de los UAV.
- Cálculo de que algún UAV no tiene batería suficiente para el plan completo de tareas.
- El nodo "*Planner*" detecta que algún UAV en recarga tiene ya batería suficiente para una tarea que conviene ejecutar ya.
- El nodo "*Agent*" comunica que ha terminado de recargar su batería.
- El nodo "*Agent*" comunica que una tarea ha terminado con éxito.
- El nodo "*Agent*" comunica que una tarea ha terminado con fracaso.

A priori, se tendrán en cuenta algunas restricciones de a la hora de realizar una re-asignación de tareas, aunque esas consideraciones podrían variar cuando se implemente al solución definitiva. Por ejemplo, en lugar de prohibir esas acciones, se podría asignar un coste alto a ellas, de forma que el planificador no las tenga prohibidas, pero que solo las lleve a cabo cuando sea realmente óptimo según sus cálculos. Las restricciones de las que se habla serían:

- No desalojar tareas que sean de máxima prioridad.
- No desalojar tareas que sean de igual o mayor prioridad que la tarea que la sustituiría.
- Definir qué tipo de tareas pueden ser desalojadas y cuáles no.

Pérdida de conexión

Se realizará ahora un análisis de diferentes escenarios en los que podría ocurrir una pérdida de conexión, comentando cómo se prevé que reaccione en cada caso la arquitectura planteada:

- Desconexión sin batería suficiente para la tarea actual: si da la casualidad de que la conexión se pierde al momento en que se comprueba y comunica si habrá batería suficiente para terminar la tarea actual o no, el nodo "*Planner*" nunca recibiría la información de que el UAV se ha detenido para recargar y sus tareas han sido des-asignadas.
    - Nodo "*Agent*": En este caso su funcionamiento sería a prueba de fallos, ya que la condición de batería suficiente se calcula en este nodo y se auto vacía la cola de tareas para provocar que el BT detecte al UAV en estado ocioso y lo dirija hacia la estación de

recarga, en este caso actuando punto de seguridad. En este caso, el UAV simplemente se quedaría recargando hasta que se recupere la conexión y se le asignen nuevas tareas.

  - Nodo "*Planner*": por su parte, el nodo "*Planner"* se enteraría de que el UAV se ha desconectado por uno de los siguiente medios: tiempo máximo sin recibir balizas o "Action Server" no detectado en alguno de los canales de comunicación existentes entre "*Planner*" y "*Agent*". Una vez notado que el UAV se ha desconectado, se asume que por seguridad, este ha vaciado su cola de tareas y ha aterrizado, por lo que se reasignan sus tareas para que sean completadas por otro UAV.

- Desconexión con batería suficiente para cumplir la tarea actual o el plan completo: en este caso, ha de existir un protocolo de actuación para que las tareas no queden sin hacer ni se realicen por duplicado. El nodo "*Planner*" detectaría la pérdida de conexión a través de alguno de los dos medios mencionados anteriormente (balizas, "*Action Server*"). El nodo"*Agent"* detectaría la pérdida de conexión a su vez a través de las balizas que emite con este fin el nodo central.

  - Se podría establecer entonces que, en caso de pérdida de conexión, el UAV vacíe su lista de tareas y el nodo "*Planner*" asuma que el UAV ha vaciado o va a vaciar su lista de tareas y que, por tanto, ejecute una re-asignación de tareas.

  - En caso de que la tarea sea de prioridad máxima, aunque se disponga de batería suficiente, por seguridad, lo mejor es igualmente vaciar la lista de tareas y pasar al agente a un estado de espera o de recarga.

Estado actual del trabajo

En este punto del trabajo hace tiempo que se han comenzado ya las pruebas de simulación. Para ello se tienen varios archivos "*.launch*" que configuran diferentes simulaciones según lo que se desee probar. Recientemente se ha estado organizando de nuevo estos archivos para mejorarlos y hacer que sean más sencillos de modificar y emplear.

Como ya se ha comentado, algunas partes importantes del código como el planificador de tareas, los cálculo de batería suficiente, o algunos nodos del BT, se encuentran en una versión simplificada con el objetivo de que permitan probar otras partes del código manteniendo su estructura final.

Respecto a la librería para BT de la cual anteriormente se ha hablado [3], comentar que no está tan bien documentada como parecía en un principio, ya que los ejemplos están redactados de forma muy simplificada y surgen muchas dudas de implementación a la hora de integrar el BT con una aplicación de ROS real. Dato que tampoco hay una comunidad en la que comentar problemas, su implementación está dando más problemas de los inicialmente previstos y está llevando también más tiempo del estimado. Además, parece que algunos usuarios han reportado, a través de Github, ciertos bugs que parecen encajar con algunos problemas detectados en las simulaciones y que afectarían al funcionamiento de la arquitectura. Aunque el mantenimiento del repositorio sigue activo y su última actualización es relativamente reciente, estos bugs reportados permanecen sin respuesta, por lo que aún no se descarta que en futuras pruebas se determine que no es un problema de la librería sino del uso que le están dando estos usuarios, ya que como se ha dicho, en la documentación faltan puntos importantes que solo aparecen a la hora de implementar nodos más

complejos. En caso de que realmente sea un bug, tendrá que ser solucionado manualmente o bien realizar un nuevo cambio de librería, ya que no se prevé que estos sean arreglados a corto plazo.

