UNIVERSIDAD DE SANTIAGO DE CHILE

FACULTAD DE INGENIERÍA

Departamento de ingeniería informática

PROTOCOLO DE NAVEGACIÓN ENTRE UAVS, BAJO ARQUITECTURA DE COMUNICACIÓN ROS, PARA EXPLORACIONES DE RECONOCIMIENTO

RICARDO FABIÁN ZÚÑIGA ANTIÑIRRE

Profesor Guía: Arturo Álvarez Cea

Trabajo de titulación en conformidad a los requisitos para obtener el título de Ingeniero Civil en Informática

SANTIAGO - CHILE

2016

RESUMEN

El término UAV (*Unmanned Aerial Vehicle*), son las siglas en inglés para referirse a un vehículo aéreo no tripulado. Un UAV, es un sistema que agrupa un conjunto de subsistemas, que apoyan a la aeronave, para funcionar sin tripulación a bordo.

Los UAVS, se utilizan para distintos tipos de misiones, civiles, militares y de exploración. Un nuevo ámbito, que se empieza a estudiar, es el uso de UAVs para fines de exploración, donde las circunstancias pueden ser de difícil acceso, o en situaciones de condición climáticas extremas, en la cual las máquinas, pueden ser de gran utilidad en el sentido de obtener información necesaria para evitar potenciales riesgos de accidentes de personal humano.

El presente proyecto, ofrece un protocolo de exploración entre UAVs, que contribuye al estudio de nuevas arquitecturas para trabajo con máquinas.

PALABRAS CLAVES: UAV, ROS, ROSCORE, SOCKET, PROTOCOLO, PARROT.

TABLA DE CONTENIDO

ÍNDICE DE TABLAS

ÍNDICE DE ILUSTRACIONES

CAPÍTULO 1: INTRODUCCIÓN

El propósito de este capítulo es describir, en términos generales, el planteamiento del proyecto. Por otra parte, se examinan los antecedentes y motivaciones asociadas a su desarrollo, con el objeto de presentar la problemática y así proponer una solución.

1.1. ANTECEDENTES Y MOTIVACIÓN

El término UAV (*Unmanned Aerial Vehicle*), son las siglas en inglés para referirse a un vehículo aéreo no tripulado. Un UAV, es un sistema que agrupa un conjunto de subsistemas, que apoyan a la aeronave. Dentro de los subsistemas, se encuentran: control automático de vuelo, módulos de interoperabilidad, sistemas de recuperación de fallos, sistemas de despegue, suspensión, y a su vez un conjunto de cargas útiles, que se acomodan a la arquitectura del diseño, para operar sobre distintos escenarios (Austin, 2010).

Los UAVs pueden ser utilizados para ámbitos civiles, como por ejemplo, obtener información topográfica de alta densidad, control y gestión de cauces hídricos, registros de magnetometría en alta resolución para faenas mineras, mapas de calor en agricultura para visualizar el uso de las aguas, brotes de enfermedades y estadísticas de salud de plantes, también para ámbitos militares, como vigilancia de actividad hostil, retransmisión de señales de radio e inteligencia electrónica.

Un nuevo ámbito, que se empieza a estudiar, es el uso de UAVs para fines de exploración (Flyability, 2014), donde las condiciones estructurales son de difícil acceso, o con situaciones climáticas extremas, en la cual las máquinas pueden ser de gran utilidad para obtener información relevante, como por ejemplo, registro de niveles de radiación en un desastre nuclear, de esta forma se evita el riesgo potencial de accidentes o lesiones de una exploración tradicional con personal humano. Por otro lado, dada las características de una exploración donde no existe control manual de las máquinas y el éxito radica en retornar con información relevante ante cualquier tipo de fallos, es necesario el envío de más de un explorador, para reducir el riesgo de fracaso. Luego, para poder lograr este objetivo, es necesario establecer un protocolo de navegación que asegure que los UAVs exploradores, establezcan un protocolo secundario de retorno ante cualquier tipo de fallos.

Los UAVs, por lo general, poseen los mismos elementos que los sistemas basados en aviones tripulados, con la diferencia, que la tripulación aérea, es considerada un subsistema dentro de las interfaces de control electrónico (Austin, 2010).

Los UAVs, son desarrollados para poseer ciertos grados de inteligencia autónoma, tales como, la posibilidad de comunicarse con las estaciones de control, e informar sobre los datos de sus cargas útiles. También tienen la capacidad de transmitir información importante en relación a su condición actual, autonomía restante, temperatura de sus componentes, altitud y velocidad. Si se produce algún tipo de fallo en cualquiera de los subsistemas o componentes, el UAV puede ser diseñado para tomar medidas correctivas, como también, alertar a su operador en tierra. Estas aeronaves pueden ser de gran utilidad, gracias a su pequeño tamaño, sus capacidades para capturar vídeo y ser controlados de forma remota (Austin, 2010)

Para que un UAV pueda comportarse como tal, los sistemas mínimos que debe tener son los siguientes:

- Placa controladora: consiste en una placa que contiene un microcomputador, que incluye instrucciones que debe seguir el UAV. La placa está instalada a bordo, y en ella van conectados los diferentes sensores externos, como el GPS, el sensor de tensión/corriente, el sistema de telemetría y los puertos de comunicación que permiten comandar la máquina. Existen otros sensores, como el magnetómetro, barómetro y acelerómetro.(Austin, 2010)
- Sistema GPS: un sistema autónomo que puede ubicar la posición, la altura y la velocidad del vehículo y transmitir los datos en forma serial.
- Barómetro digital: un sensor barométrico de estado sólido que se utiliza como altímetro.
- Sensor de tensión y corriente: un sistema que mide la tensión de la batería, tanto en los motores como en los sistemas electrónicos.
- Sistema de telemetría: consiste en un sistema de transmisión de datos bidireccional, que permite transmitir a tierra los datos de posición, altura, velocidad, estado de la batería.

Tal es el caso de Elios (Flyability, 2014), el primer UAV de tipo explorador, diseñado para profesionales de inspección industrial. Su diseño, permite por primera vez el acceso a lugares complejos en una serie de aplicaciones, donde su utilización puede ser de gran ayuda,

especialmente para evitar misiones que revistan un grado de peligro, o simplemente imposibles de realizar (Flyability, 2014).

El diseño de este UAV, es de tipo jaula protectora, lo que permite, por ejemplo, el acceso remoto a las calderas, tanques, recipientes a presión, túneles y otros entornos complejos dentro de una planta industrial. Dada las características de su diseño, es seguro volar cerca e incluso en contacto con los seres humanos y el medio ambiente circundante. Por lo tanto, se puede utilizar cuando la planta está todavía en funcionamiento, sin ningún tipo de riesgo de accidente o lesión.

Por otra parte, además del enfoque en el aspecto técnico industrial, una nueva característica en el estudio de los UAVs se orienta a fomentar las capacidades de las máquinas para exploraciones de reconocimiento, con más de una máquina de exploración, sobre misiones donde no exista un control manual y se haga necesario la autonomía y algún grado de inteligencia en la toma de decisiones. También es imprescindible retornar con la información, con el objetivo de aplicar protocolos secundarios en caso de contingencia. Para la aplicación de estas características, se hace necesario crear un protocolo de navegación entre UAVs, utilizando mapas georreferenciados, además de crear la arquitectura de comunicación entre las máquinas utilizando alguna plataforma de paso de mensajes y un cliente de comandos en tierra, para ejecutar la logística de la misión.

1.2. DESCRIPCIÓN DEL PROBLEMA

Dada una misión de exploración de reconocimiento, donde participa más de un UAV, se busca asegurar el retorno de información relevante, perteneciente a las cargas útiles, de al menos una máquina de exploración, para esto es necesario utilizar protocolos de navegación a través de georreferenciación, bajo una arquitectura definida y un cliente de comandos, con el fin de aplicar protocolos secundarios en caso de contingencias y fallos, con el objetivo de evitar misiones de exploración con personal humano.

1.3. SOLUCIÓN PROPUESTA

1.3.1. Características de la solución

Se contempla el diseño de una arquitectura de comunicación cliente-servidor, basada en la plataforma ROS Fuerte (*Robotic Operative System*, 2012) bajo sistema operativo Linux Ubuntu 12.04. Esta plataforma proporciona una serie de herramientas y bibliotecas para desarrollo de aplicaciones en robótica, que permiten el paso de mensajes entre una estación cliente, el servidor y la máquina que ejecutará las órdenes.

La arquitectura de comunicación, poseerá dentro de sus características, un protocolo de navegación entre UAVs, que permitirá el despliegue de la navegación para una misión específica, a través de un GPS de tipo *Fligth Recorder*, donde se marcarán los puntos de exploración o *waypoints*.

El paso de mensajes entre el cliente y el servidor se realizará a través de un programa de comandos de tipo *socket stream*, bajo sistema operativo Linux Ubuntu 12.04.

1.3.2. Propósito de la solución

El uso de UAVs para exploraciones de reconocimiento, en zonas de difícil acceso o con condiciones climáticas adversas, es de gran ayuda, para evitar riesgos potenciales y accidentes de personas. Desde esta perspectiva, las máquinas cumplen un papel fundamental, puesto que con el avance de la tecnología y comunicaciones, es posible reemplazar en gran parte la capacidad humana para misiones de alto riesgo.

Por lo tanto, el propósito de la solución se encuentra orientado bajo dos enfoques, el primero es apoyar el estudio y análisis de nuevas arquitecturas basadas en protocolos de navegación autónomos, utilizando plataformas de comunicación de paso de mensajes basadas en ROS, el segundo enfoque, está dirigido a establecer un protocolo secundario de retorno de UAVs, que asegure el éxito de una misión en caso de fallos, con el objetivo de evitar riesgos de exploración con personal humano.

1.4. OBJETIVOS Y ALCANCES DEL PROYECTO

1.4.1. Objetivo general

Crear un protocolo de navegación entre vehículos aéreos no tripulados (UAVs, *Unmanned Aerial Vehicle*), para exploraciones de reconocimiento utilizando georreferenciación, bajo arquitectura de comunicación ROS *(Robotic Operative System*, 2012) y cliente de comandos *socket stream*.

1.4.2. Objetivos específicos

1. Definir características para exploraciones de reconocimiento.
2. Crear protocolo de navegación para UAVs, utilizando georreferenciación, con características propias de exploraciones reconocimiento.
3. Crear servidor de paso de mensajes utilizando plataforma ROS.
4. Crear arquitectura de comunicación entre UAVs Parrot AR.DRONE 2.0 y plataforma de comunicación ROS.
5. Crear cliente de comandos *socket stream*.
6. Validar protocolo de navegación, mediante pruebas de simulación y en terreno para vuelos de exploración.

1.4.3. Alcances

En relación a lo establecido anteriormente, se tiene en cuenta los siguientes alcances y limitaciones:

- Se contempla el uso de la plataforma ROS, como base el desarrollo de la arquitectura servidora de paso de mensajes y protocolos de comunicación.
- Se contempla el uso de *socket stream*, programados en lenguaje C, como cliente de comandos.
- La solución se encuentra sujeta a la arquitectura de un UAV de tipo Parrot AR.Drone 2.0, por tanto, la arquitectura y protocolos de comunicación serán evaluados sobre este tipo máquinas.
- Las condiciones operativas de vuelo del UAV Parrot AR.Drone 2.0, serán de tipo estructurado, con un espacio definido para la realización de las pruebas.

- Para estudiar el protocolo de navegación, y la arquitectura de paso de mensajes entre las máquinas y el servidor, se georreferenciarán diferentes tipos de misiones.

1.5. METODOLOGÍAS Y HERRAMIENTAS UTILIZADAS

El desarrollo del protocolo de navegación entre UAVs, la integración de dispositivos complementarios y el cliente de comando de paso de mensajes, implica gran cantidad de evaluaciones en terreno, para comprobar si la arquitectura implementada funciona de forma correcta según los objetivos planteados. Para apoyar el proceso de desarrollo, se ha decidido utilizar una metodología ágil, debido a la flexibilidad que aportan en su desarrollo, entre los tipos de metodologías existente, se optó por AUP (*Agile Unified Process*) en vez de otras como *Scrum*, cuyos beneficios destacan más en la implementación de *software* y la gestión de tareas del equipo de trabajo. AUP es una metodología que simplifica el desarrollo dirigido por test y la gestión de cambios, lo cual en conjunto con el carácter iterativo la hacen apropiada para esta memoria. Las etapas de esta metodología son básicamente (Laboratorio Nacional de Calidad del Software de INTECO, 2009):

- Concepción: Se definen los alcances y potenciales arquitecturas del sistema.
    - Se utilizará un UAV de prueba de tipo Parrot 2.0, bajo plataforma ROS y cliente de comandos *socket stream.*
- Elaboración: Se analizan los requisitos del sistema y se valida la arquitectura.
    - Se realizan pruebas funcionales sobre el UAV y se valida la arquitectura a través de comandos de pruebas y establecimiento de comunicación bidireccional con GPS.
- Construcción: Se desarrolla el sistema y se prueba en el ambiente de desarrollo.
    - Se establece arquitectura base bajo plataforma ROS, se crean clientes *socket* y se realizan pruebas unitarias en ambiente de test. En esta parte se desarrollará el protocolo de comunicación, y el paso de mensajes entre la plataforma y el UAV.
- Transición: Se prueba el sistema para validarlo antes de enviarlo a producción.
    - En esta parte, se realizarán pruebas en terreno, para validar la acción de los protocolos de retorno establecidos.

Es importante mencionar que en el marco central del trabajo, se utilizará en gran medida esta metodología, sin embargo, no se descarta modificarla o utilizar otras técnicas que apoyen el

proceso de investigación y desarrollo, como por ejemplo, utilizar un mecanismo híbrido entre la metodología AUP y el método científico, ya que el proyecto requiere para su desarrollo de investigación, experimentación y validación de resultados. Luego las etapas del método científico:

- Observación: Análisis del fenómeno.
- Hipótesis:
    - "Se considera que el establecimiento de protocolos primarios y secundarios de retorno asegura el éxito de una misión de exploración".
- Experimentación: Verificación de las hipótesis
- Predicción: Hipótesis con mayores probabilidades.
- Conclusiones: Hipótesis demostradas

Luego las etapas del método científico, se incluirían como una sub-etapa dentro del ítem de construcción:

- Concepción
- Elaboración
- Construcción
    - Observación
    - Hipótesis
    - Experimentación
    - Conclusiones
- Transición

1.5.1. Herramientas de desarrollo

Las herramientas a utilizar (*hardware* y *software*) para el desarrollo del proyecto son las siguientes:

- UAV Parrot AR.Drone 2.0:
    - Peso: 436 grm.
    - Dimensiones: 45 * 29 cm.
    - Procesador ARM Cortex A8 de 32 bits a 1 GHz.
    - Linux 2.6.32.
    - RAM DDR2 de 1 GB.
    - USB 2.0 de alta velocidad.
    - Giroscopio de 3 ejes.

    - Acelerómetro de 3 ejes +/-50 mg.
    - Magnetómetro de 3 ejes, precisión 6°.
    - Sensor de presión +/- 10 PA (80 cm a nivel del mar).
    - Sensor de ultrasonidos para medición de altitud respecto al suelo.
    - Cámara QVGA vertical a 60 FPS para medición de la velocidad respecto al suelo.
    - Batería recargable Li-Po de 3 elementos, 1.000 MAH.
    - Tiempo de carga 1 hora 30 minutos, autonomía 12 minutos.
    - Cámara frontal HD. 720p 30 FPS, objetivo gran angular: 92°.
    - Cámara vertical diagonal a 64°, sensor CMOS, 60fps, 320*210 QVGA.
- GPS *Fligth Recorder.*
- Estación de trabajo:
    - Sistema operativo: Linux Ubuntu 12.04 (64 bits).
- Documentación:
    - Open Office 2014.
    - LaTeX.
- Desarrollo:
    - Lenguaje de programación Python.
    - Lenguaje de programación C.
- GNU *Compiler Collection* (GCC).
- Editor VIM.
- Control de versiones SVN.

1.6. ORGANIZACIÓN DEL DOCUMENTO

El presente trabajo está dividido en cinco capítulos considerando éste como el primero. En el Capítulo 2, se formalizan los fundamentos de los UAVS y diferentes protocolos para misiones de exploración y reconocimiento, también se analiza con la teoría de sobre diferentes tipos de UAVs, su mecánica relacionada, tipos de drones de exploración, protocolos de navegación y enlace de datos, la descripción de la plataforma ROS y clientes de comunicación a través de socket.

En el Capítulo 3 se diseña el protocolo de navegación propuesto, en el Capítulo 4, se desarrolla la arquitectura de comunicación ROS, en el Capítulo 5, se diseña de la estación cliente *socket*, y en el Capítulo 6, se analizan las conclusiones.

CAPÍTULO 2: MARCO TEÓRICO

En este capítulo se presentan los conceptos utilizados para el desarrollo de esta memoria, permitiendo comprender, de mejor manera, el contexto del tema propuesto. Se presenta a modo general temas tales como la arquitectura de los UAVs, tipos de drones de exploración, protocolos de navegación, protocolos de enlace de datos, plataforma ROS (Robot Operating System) y tipos de *socket.*

2.1. UAVs

Los UAVs, por lo general, poseen los mismos elementos que los sistemas basados en aviones tripulados, con la diferencia, que la tripulación aérea, es considerada un subsistema dentro de las interfaces de control electrónico (Austin, 2010).

Los UAVs, son desarrollados para poseer ciertos grados de inteligencia autónoma, tales como, la posibilidad de comunicarse con las estaciones de control, e informar sobre los datos de sus cargas útiles. También tienen la capacidad de transmitir información importante en relación a su condición actual, autonomía restante, temperatura de sus componentes, altitud y velocidad. Si se produce algún tipo de fallo en cualquiera de los subsistemas o componentes, el UAV puede ser diseñado para tomar medidas correctivas, como también, alertar a su operador en tierra. Estas aeronaves pueden ser de gran utilidad, gracias a su pequeño tamaño, sus capacidades para capturar vídeo y ser controlados de forma remota.

Tal es el caso de Elios (Flyability, 2014), el primer UAVs de tipo explorador, diseñado para profesionales de inspección industrial. Su diseño, permite por primera vez el acceso a lugares complejos, en una serie de aplicaciones, donde su utilización puede ser de gran ayuda, para evitar misiones que revistan un grado de peligro, o simplemente imposibles de realizar.

Además del enfoque en el aspecto industrial, una nueva característica sería orientar las capacidades de las máquinas, a exploraciones de reconocimiento, con más de una máquina de exploración, misiones donde no exista un control manual y se haga necesario la autonomía y algún grado de inteligencia en la toma de decisiones. También es imprescindible, retornar con la información, con el objetivo de aplicar protocolos secundarios en caso de contingencia. Para la aplicación de estas características, se hace necesario crear un protocolo de navegación entre UAVs, utilizando mapas georreferenciados, además de crear la arquitectura de comunicación

entre las máquinas, utilizando alguna plataforma de paso de mensajes y un cliente de comandos en tierra, para ejecutar la logística de la misión.

2.1.2. Quadrotores

A continuación se describe la estructura mecánica del UAV AR.Drone 2.0, tomando como referencia la guía del desarrollador creada por Stephane Piskorski (Piskorski, 2012), dentro de los puntos se trata:

- Estructura mecánica quadrotores.
- Parrot AR.Drone 2.0.
- Motores.
- Baterías.
- Sensores de movimiento.
- Video *streaming*, etiquetas y detección de *roundel*.
- Conexión y red Wifi.
- Servicios de comunicación entre el AR.Drone 2.0 y un dispositivo. cliente.
- Cargas útiles, *payloads*.

La estructura mecánica comprende cuatro rotores, unidos a los cuatro extremos de un cruce al que se unen la batería y el *hardware* RF. Cada par de rotores opuestos gira en la misma dirección. Un par gira en el sentido de las agujas del reloj, los otros en el sentido opuesto (Figura 2-1).

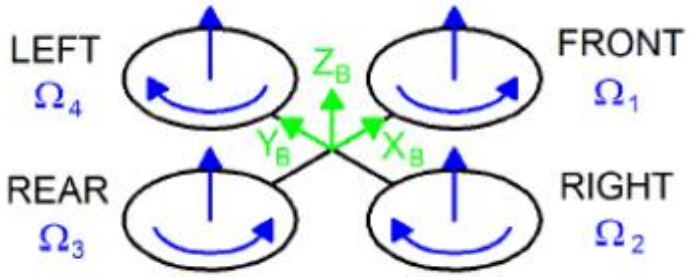

Figura 2-1. Estructura mecánica quadrotor AR. Drone 2.0. Extraído de Piskorski (2012).

En la Figura 2.2, se puede apreciar con mayor detalle los movimientos de los cuatro rotores.