Por último, se estaba trabajando en un problema encontrado en el paso de información entre nodos del BT. Para permitir que los BT sean fácilmente re-utilizables y escalables entre otras cosas, durante las fases de diseño de la librería, sus creadores decidieron que los nodos no podían permitir el paso de parámetros al ser llamados [14]. Sin embargo, sabían que era necesario un método para compartir información entre nodos dado que para sus aplicaciones en robótica muchos nodos necesitarían de datos de entrada para su ejecución y su mejor reutilización dentro del código. La solución a la que llegaron durante las fases de diseño de la librería fue la implementación de "blackboards", que son algo así como variables globales, y como tal, no es recomendable su uso; y la implementación de puertos, que conectan un nodo con otro y pueden ser de entrada y salida. Estos puertos permiten únicamente cadenas de texto, por lo que para compartir otro tipo de información es necesario la definición de métodos de conversión de clases propias a cadenas de texto y viceversa. Como solución para el problema de la información, se decidió compartir la dirección del puntero a la clase que contiene tanto al BT como toda la información de las tareas y los sensores del UAV, de forma que cada nodo del BT disponga simultáneamente de toda la información y pueda leer la que necesite. En las últimas pruebas realizadas se detectó funcionamientos extraños en el BT, funcionando correctamente unas veces y otras no. Esto se cree que es debido a que el puntero cambia con el tiempo. El problema es que al no poder transmitir información a los nodos más que durante su inicialización, el puntero a la clase exterior no puede ser actualizado. Las próximas horas de trabajo se destinarán a estudiar si es viable actualizar este puntero en cada ejecución a través de los puertos mencionados, o si por el contrario han de comenzar a definirse métodos de conversión de datos a cadenas de texto. Serán necesarias simulaciones en cualquiera de los dos casos.

Cuando esto esté listo, el BT ya estará prácticamente terminado, quedando únicamente las mejoras necesarias en aquellos nodos en los que se ha programado una versión simplificada para permitir la realización de pruebas en el resto de nodos. En este punto ya el flujo de trabajo será mucho más dinámico, ya que implementar mejoras es mucho más rápido y sencillo que construir toda una arquitectura desde cero y con la necesidad de aprender sobre el lenguaje de programación y el uso de extensas librerías como se ha venido haciendo desde el comienzo de la actividad descrita.

Referencias

[1] Wim (9 de agosto de 2010). ROS Wiki. smach/Tutorials/StateMachine container. http://wiki.ros.org/smach/Tutorials/StateMachine%20container

[2] Colledanchise, M., Sarantopoulos, I., Abou, A., & Hageman, R. (22 de octubre de 2018). Github. miccol /ROS-Behavior-Tree. https://github.com/miccol/ROS-Behavior-Tree

[3] Faconti, D. (28 de junio de 2021). BehaviorTree.Cpp. https://www.behaviortree.dev/

[4] Faconti, D. et al. (2 de mayo de 2021). Github. BehaviorTree/Groot. https://github.com/BehaviorTree/Groot

[5] Colledanchise, M., & Ögren, P. (2016). How behavior trees modularize hybrid control systems and generalize sequential behavior compositions, the subsumption architecture, and decision trees. IEEE Transactions on robotics, 33(2), 372-389.

[6] Colledanchise, M., & Ögren, P. (2018). Behavior trees in robotics and AI: An introduction. CRC Press.

[7] Simpson, C. (17 de julio de 2014). GAMASUTRA. Behavior trees for AI: How they work. https://www.gamasutra.com/blogs/ChrisSimpson/20140717/221339/Behavior_trees_for_AI_How_they_work.php

[8] Garg, N., Hsu, D. and Lee, W. S. (2019) DESPOT-α: Online POMDP Planning With Large State And Observation Spaces. Robotics: Science and Systems (RSS).

[9] Cai, P., Luo, Y., Hsu, D. and Lee, W. S. (2018) HyP-DESPOT: A Hybrid Parallel Algorithm for Online Planning under Uncertainty. Arxiv.

[10] Best, G., Cliff, O. M., Patten, T., Mettu, R. R., & Fitch, R. (2019). Dec-MCTS: Decentralized planning for multi-robot active perception. The International Journal of Robotics Research , 38 (2–3), 316–337.

[11] S. Chen, F. Wu, L. Shen, J. Chen and S. D. Ramchurn, "Decentralized Patrolling Under Constraints in Dynamic Environments," in IEEE Transactions on Cybernetics , vol. 46, no. 12, pp. 3364-3376, Dec. 2016.

[12] J. Renoux, A. Mouaddib and S. L. Gloannec, "A decision-theoretic planning approach for multi-robot exploration and event search," 2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Hamburg, 2015, pp. 5287-5293.

[13] Omidshafiei, Shayegan, Ali–Akbar Agha–Mohammadi, Christopher Amato, Shih–Yuan Liu, Jonathan P How, and John Vian. “Decentralized Control of Multi-Robot Partially Observable Markov Decision Processes Using Belief Space Macro-Actions.” The International Journal of Robotics Research, 36, no. 2 (2017): 231–58.

[14] Faconti, D. (28 de junio de 2021). BehaviorTree.Cpp. Final report https://raw.githubusercontent.com/BehaviorTree/BehaviorTree.CPP/master/MOOD2Be_final_report.pdf