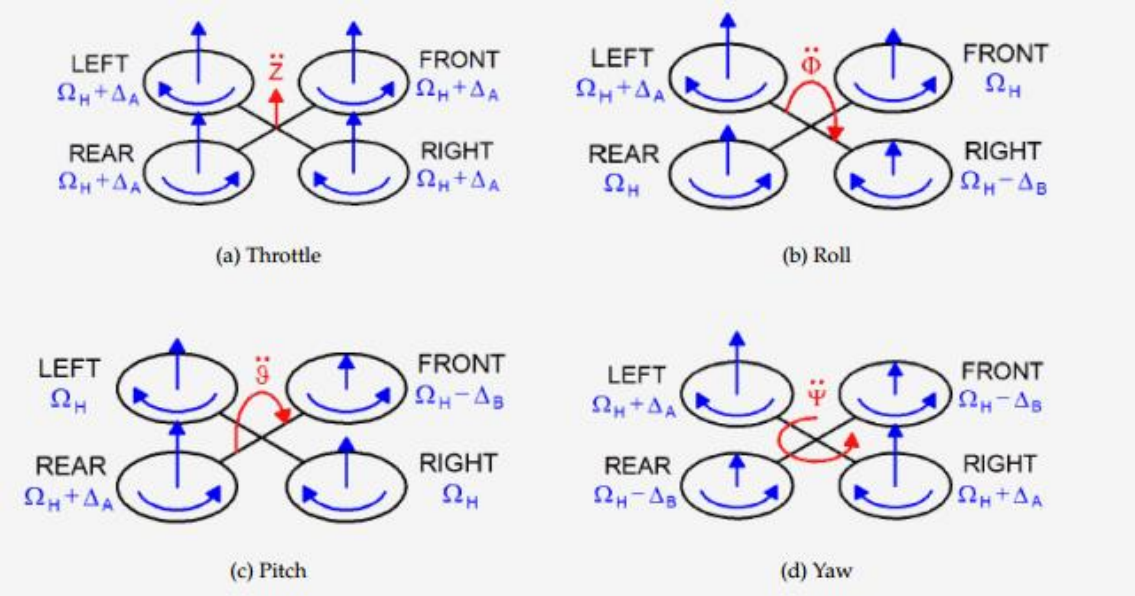

Figura 2-2. Movimientos quadrotor AR. Drone 2.0. Extraído de Piskorski (2012).

Los tipos de maniobras de un quadrotor, se obtienen cambiando los ángulos de tono (*pitch*), rodillo (*roll*) y guiño (*yaw*) (Figura 2-3).

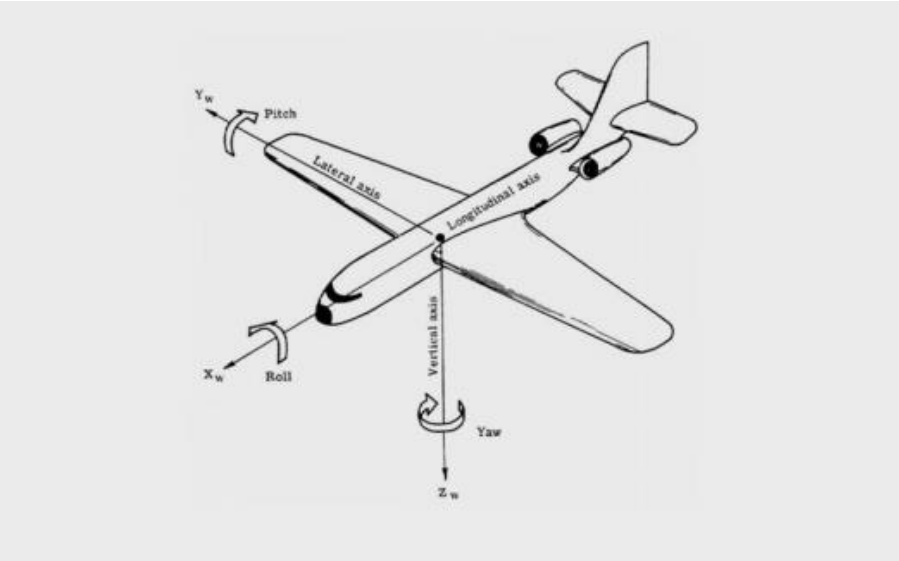

Figura 2-3. Tipos de maniobras quadrotor. Extraído de Piskorski (2012).

La variación de las velocidades de los rotores izquierdo y derecho en la dirección opuesta, produce el movimiento del rodillo. Esto permite ir adelante y atrás. Diferentes velocidades de los rotores, delanteros y traseros de la manera opuesta produce el movimiento de tono. Al variar la velocidad de cada par de rotores, la dirección opuesta produce el movimiento de guiño. Esto permite girar a la izquierda y derecha.

2.1.1. Parrot AR.Drone 2.0

AR.Drone 2.0, es un UAV, diseñado por la empresa francesa Parrot. Funciona propulsado por cuatro motores eléctricos, cuenta con un microprocesador y una serie de sensores, entre los cuales se incluyen dos cámaras, más un conector WIFI integrado que le permite enlazar a dispositivos móviles personales que cuenten con los sistemas operativos iOS, *Android* o *Linux*. Esto permite controlar al quadrotor directamente desde un dispositivo móvil, mientras se reciben las imágenes y datos de telemetría que los sensores del UAV se encuentran recuperando, la Figura 2-4, muestra el diseño del Ar.Drone 2.0 con sus respectivas carcasas para actividades de interior como también de exterior.

Figura 2-4. Parrot AR. Drone 2.0. Extraído de Piskorski (2012).

2.1.2. Motores

El AR.Drone 2.0 está alimentado con motores sin escobillas con tres fases de corriente controlada por un microcontrolador.

El AR.Drone 2.0 detecta el tipo de motores que están conectados y automáticamente ajusta los controles del motor. El AR.Drone 2.0 detecta si todos los motores están girando o están detenidos. En caso de que una hélice giratoria encuentre algún obstáculo, el AR.Drone 2.0 detecta si la hélice está bloqueada, en tal caso detiene todos los motores inmediatamente.

2.1.3. Baterías

El AR.Drone 2.0, utiliza baterías cargadas de 1000 *mAh*, 11.1 *V LiPo* para volar. Al volar el voltaje disminuye de la carga completa (12.5 *voltios*) a la carga baja (9 *voltios*). Los monitores de batería del AR.Drone 2.0, convierten este voltaje en un porcentaje de vida de la batería (100% si la batería está completa, 0% si la batería está baja). Cuando el UAV detecta un voltaje bajo en batería, primero envía una advertencia, luego automáticamente aterriza. Si el voltaje alcanza un nivel crítico, el conjunto, el sistema se apaga para evitar comportamientos inesperados.

2.1.4. Sensores de movimiento

AR.Drone 2.0 tiene sensores de movimiento, que se encuentran bajo el casco central. Los sensores cuentan con medición inercial de 6 DOF, MEMS, esto proporciona mediciones de tono, balanceo y guiño. Las medidas inerciales se usan para la estabilización automática de tono, balanceo, guiñada y control de inclinación. Posee un telémetro de ultrasonido que proporciona medidas de estabilización automática de altitud y control de velocidad vertical asistido. Una cámara que apunta hacia el suelo, la que proporciona medidas de velocidad en tierra. Por último, posee 3 DOF a la IMU, con un magnetómetro de 3 ejes. También añade un sensor de presión para permitir mediciones de altitud a cualquier altura. La Figura 2-5 muestra el trabajo de los sensores que se encuentran bajo el casco del UAV.

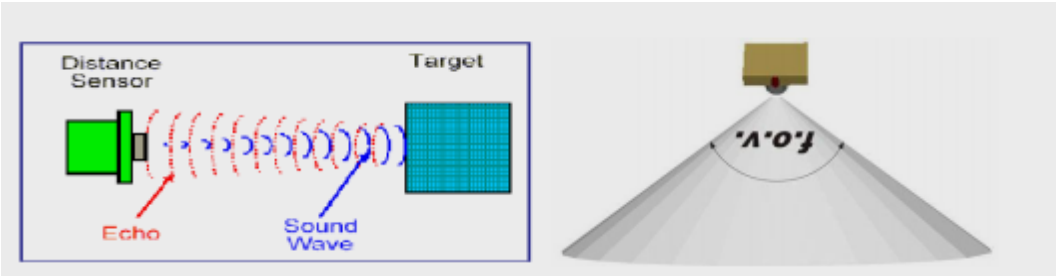

Figura 2-5. Sensores AR. Drone 2.0. Extraído de Piskorski (2012).

2.1.5. Video *streaming*, etiquetas y detección de *roundel*

- AR. Drone 2.0, posee una cámara frontal como sensor CMOS, con una lente de ángulo de 90 grados.

- Automáticamente codifica y transmite las imágenes entrantes al dispositivo host o estación base.

- AR.Drone 2.0 utiliza resolución de imagen de 360p (640x360) o 720p (1280x720) para ambas cámaras. La velocidad de fotogramas del flujo de vídeo se puede ajustar entre 15 y 30 FPS.

2.1.6. Tipo de conexión y red WIFI

AR.Drone 2.0 puede ser controlado desde cualquier dispositivo cliente que soporte WIFI. El proceso es el siguiente:

- AR.Drone crea una red WIFI con un ESSID, y auto asigna una dirección IP libre, impar (normalmente 192.168.1.1).
- El operador conecta el dispositivo cliente a esta red ESSID.
- El dispositivo cliente solicita una dirección IP desde el servidor DHCP del AR. Drone.
- El servidor DHCP de AR.Drone concede al cliente una dirección IP.

2.1.7. Servicios de comunicación entre el AR.Drone 2.0 y un dispositivo cliente

- El control de AR.Drone se realiza a través de tres servicios de comunicación principales.
- El control y la configuración del drone, se realiza mediante el envío de comandos AT en el puerto UDP 5556.
- Información sobre el drone (como su estado, su posición, velocidad, velocidad de rotación del motor, etc.), llaman a *navdata*, luego son enviados por el drone a la estación cliente por el puerto UDP 5554. Estos *navdata*, incluyen etiquetas de información de detección, que se pueden ser utilizados para crear simulaciones de realidad aumentada.
- Un flujo de vídeo, es enviado por el AR.Drone, a la estación cliente en el puerto 5555.

2.1. Cargas útiles, *payloads*

- Cámaras:
    - AR.Drone 2.0, utiliza una cámara frontal de alta definición (720p-30FPS). Esta cámara se puede configurar para transmitir imágenes de 360p (640 * 360) o 720p (1280 * 720).
    - Las imágenes de resolución completa, sólo están disponibles para los algoritmos de detección y la toma de fotografías.
    - La cámara es de tipo QVGA (320 * 240) de 60 FPS. Las imágenes de esta cámara puede aumentar a 360p o 720p para la transmisión de video.
- Puerto USB:
    - AR.Drone 2.0, tiene un puerto USB principal, con un conector USB-A estándar, que se puede utilizar para la grabación de videos y conexión de diferentes *payloads*.

2.2. DRONES DE EXPLORACIÓN

2.2.1. Flyability Elios

Flyability es una empresa suiza que construye UAVS, para explorar lugares inaccesibles. Permite que las máquinas sean utilizadas de manera segura en ciudades, edificios, en contacto con personas, y también resuelve dos problemas críticos: colisiones y riesgos de lesiones. El principal mercado de la empresa se encuentra en la inspección industrial, donde evita el envío de personas a lugares peligrosos como lo son las industrias petroleras, gasíferas y marítimas. También, trabaja con los profesionales de búsqueda y rescate y seguridad, para evaluar situaciones de emergencia sin poner en peligro vidas humanas. Flyability es el ganador del premio *USD 1M Drones for Good Award 2015*, con el desarrollo de Elios (Figura 2-6).

Flyability Elios ha sido diseñado para ser utilizado para la inspección en interiores de espacios reducidos. Incluye una cámara HD mejorada con sensor infrarrojo incorporado por FLIR, una de las principales empresas fabricantes de sensores de cámara. Esto permite al drone obtener imágenes térmicas en tiempo real de la zona (Flyability, 2014).

Elios puede controlar más allá de la línea visual, gracias a un módulo de cámara giratoria de 180 grados. Posee LEDS incorporados, que permiten la inspección de espacios oscuros y complejos. La arquitectura tiene compatibilidad con tarjetas SD, para que las imágenes de video se pueden almacenar y analizar posteriormente (Flyability, 2014).

Se encuentra diseñado con fibra de carbono, lo que le proporciona resistencia a todo tipo de obstáculos. La elección de la fibra de carbono como el material para la jaula viene como resultado de sus propiedades únicas para ser ligero y fuerte (Flyability, 2014).

La Figura 2-6, muestra el diseño esférico del UAV Flyability Elios ideado para exploraciones industriales.

Figura 2-6. UAV Flyability Elios. Extraído de Flyability (2014).

En cuanto a sus especificaciones técnicas, Elios tiene un sistema de vuelo posee 4 motores eléctricos sin escobillas, tiene un peso de 700gr, que incluye la batería, cargas útiles y jaula de protección, el tiempo de vuelo alcanza hasta 10min, la velocidad varía entre 1,5 m/s (en modo normal) 2,5 m / s (en modo de vuelo de alta velocidad), tiene piloto automático y control IMU, magnetómetro, barómetro, puede operar entre -10 a 35° C.

Posee un sistema de comunicación digital, bidireccional, de largo alcance de video y datos de enlace descendente a RC, *Command and Data uplink* a UAV, frecuencia2.4GHz, distancia de hasta 5km, control remoto opcional, cámara operador.

En cuanto a sus cargas útiles (*payloads*), tiene una cámara principal de video FHD (1920 x 1080) a 30fps, con un rango de inclinación: + 90 ° hasta -45 ° hacia abajo, grabación a bordo y transmisión al piloto y operador de cámara, tiene un campo horizontal visual de 130 grados, un campo vertical visual 100 grados, lo que suma un total campo visual: 235 grados (considerando la rotación arriba/abajo de la carga útil).

Cuenta a su vez, con una cámara térmica de video de 160x120 pixeles en 9fps, puede grabar a bordo y opcionalmente *streamed*, tiene un campo horizontal de vista de 56 grados, y un campo vertical de vista de 42 grados.

Tiene un sistema de iluminación es de tipo LED, de alta e ciencia, 5 *arrays* para iluminación uniforme en la parte delantera, superior e inferior del UAV y un haz de luz adaptable controlado por paso de la cámara.

Por último, en cuanto a la autonomía, *Elios* tiene una batería de polímero de litio, 3 células, 2800mAh, 33.08Wh, el tiempo de carga es de 1 hora (Flyability, 2014).

2.3. PROTOCOLOS DE NAVEGACIÓN DE UAVs

Los protocolos de navegación, se pueden de definir como instrucciones, que permiten guiar una acción, o establecer ciertas bases para el desarrollo de un procedimiento. Luego, los protocolos de navegación para los UAVs, se identifican como instrucciones, que generalmente se define bajo puntos de referencia o *waypoints*, es decir, coordenadas geográficas que la máquina recorre de forma secuencial, para completar una misión. Generalmente, los *waypoints*, se utilizan para supervisar zonas específicas de interés.

Bajo un protocolo de navegación, es posible delimitar una zona mediante un polígono, de forma que la máquina no pueda volar fuera del área, es decir, sobrevolar de forma inteligente zonas geográficas predefinidas, o ser modificadas directamente desde la estación base de control.

Por otra parte, de forma interna, los sistemas de control de vuelo, emiten señales en tiempo real, como por ejemplo elevación y velocidad de desplazamiento. Por medio de la posición geográfica, determinadas por el GPS, se puede obtener datos de latitud, longitud y altura, además, correcciones de vuelo entre la posición actual y el *waypoint* objetivo, generando as las consignas de vuelo necesarias. Otras funciones importantes son mantener la posición actual en caso de variaciones en las condiciones climáticas, como también retornar a la estación base en caso de fallos, por ejemplo, estabilidad de los rotores o batería baja (Austin, 2010).
Los protocolos de navegación, son utilizados en diferentes tipos de misiones, por ejemplo:

- Misiones Civiles
    - Levantamiento de mapas.
    - Inspección de obras civiles.
    - Monitoreo y cuidado de cultivos.
    - Búsqueda y rescate.
- Misiones militares
    - Vigilancia y reconocimiento.
    - Misiones de ataque.
    - Señuelos.

A continuación, se muestra el resultado de fotografías en un área agrícola multiespectral, para el análisis de índices de vigor de cultivos, las fotografías fueron tomadas por un UAV de tipo RUAS X1 (Figura 2-7).

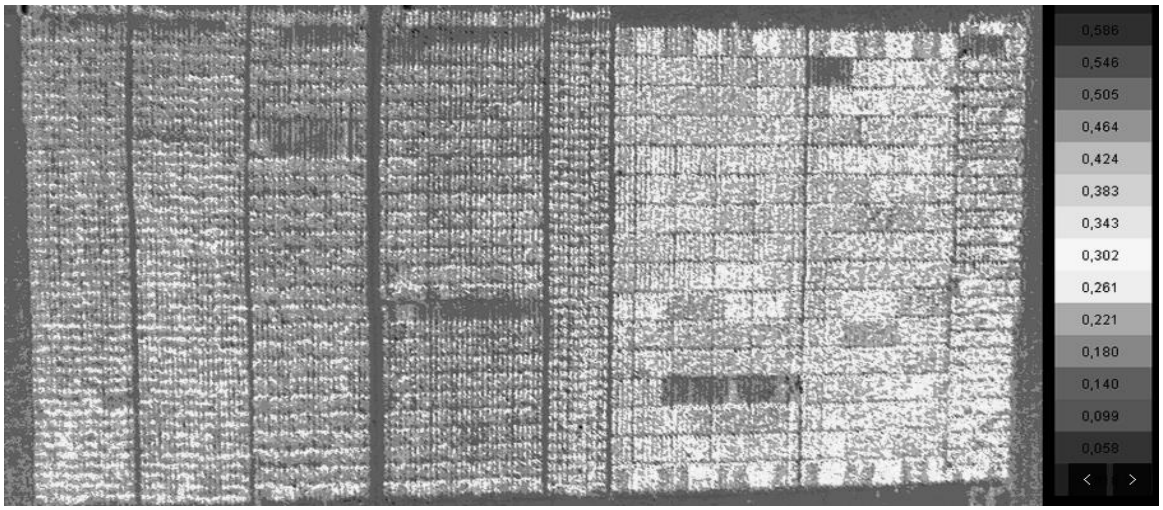

Figura 2-7. Análisis de vigor en la zona de cultivo. Extraído de Sisar (2014).

Los índices de vigor son medidas cuantitativas, basados en valores digitales, que tienden a medir biomasa. Usualmente el índice de vegetación es una combinación de bandas espectrales, que combinados en una forma diseñada por expertos, produce un valor, que indica la cantidad o vigor de la vegetación dentro de un pixel. De esta forma, se permite estimar y evaluar el estado de salud de la vegetación, en base a la medición de la radiación que las plantas emiten o reflejan (Sisar, 2014).

Posteriormente, el protocolo que permite guiar la acción para obtener los índices de vigor de los cultivos, puede configurarse a través de un protocolo que agrupe las siguientes cuatro etapas:

- Apreciación física del territorio.
- Planeación de vuelo.
- Ejecución del vuelo.
- Procesamiento de imágenes.

2.3.1. Apreciación física del territorio

En esta etapa, el objetivo es apreciar de forma física el territorio, con el objetivo de identificar posibles obstáculos, que puedan causar interrumpir el vuelo del UAV. Estos obstáculos pueden ser antenas, arboles, líneas de alta tensión, entre otros. De acuerdo a esta apreciación, es posible fijar la altura de vuelo o cambiar la ruta (Martínez, 2015).

2.3.2 Planeación de vuelo

Una vez realizada la etapa anterior, se lleva a cabo la planeación de vuelo, que consiste en configurar los parámetros de fotografía, los cuales son adaptados a las distintas condiciones del plan, como por ejemplo, el tiempo y la rapidez con que se tienen que capturar las imágenes de acuerdo a la velocidad y la superficie del terreno (Martínez, 2015).

En esta etapa, se puede utilizar la aplicación de código abierto *Mission Planner*, donde se puede programar el vuelo autónomo del UAV, la altitud, velocidad y también un intervalo de tiempo para la captura de imágenes (Figura 2-8).

Figura 2-8. Ejemplo planeación de vuelo a través de *Mission Planner*. Extraído de Martínez (2014).

2.3.3. Ejecución del vuelo

Después de diseñar el plan de vuelo, estos se tienen que enviar al UAV para ejecutar la misión. Se puede utilizar la aplicación (*Mission Planner*), para comunicar el UAV mediante telemetría utilizando el protocolo MavLink. Principalmente, dentro del plan de vuelo se encuentra la trayectoria georreferenciada de cada *waypoint* (Martínez, 2015).

La Figura 2-9 muestra el detalle de una ejecución de vuelo sobre territorio definido utilizando el *software Mission Planner*.

Figura 2-9. Ejemplo ejecución de vuelo a través de *Mission Planner*. Extraído de Martínez (2014).

2.3.1. Procesamiento de imágenes

Finalizada la misión de vuelo y capturadas las imágenes, se tiene que procesar la información, se pueden utilizar aplicaciones como *Agisoft PhotoScan Pro*, para verificar la calidad, generar ortomapas finales, como también, modelos tridimensionales (Martínez, 2015).

2.4. PROTOCOLOS DE ENLACE DE DATOS

Los protocolos de enlace de datos, son utilizados para manejar los sistemas de transmisión de control y telemetría, estos utilizan diferentes tipos de protocolos de paso de mensajes entre la estación base y el UAV, entre los que se encuentra MAVLink.

2.4.1. MAVLink, *Micro Air Vehicle Link*

MAVLink es un protocolo de comunicación, basado en una biblioteca de paso de mensajes, utilizados para mantener la comunicación entre UAVS y la estación de control. Se trata de una biblioteca liviana, de fácil integración, puesto que se encuentra basada en cabeceras en lenguaje C. Estos mensajes empaquetan estructuras de datos y se envían a través de canales serie bidireccionales con poco *overhead* (Qgroundcontrol, 2010).

La arquitectura de los paquetes de datos MAVLink se desarrolla bajo estándares CAN (*Controller Area Network*) y SAE AS-4.

La Figura 2-10, muestra el esquema de campos de la trama MAVLink.

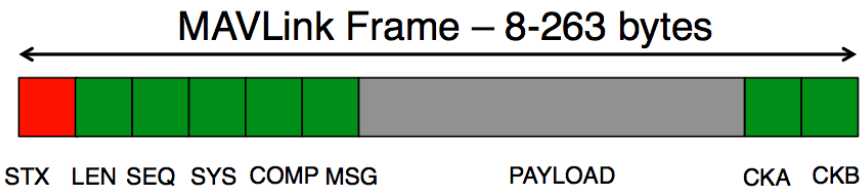

Figura 2-10. Trama de datos MAVLink. Extraído de Qgroundcontrol (2010).

La Tabla 2-1 se describe los campos de la trama del protocolo MAVLink.

Tabla 2-1. Descripción trama de datos MAVLink. Extraído de Qgroundcontrol (2010).

Byte	Contenido	Valor	Explicación
0	*Start Transmission*	0xFE	Indica el inicio de un nuevo mensaje.
1	*Length*	0 – 255	Longitud de los datos.
2	*Sequence*	0 - 255	Secuencias transmitidas.
3	*System ID*	1 - 255	Identificación del sistema que envía el mensaje.
4	*Component ID*	0 - 255	Identificación del componente que envía el mensaje.
5	*Message ID*	0 - 255	Identificación del mensaje.
6 a (n+6)	*Payload*	(0 - 255) bytes	Datos del mensaje.
(n+7) a (n+8)	CRC		Checksum del mensaje.

La trama de paquetes comienza con un byte de inicio, STX. Una vez que se comprueba que se recibe este byte, se lee la longitud del mensaje en el siguiente campo. Conociendo la longitud de los datos que se han enviado, se puede obtener y veri car los bytes de *checksum*. La pérdida de alguno de los bytes del mensaje ocasiona que el *checksum fail*, en este caso se desprecia el mensaje, y el receptor queda a la espera de uno nuevo. Si el *checksum* coincide, se procesa el paquete MAVLink, y se vuelve a esperar a que se reciba un nuevo byte de inicio de mensaje (Qgroundcontrol, 2010).

2.5. ROBOT OPERATING SYSTEM (ROS)

Robot Operating System (ROS) es un conjunto de bibliotecas de software, que agrupa controladores y algoritmos, para el desarrollo de aplicaciones en robótica. El objetivo de ROS es simplificar el comportamiento robótico complejo, para una amplia variedad de plataformas (ros.org, 2012).

ROS, posee un diseño modular y distribuido que permite a los desarrolladores utilizar los componentes de acuerdo a las necesidades y características de la implementación. Por otra parte, el diseño distribuido de ROS. También fomenta que una amplia cantidad de desarrolladores contribuyan con paquetes de software, que agregan valor sobre el núcleo del sistema. Dentro los paquetes de software existen pruebas conceptuales de nuevos algoritmos hasta controladores y capacidades de calidad industrial. La comunidad de usuarios de ROS, se basa en una arquitectura común, con el objetivo de proporcionar un punto de integración que ofrece acceso a controladores de hardware, capacidades de robot genéricas, herramientas de desarrollo, bibliotecas externas útiles, entre otros temas (ros.org, 2012).

En los últimos años, ROS ha crecido para incluir una gran comunidad de usuarios de todo el mundo. Históricamente, la mayor de los usuarios estaban en laboratorios de investigación, pero cada vez más se ve la adopción del sistema en el sector comercial, particularmente en la robótica industrial y de servicios (ros.org, 2012).

ROS por sí mismo ofrece mucho valor a la mayoría de los proyectos de robótica, pero también ofrece una gran oportunidad para trabajar en red y colaborar con proyectos y desarrolladores que forman parte de la comunidad ROS. Por otra parte, el núcleo de ROS esta licenciado bajo la licencia estándar BSD de tres clausulas, lo que permite la redistribución y modificación de los fuentes conservando los derechos de copyright (ros.org, 2012).

2.5.1. Infraestructura de comunicaciones plataforma ROS

En el nivel más bajo, ROS ofrece una interfaz de paso de mensajes que proporciona comunicación interproceso, que se conoce comúnmente como *middleware*. El *middleware* de ROS proporciona la siguiente estructura (ros.org, 2012):

- Paso de mensajes *anonymous publish/subscribe*.
- Grabación y reproducción de mensajes.
- Llamadas de procedimiento remoto *request/response*.
- Sistema de parámetros distribuidos.

2.5.1.1. Paso de mensajes

Un sistema de comunicación, es una de las primeras necesidades que surgen, cuando se implementa una aplicación en robótica. Una de las características del sistema de mensajería incorporado de ROS, es que administra los detalles de la comunicación entre los nodos distribuidos, a través del mecanismo *anonymous publish/subscribe*.

Esto obliga a implementar interfaces claras entre los nodos del sistema, mejorando as la encapsulación y promoviendo la reutilización de código. La estructura de las interfaces de mensajes, se define en el tipo de mensaje IDL (*Interface Description Language*) (ros.org, 2012).

2.5.1.2. Grabación y reproducción de mensajes

Debido a que el sistema *publish/subscribe* es anónimo y además asincrónico, los datos pueden ser fácilmente capturados y reproducirse sin ningún tipo de cambios en el código. Por ejemplo, se tiene la Tarea A, que lee los datos de un sensor y está desarrollando la Tarea B, que procesa los datos producidos por la Tarea A. ROS facilita la captura de los datos publicados por la Tarea A en un archivo y los vuelve a publicar en un momento posterior. La abstracción del paso de mensajes permite que la tarea B sea indiferente con respecto a la fuente de datos, que podría ser la tarea A o el archivo de registro. Este es un potente patrón de diseño que puede reducir significativamente el esfuerzo de desarrollo y promover la flexibilidad y la modularidad del sistema (ros.org, 2012).

2.5.1.3. Llamadas a procedimiento remoto

La naturaleza asíncrona de la mensajería *publish/subscribe* funciona para muchas necesidades de comunicación en robótica, pero a veces se requieren interacciones síncronas de *request/response* entre los procesos. El *middleware* de ROS proporciona esta capacidad utilizando servicios, es decir, los datos que se envían entre los procesos en una llamada de servicio se define con el mismo mensaje de tipo IDL (ros.org, 2012).

2.5.1.4. Sistema de parámetros distribuidos

El *middleware* de ROS proporciona una manera para que las tareas compartidas en relación a la información de configuración, sean a través de un almacenamiento de tipo global. Este sistema permite modificar las tareas, e incluso, que las tareas cambien la configuración de otras tareas asociadas (ros.org, 2012).

2.5.2. Instalación plataforma ROS

La Tabla 2-2 muestra los comandos realizados en la *shell* para preparar fuentes de la plataforma.

Tabla 2-2. Preparación de fuentes. Elaboración propia, 2016.

sudo sh -c 'echo "deb http:/packages.ros.org/ros/ubuntu/ $(lsb release -sc) main)) /etc/apt/sources.list.d/ros-latest.list'

La Tabla 2-3 muestra los comandos realizados en la *shell* para preparar las claves de la plataforma.

Tabla 2-3. Preparación de *keys*. Elaboración propia, 2016.

sudo apt-key adv {keyserver hkp://pool.sks-keyservers.net {recv-key 0xB01FA116

La Tabla 2-4 muestra los comandos realizados en la *shell* para la instalación de la plataforma ROS Desktop-Full.

Tabla 2-4. Instalación ROS Desktop-Full, y herramientas independientes. Elaboración propia, 2016.

```
sudo apt-get update
sudo apt-get install ros-indigo-desktop-full
sudo rosdep init
rosdep update
```

La Tabla 2-5 muestra los comandos realizados en la *shell* para la configuración de entorno.

Tabla 2-5. Preparación del entorno (*shell*). Elaboración propia, 2016.

```
echo source /opt/ros/fuerte/setup.bash))>/.bashrc /.bashrc
```

La Tabla 2.6 muestra los comandos realizados en la shell para la con gurarion de entorno para un nuevo usuario.

Tabla 2-6. Configuración entorno usuario. Elaboración propia, 2016.

```
echo "source /opt/ros/fuerte/setup.bash))> /.bashrc
source /.bashrc

mkdir -p =catkin ws=src
cd =catkin ws=src
catkin init workspace
cd =catkin ws=
catkin make
echo "source =catkin ws=devel=setup:bash"" >> =:bashrc source
=:bashrc
echo $ROS PACKAGE PATH
```

2.5.3. Sistema de archivos ROS

- *Package*.
    - Los *package* son la unidad de organización de software del código ROS. Cada paquete puede contener bibliotecas, ejecutables, *scripts* y otras piezas de *software*.
    - *Manifest*::
        - Metadatos de un paquete, su función principal es definir dependencias entre paquetes a través del archivo (package.xml).
- *Metapacks*.
    - Colección de paquetes que forman una biblioteca de nivel superior, anteriormente llamadas pilas (ros.org, 2012).

2.5.1. Entorno de trabajos ROS

La Tabla 2.7 muestra la estructura de directorios y archivos creados al momento de instalar la plataforma.

Tabla 2-7. Entorno de trabajo y sistema de archivos ROS. Extraído de ros.org (2012).

```
workspace_folder/
   build/
   devel/
   src/
      CMakeLists.txt
      package_1/
         CMakeLists.txt
         package.xml
      ...
      package_n/
         CMakeLists.txt
         package.xml
      meta_package/
            sub_package_1/
                  CMakeLists.txt
                   package.xml                              ...
```

```
                sub_package_n/
                        CMakeLists.txt
                        package.xml
              meta_package/
                        package.xml
```

Luego, la estructura de *package*, se con gura de la siguiente forma (ros.org, 2012):

- CMakeLists.txt: archivo de configuración de la compilación.
- Package.xml: metadatos y dependencias del *package*.
- Mainpage.dox: doc información del *package*.
- *Include/package*$_i$: archivos de encabezado en lenguaje C++.
- Src /: directorio de código fuente.
- *Launch* /: donde se almacenan los archivos de lanzamiento.
- Msg /: mensaje (.msg).
- Tipos de srv /: service (.srv).
- *Scripts* /: *scripts* ejecutables.

2.5.5. Herramientas de línea de comandos rosbash-ROS

- Comando rospack: Herramientas de gestion ROS (Tabla 2-8)

Tabla 2-8. ROS *package management tool*. Elaboración propia, 2016.

rospack list
rospack find turtlesim
rospack depends turtlesim
rospack profile

- Comando roscd: Comando de cambio de directorio para ROS (Tabla 2-9)

Tabla 2-9. Cambio de directorio ROS. Elaboración propia, 2016.

roscd
roscd turtlesim
ls (standard linux shell command)

- Comando rosls: Listar el contenido de un paquete ROS (Tabla 2-10)

Tabla 2-10. Listar paquetes ROS. Elaboración propia, 2016.

roscd (retorna al directorio de trabajo)
rosls turtlesim

2.5.6. Nodos

Los nodos son procesos que realizan cálculos específicos, por ejemplo:

- Control de los rotores de un robot.
- Adquirir imágenes de la cámara.
- Realizar localización.
- Realizar la planificación de rutas.
- Proporcionar visualización grafica del sistema.

2.5.7. Master

- Master es el nodo central de ROS, llamado roscore.
- Almacena temas y servicios de información de registro, para nodos ROS.
- Los nodos establecen comunicación según corresponda.
- Realiza devoluciones de llamada a los nodos cuando la información de registro presenta cambios.
- Permite que los nodos creen conexiones de forma dinámica a medida que se ejecutan nuevos nodos.

2.5.8. Servidor de parámetros

Desde una *shell* linux, el modo de iniciar el nodo principal es a través del siguiente comando (Tabla 2-11).

Tabla 2-11. Inicio nodo principal. Elaboración propia, 2016.

Roscore

En otro terminal, se valida el servidor de parámetros (Tabla 2-12).

Tabla 2-12. Inicio servidor de parámetros. Elaboración propia, 2016.

rosparam list rosparam get /rosdistro rosparam get /rosversion

2.5.9. Mensajes

- Los mensajes son una estructura de datos, que consiste en campos tipados, como por ejemplo un entero o un real.
- Tipos de datos primitivos y matrices anidadas son compatibles con:
  - Int 8, 16, 32, 64.
  - Float 32, 64.
  - String.
  - Time.
  - Duration.
  - Array

Los mensajes son enrutados a través de un sistema de transporte con semántica de tipo *publish/subscribe*.

2.5.10. Tópicos

- Un nodo envía un mensaje, que se publica en un tópico determinado.
- El tipo de tema se define por el tipo de mensaje que se publica.
- Un nodo que requiere un cierto tipo de datos, debe suscribirse al tópico apropiado.
- Se permite múltiples *publishers/subscribers* en el mismo tópico.
- Un solo nodo puede realizar el proceso (*publish/subscribe*) en varios tópicos.
- Los editores y los suscriptores generalmente desconocen existencia.
- El modelo *publish/subscribe* es un paradigma flexible. (*many-to-many, one-way transport*).

2.5.11. Diagrama de Tópicos

La Figura 2-11 muestra el diagrama de tópicos RPC, utilizados por la plataforma para el paso de mensajes entre cliente/servidor.

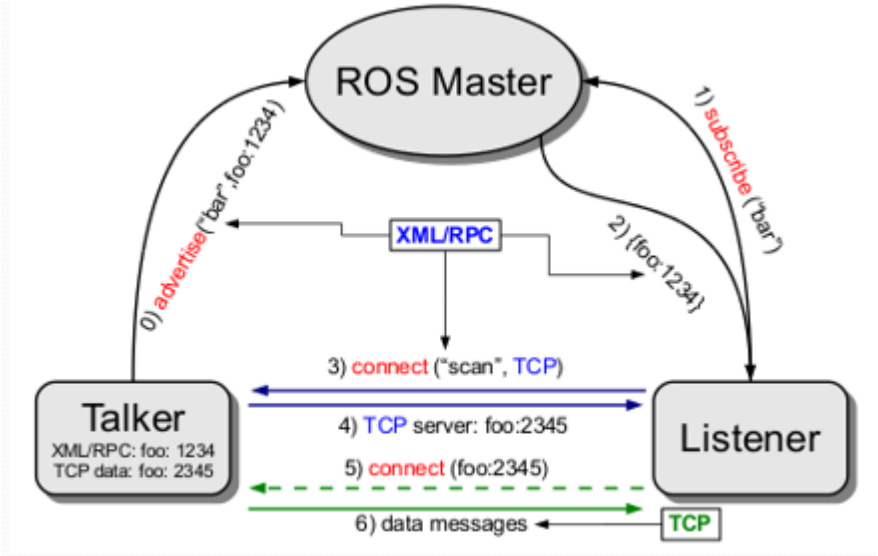

Figura 2-11. Diagrama de tópicos RPC. Extraído de ros.org (2012).

2.5.12. Diagrama de Servicios

- Publicar / suscribir paradigma no apropiado para los servicios.
- Los servicios implementan la funcionalidad de petición / respuesta.
- Par de estructuras de mensajes: uno para la solicitud y otro para la respuesta.
- Un proveedor de nodos ofrece un servicio bajo un nombre específico.
- Un nodo cliente utiliza el servicio enviando el mensaje de solicitud y aguarda la respuesta.
- Desde la perspectiva del programador, funciona como una llamada a procedimiento remoto.

La Figura 2-12 muestra el diagrama de servicios, utilizados por la plataforma en las publicaciones, paso de mensajes y llamadas de procedimiento remoto.

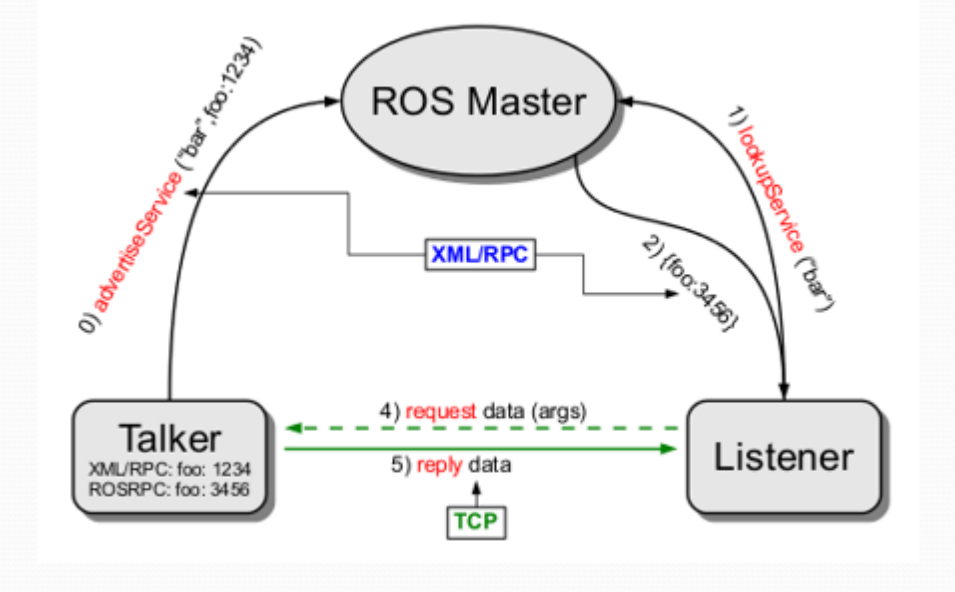

Figura 2-12. Diagrama de Servicios. Extraído de ros.org (2012).

2.5.13. ROSCORE / ROSRUN

ROSCORE es una colección de nodos y programas que son requisitos de un sistema basado en ROS. Debe tener un ROSCORE en ejecución para que los nodos ROS se comuniquen. Se inicia con el comando roscore (ros.org, 2012).

Por otra parte ROSRUN, es un comando interno que permite iniciar los *package* solicitados como también ejecutar el paso de parámetros.

- roscore
- rosrun cmd_vel_publisher cmd_vel_publisher_node
- rosrun cmd_vel_publisher cmd_vel_publisher_node _Max_Constant_Vel:=0.5

2.5.14. Funciones claves de ROS

- Abstracción de hardware y control de dispositivos de bajo nivel.
- Independencia del lenguaje de programación.
- Implementación de una amplia gama de herramientas y algoritmos de uso común.
- Mensaje que pasa entre procesos (independiente del sistema operativo).
- Gestión estandarizada de paquetes.
- Conjunto útil de comandos de *shell*.

2.5.15. Lenguajes de programación utilizados por ROS

- Python
- C++.
- Lisp.

2.6. SOCKET

Tiempo atrás, conectarse a través de red, implicaba realizar conexiones punto a punto por medio de una línea serial dedicada; ningún otro equipo utilizaba el mismo circuito, y UNIX utilizaba el protocolo UUCP(UNIX-to-UNIX *copy*) para mover archivos de un lado a otro. A medida que la tecnología avanzaba, el concepto de compartir la misma línea de transmisión se hizo factible, es decir, cada máquina se identifica de manera única para tomar turnos de transmisión. Hay varios métodos para compartir tiempo en la red y muchos trabajan bastante bien, entre ellas la comunicación por *Socket* (BracaMan, 2009).

Los socket trabajan con programas y sistemas que se ejecutan en modo simultáneo, esto significa, que se necesita considerar la sincronización, el tiempo y la gestión de los recursos. Los sockets enlazan tareas asíncronas mediante de un único canal bidireccional. Esto podría conducir a problemas de *deadlock/starvation*. Luego, con buena planificación de tareas, como por ejemplo la división de la carga, se puede evitar la mayor a de estos problemas, el tiempo y la gestión de los recursos, reduce la carga del servidor, aumentando de esta forma el rendimiento (BracaMan, 2009).

Por otra parte, la red fue diseñada para ser de forma completa un conmutador de paquetes. Cada paquete debe tener toda la información necesaria, como el fuente y las direcciones de destino. El paquete cambia de una maquina a otra a lo largo de los enlaces. Si la red pierde un enlace al pasar un mensaje, el paquete encuentra otra ruta (*packet switching*), o en caso de error, el *router* puede rebotar si no llega al host, esto asegura la confiabilidad de los datos. Las interrupciones o caminos rotos en la red, resultan ser probablemente cortes en la señal (BracaMan, 2009).

2.6.1. Direccionamiento TCP/IP

Las redes admiten diferentes tipos de protocolos. Los programadores han adaptado algunos de ellos, para abordar algunos problemas específicos, como la comunicación a través de radio/microondas; otros intentan resolver la confiabilidad de los datos en red. El protocolo TCP/IP (*Transmission Control Protocol/Internet Protocol*), se centra en datos de paquetes unitarios y el potencial número de canales de comunicación perdidos. En cualquier momento, el

protocolo intenta encontrar una nueva ruta por medio de algoritmos, cuando un segmento de red falla (*deadlock/starvation)* (BracaMan, 2009).

Fundamentalmente, cada paquete contiene los datos, principalmente la dirección del origen y de destino. Cada capa en el protocolo, en este caso (TCP/IP), agrega su propia firma y otros datos en el *wrapper*, al paquete de transmisión. Cuando se transmite el *wrapper*, ayuda al receptor a reenviar el mensaje a la capa apropiada para esperar la lectura, posteriormente cada máquina conectada a la red, tiene una dirección (IP) única.

2.6.2. Tipos de Socket

Los sockets son una forma de comunicarse con las maquinas a través de descriptores de ficheros, los cuales son un fichero abierto en una conexión a un terminal. La comunicación de los sockets, pueden ser de dos tipos:

- Socket de Flujo: Los sockets de flujo, están desarrollados para conexiones de tres pasos, *three way handshake*, es decir, el cliente pide autorización para conectarse, luego acepta o declina, a partir de ahí comienza el paso de mensajes.
- Socket de datagramas: Los sockets de tipo UDP, no soportan conexiones de tres pasos, el cliente, envía los datos al servidor, sin necesidad de conexión. Si se pierde un paquete en el camino, no se puede recuperar y por tanto no llegar a destino, (el presente trabajo utilizará socket de tipo TCP).

2.6.3. Estructuras de Sockets en C

A continuación, se presenta las estructuras de *sockets* utilizadas en programación en lenguaje C para la construcción de clientes y servidores (BracaMan, 2009).

2.6.3.1. SOCKADDR

Estructura que contiene información genérica del socket, donde incluye el tipo de familia de la dirección y dirección del protocolo (Tabla 2-13).

Tabla 2-13. Estructura SOCKADDR. Extraído de BracaMan (2009).

```
1 struct sockaddr
2 { unsigned short sa_family ;
3 char sa_data [14];
4 };
```

2.6.3.2. SOCKADDR IN

Estructura que contiene la referencia a los elemento del socket (Tabla 2-14).

Tabla 2-14. Estructura SOCKADDR IN. Extraído de BracaMan (2009).

```
1 struct sockaddr_in
2 {
3 short int sin_family ;
4 unsigned short int sin_port ;
5 struct in_addr sin_addr ;
6 unsigned char sin_zero [8];
                    7 };
```

2.6.3.3. IN ADDR

Estructura que se utiliza como UNION de la dirección sin ADDR (Tabla 2-15).

Tabla 2-15. Estructura IN ADDR. Extraído de BracaMan (2009).

```
1 struct in_addr
2 {
3 unsigned long s_addr ;
4 };
```

2.6.3.4. HOSTENT

Estructura que se utiliza para obtener información desde el nodo remoto (Tabla 2-16).

Tabla 2-16. Estructura HOSTENT. Extraído de BracaMan (2009).

```
1 struct hostent
2 {
3 char * h_name ;
4 char ** h_aliases ;
5 int h_addrtype ;
6 int h_length ;
7 char ** h_addr_list ;
8 # define h_addr h_addr_list [0]
9 };
```

2.6.4. Funciones de *Socket*

A continuación, se describen las principales funciones utilizadas en *sockets,* en lenguaje de programación C.

2.6.4.1 Socket()

La función socket(), devuelve un descriptor de socket, el cual se utilizará para posteriores llamadas al sistema (BracaMan, 2009).

- **domain**: Se pueden utilizar como AF INET (protocolos ARPA) o AF UNIX (protocolos internos), son los más utilizados.
- **type**: Se especifica la clase de socket, puede ser de flujos (SOCK STREAM) o de datagramas (SOCK DGRAM).
- **protocol**: Se establece el protocolo a 0.

La Tabla 2-17 muestra la declaración de cabecera de la función socket().

Tabla 2-17. Función socket(). Extraído de BracaMan (2009).

```
1 # include <sys / types .h>
2 # include <sys / socket .h>
3 int socket ( int domain ,int type , int protocol )
```

2.6.4.2 Bind()

La función bind() se utiliza cuando se considera que los puertos de la máquina local, se utilizarán posteriormente. Su función es asociar un socket con un puerto (BracaMan, 2009).

- **fd**: Descriptor del socket, devuelto por la llamada a la función socket().
- **my addr**: Puntero a una estructura sockaddr.
- **addrlen**: Longitud de la estructura sockaddr, donde apunta el puntero my addr.

La Tabla 2-18 muestra la declaración de cabecera de la función bind().

Tabla 2-18. Función bind(). Extraído de BracaMan (2009).

```
1 # include <sys / types .h>
2 # include <sys / socket .h>
3 int bind ( int fd , struct sockaddr * my_addr ,int addrlen );
```

2.6.4.3 Connect()

La función connect() se usa para conectar a un puerto de una dirección IP. Devolverá (-1) si ocurre algún error (BracaMan, 2009).

- **fd**: Fichero descriptor del socket, devuelto por la llamada a la función socket().
- **serv addr**: Puntero a la estructura sockaddr, la cual contiene la dirección IP destino y puerto.
- **addrlen**: Al igual que la función bind(), este argumento se establece como la longitud de la estructura sockaddr, sizeof(struct sockaddr).

La Tabla 2.19 muestra la declaración de cabecera de la funcion connect().

Tabla 2-19. Función connect(). Extraído de BracaMan (2009).

```
1 # include <sys / types .h>
2 # include <sys / socket .h>
3 int connect ( int fd , struct sockaddr * serv_addr , int addrlen );
```

2.6.4.4 Listen()

La función listen(), se utiliza para esperar conexiones entrantes, lo cual significa esperar que un cliente se conecte a la máquina (BracaMan, 2009).

- **fd**: Fichero descriptor del socket, retorna de la llamada a la función socket().
- **backlog**: Número de conexiones permitidas.

La Tabla 2-20 muestra la declaración de cabecera de la función listen().

Tabla 2-20. Función listen(). Extraído de BracaMan (2009).

```
1 # include <sys / types .h>
2 # include <sys / socket .h>
3 int listen ( int fd ,int backlog );
```

2.6.4.5 Accept()

Cuando una máquina quiere conectarse a través de la red a otra máquina, utiliza accept() para conseguir establecer conexión (BracaMan, 2009).

- **fd**: Fichero descriptor del socket, que fue devuelto por la funci_on listen().
- **addr**: Puntero a estructura sockaddr in, donde se determina el nodo que se contacta y de que puerto.
- **addrlen**: Longitud de la estructura a la que apunta addr, se establece como sizeof(struct sockaddr in).

La Tabla 2-21 muestra la declaración de cabecera de la funcion accept().

Tabla 2-21. Función accept(). Extraído de BracaMan (2009).

```
1 # include <sys / types .h>
2 # include <sys / socket .h>
3 int accept ( int fd , void *addr , int * addrlen );
```

2.6.4.6 Send()

La función send(), envía datos utilizando el protocolo de socket designado, flujos o datagramas (BracaMan, 2009).

- **fd**: Fichero descriptor del socket.
- **msg**: Puntero apuntando al dato que se desea enviar.
- **len**: Longitud en bytes del dato que se envía.
- **flags**: Se establece en 0.

La Tabla 2-22 muestra la declaración de cabecera de la función send().

Tabla 2-22. Función send(). Extraído de BracaMan (2009).

```
1 # include <sys / types .h>
2 # include <sys / socket .h>
3 int send ( int fd , const void *msg , int len , int flags );
```

2.6.4.7 Recv()

La función recv(), devuelve el número de bytes que se encuentran en el *buffer* (BracaMan, 2009).

- **fd**: Descriptor del socket donde se leen los datos.
- **buf**: *Buffer* donde se almacena la información.
- **len**: Longitud del buffer.
- **flags**: Se establece en 0.

La Tabla 2-23 muestra la declaración de cabecera de la función recv().

Tabla 2-23. Función recv(). Extraído de BracaMan (2009).

```
1 # include <sys / types .h>
2 # include <sys / socket .h>
3 int recv ( int fd , void *buf , int len , unsigned int flags );
```

2.6.4.8 Close()

La función close(), se utiliza para cerrar la conexión del descriptor del socket (BracaMan, 2009). La Tabla 2-24 muestra la declaración de cabecera de la función close().

Tabla 2-24. Función close(). Extraído de BracaMan (2009).

```
1 # include <unistd .h>
2 close(fd);
```

2.6.4.9 Shutdown()

La función shutdown(), establece mecanismos de inhabilidad en la comunicación (BracaMan, 2009)

- **fd**: Descritor del socket.
- **how**: 0 (prohibido recibir), 1 (prohibido enviar), 2 (prohibido enviar y recibir).

La Tabla 2-25 muestra la declaración de cabecera de la función shutdown().

Tabla 2-25. Función shutdown(). Extraído de BracaMan (2009).

```
1 # include <sys / socket .h>
2 int shutdown ( int fd , int how );
```

2.6.4.10 Gethostname()

La función gethostname(), se utiliza para obtener el nombre de la máquina local (BracaMan, 2009).

- **hostname**: Puntero a al arreglo del nodo actual.
- **size**: Longitud en bytes del arreglo.

La Tabla 2-26 muestra la declaración de cabecera de la función gethostname().

Tabla 2-26. Función gethostname(). Extraído de BracaMan (2009).

```
1 # include <unistd .h>
2 int gethostname ( char * hostname , size_t size );
```

2.6.5. Ejemplo de servidor de flujos

El presente código muestra un ejemplo de la programación de un servidor de flujos en lenguaje de programación C.

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PORT 3550 /* puerto de comunicaci\'on */
#define BACKLOG 2 /* El n\'umero de conexiones permitidas */

main()
{

   int fd, fd2; /* los ficheros descriptores */

   struct sockaddr_in server;
   /* Informaci\'on de la direcci\'on del servidor */

   struct sockaddr_in client;
   /* Informaci\'on de la direcci\'on del cliente */

   int sin_size;

   /* Llamada al socket() */
   if ((fd=socket(AF_INET, SOCK_STREAM, 0)) == -1 ) {
      printf("error en socket()\n");
      exit(-1);
   }

   server.sin_family = AF_INET;

   server.sin_port = htons(PORT);
   /* conversiones */

   server.sin_addr.s_addr = INADDR_ANY;
   /* INADDR_ANY coloca nuestra direcci\'on IP autom\'aticamente */

```

```
36      bzero(&(server.sin_zero),8);
37      /* escribimos ceros en el reto de la estructura */
38
39
40      /* A continuaci\'on la llamada a bind() */
41      if(bind(fd,(struct sockaddr*)&server,
42              sizeof(struct sockaddr))==-1) {
43         printf("error en bind() \n");
44         exit(-1);
45      }
46
47      if(listen(fd,BACKLOG) == -1) {  /* llamada a listen() */
48         printf("error en listen()\n");
49         exit(-1);
50      }
51
52      while(1) {
53         sin_size=sizeof(struct sockaddr_in);
54         /* A continuaci\'on la llamada a accept() */
55         if ((fd2 = accept(fd,(struct sockaddr *)&client,
56                           &sin_size))==-1) {
57            printf("error en accept()\n");
58            exit(-1);
59         }
60
61         printf("Se obtuvo una conexi\'on desde %s\n",
62                 inet_ntoa(client.sin_addr) );
63         /* que mostrar\'a la IP del cliente */
64
65         send(fd2,"Bienvenido al servidor.\n",22,0);
66         /* Mensaje de bienvenida al cliente */
67
68         close(fd2); /* cierra fd2 */
69      }
70 }
```

2.6.6. Ejemplo de cliente de flujos

El presente código muestra un ejemplo de la programación de un cliente de flujos en lenguaje de programación C.

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#define PORT 3550
/* puerto de conexi\'on */

#define MAXDATASIZE 100
/* N\'umero m\'aximo de datos en bytes */

int main(int argc, char *argv[])
{
   int fd, numbytes;
   /* ficheros descriptores */

   char buf[MAXDATASIZE];
   /* donde almacenar\'a el texto recibido */

   struct hostent *he;
   /* estructura que recibir\'a informaci\'on sobre el nodo remoto */

   struct sockaddr_in server;
   /* informaci\'on sobre la direcci\'on del servidor */

   if (argc !=2) {
      printf("Uso: %s <Direcci\'on IP>\n",argv[0]);
      exit(-1);
   }

   if ((he=gethostbyname(argv[1]))==NULL){
      /* llamada a gethostbyname() */
      printf("gethostbyname() error\n");
      exit(-1);
   }
```

```c
   if ((fd=socket(AF_INET, SOCK_STREAM, 0))==-1){
      /* llamada a socket() */
      printf("socket() error\n");
      exit(-1);
   }

   server.sin_family = AF_INET;
   server.sin_port = htons(PORT);
```

```
45
46      server.sin_addr = *((struct in_addr *)he->h_addr);
47      /*he->h_addr pasa la informaci\'on de ``*he'' a "h_addr" */
48      bzero(&(server.sin_zero),8);
49
50      if(connect(fd, (struct sockaddr *)&server,
51         sizeof(struct sockaddr))==-1){
52         /* llamada a connect() */
53         printf("connect() error\n");
54         exit(-1);
55      }
56
57      if ((numbytes=recv(fd,buf,MAXDATASIZE,0)) == -1){
58         /* llamada a recv() */
59         printf("Error en recv() \n");
60         exit(-1);
61      }
62
63      buf[numbytes]='\0';
64
65      printf("Mensaje del Servidor: %s\n",buf);
66      /* muestra el mensaje de bienvenida del servidor =) */
67
68      close(fd);
69
70  }
```

CAPÍTULO 3: DISEÑO PROTOCOLO DE NAVEGACIÓN ENTRE UAVS

En el presente capítulo, se presenta el diseño del protocolo de navegación entre UAVS, orientado a exploraciones de reconocimiento, donde se describe en primera instancia, en que consiste una exploración de reconocimiento, en segundo lugar cuál es el protocolo de retorno ante fallos y por último la configuración multidrone que permite la interoperabilidad con la plataforma ROS.

3.1. EXPLORACIONES DE RECONOCIMIENTO

La Expedición Endurance, fue una de las más grandes expediciones realizadas en el continente antártico, fue planificada y dirigida por el reconocido explorador Ernest Shackleton en 1914.

Anteriormente, Shackleton habrá realizado incursiones en el subcontinente, descubriendo importantes vías de navegación y aportando algunos de los primeros datos de gran valor científico que se tienen de esta inhóspita región.

El objetivo de esta expedición era el más ambicioso de todos hasta el momento: atravesar la Antártica de costa a costa pasando por el Polo Sur, un viaje de alrededor de 300 kilómetros que sin dudas pondrá a prueba la resistencia humana. Con ello, se recogerá una valiosa información para la ciencia y, por supuesto, realzará el papel del Reino Unido como potencia mundial.

Según cuenta la historia, para reunir una tripulación acorde, Shackleton publicó una aviso en el de Londres:

"Men wanted for hazardous journey. Low wages, bitter cold, long hours of complete darkness. Safe return doubtful. Honour and recognition in event of success".

"Se buscan hombres para viaje peligroso. Salarios bajos, frío extremo, largos meses de completa oscuridad. No se asegura el regreso. Honor y reconocimiento en caso de éxito".

Se presentaron más de 5000 aspirantes de los cuales 28 partieron en agosto de 1914 desde Londres a bordo del *Endurance*.

Una vez en la Antártica, las pésimas condiciones meteorológicas hicieron que la embarcación quedara atrapada por el hielo y se hundiera el 21 de noviembre de 1915, presagiando lo peor para los tripulantes, que lograron salvar lo que pudieron para sobrevivir. No obstante, se propusieron cumplir con su misión y empleando trineos, recorrieron miles de kilómetros escribiendo una de las epopeyas más grandes que el hombre ha realizado (de la Nuez, 2014).

La Figura 3-13 muestra al bergantín *Endurance*, encallado en un bloque de hielo en las cercanías de las islas *Shetland* del Sur.

Figura 3-13. Fotografía bergantín *Endurance*. Extraído de De la Nuez (2014).

Mientras en el mundo ya se daban por perdidos, los hombres siguieron avanzando y a mediados del mes de abril de 1916 alcanzaron el archipiélago Shetland del Sur y días más tarde, la isla de Georgia del Sur. Sin recursos, Shackleton con dos de los expedicionarios partieron en busca de una estación ballenera que se encontraba del otro lado de la isla (de la Nuez, 2014)

En agosto de ese año, algo más de 24 meses después de su salida de Londres, lograron contactar con la estación y ayudados por remolcador chileno Yelcho comandado por Luis Pardo, volvieron a buscar a todos los hombres, ninguno de los cuales murió, milagrosamente. La epopeya concluyó oficialmente en 1917 (de la Nuez, 2014) Aunque la misión no se logró de forma completa, estuvieron muy cerca, pese a las condiciones en que se encontraban. Los testimonios, las cartas de navegación fueron un avance fundamental en el conocimiento profundo de un territorio desconocido y que a la larga asentó un punto de vanguardia a expediciones futuras.

La Figura 3.2 muestra a la tripulación del *Endurace* antes del rescate el 30 de agosto de 1916.

Figura 3-14. Fotografía rescate bergantín *Endurance*. Extraído de De la Nuez (2014).

3.2. PROTOCOLO DE RETORNO ANTE FALLOS

De acuerdo a lo mencionado en el marco teórico, se puede decir que los protocolos de navegación, se pueden definir como instrucciones, que permiten guiar una acción, o establecer ciertas bases para el desarrollo de un procedimiento determinado. Para este trabajo, el protocolo de navegación está orientado a exploraciones de reconocimiento, es decir, el

protocolo se basa en explorar un territorio desconocido, donde no se conoce las condiciones del terreno ni tampoco las variaciones climáticas.

En primera instancia, se define las dos etapas base del protocolo (Capítulo 2.3):

- Apreciación física del territorio.
- Planeación de vuelo.

Donde la apreciación física del territorio, se realiza de forma visual, y la planeación del vuelo, a través de *waypoints*, es decir, coordenadas geográficas que las máquinas recorrerán de forma secuencial, para completar una misión de reconocimiento. (Para la elaboración de este protocolo, se asume que las máquinas ya poseen conexión multidrone y se conectan al servidor ROS).

En la tercera etapa de ejecución de vuelo, se realiza a través de un *launch*, donde las máquinas se despliegan a cumplir la misión de reconocimiento.

Es en ésta etapa, los UAVS se conectan al servidor central y le envían los datos del reconocimiento, la idea principal, es volver con información, porque, de esta manera, es posible más adelante, enviar una misión de reconocimiento con personas.

Posteriormente, el protocolo secundario de retorno ante fallos, se asienta en esta etapa. Se define como: "Si existe pérdida de comunicación de alguno de los exploradores, inicia el protocolo secundario de retorno, que indica a todos los exploradores restantes, el retorno a la estación base principal".

De esta forma, al igual que una operación de avanzada, permite recabar información logística, para realizar una nueva planeación y seguir explorando.

Finalmente el protocolo de retorno ante fallos, se configura de la siguiente manera:

- Apreciación física del territorio.
- Planeación de vuelo.
- Ejecución de vuelo.

- Si existe pérdida de comunicación de alguno de los exploradores, inicia el protocolo secundario de retorno, que indica a todos los exploradores restantes, el retorno a la estación base principal.

- Procesamiento de la información.

3.3. CONEXIÓN MULTIDRONE

A continuación, se muestra la forma de configuración de varios UAVS a una misma red WIFI, sin romper la conexión ad-hoc predeterminada.

3.3.1 Configuración de red WIFI

En primer lugar, el *router*, se configura sin seguridad. Se configura para que se obtengan direcciones IP del tipo 192.168.1.1 en la subred 255.255.255.0. Luego, se configura el servidor DHCP, para entregar direcciones IP a los clientes a partir de 192.168.1.100, el *router* utilizado es un HUAWEI modelo E5338 (Figura 3-15).

Figura 3-15. *Router* HUAWEI E5338. Elaboración propia (2016).

La Figura 3-16 muestra la forma de configurar la seguridad del *router* HUAWEI E5338.

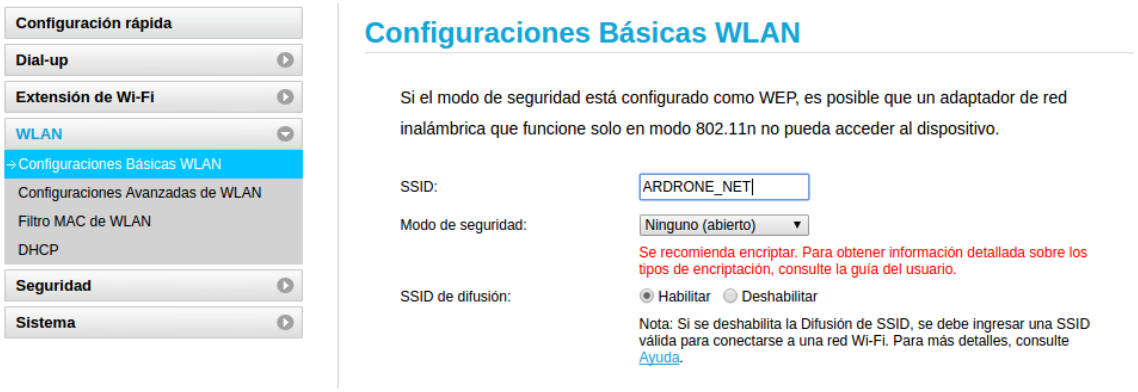

Figura 3-16. Configuración sin seguridad, *router* HUAWEI E5338 . Elaboración propia (2016).

La Figura 3-17 muestra la forma de configurar el servidor DHCP del *router* HUAWEI E5338.

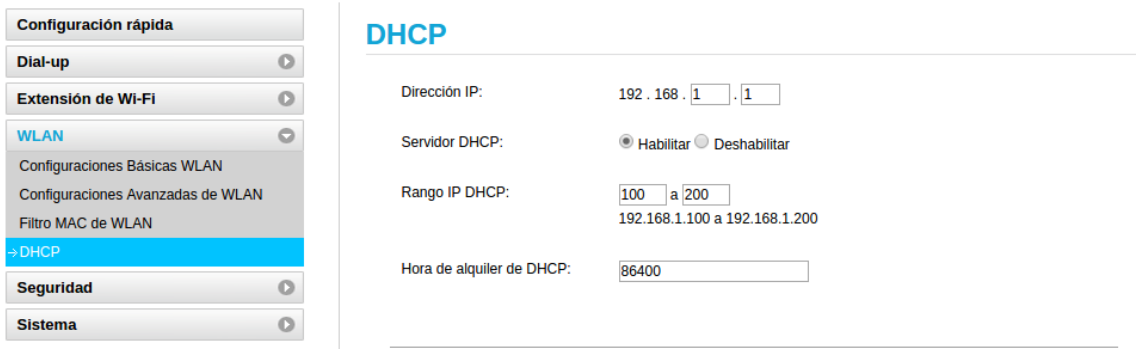

Figura 3-17. Configuración servidor DHCP, *router* HUAWEI E5338. Elaboración propia (2016).

3.3.2 Configuración AR. Drone 2.0

En este paso, se crea el archivo wi_.sh, en el sistema operativo del AR. Drone 2.0, esto permite que el UAV se conecte a la red WIFI de forma predefinida cuando se encuentre operativo. La conexión al UAV, se realiza v vía telnet (Tabla 3-27).

Tabla 3-27. Conexión telnet máquina AR. Drone 2.0. Elaboración propia, 2016.

antizun@PRIME# telnet 192.168.1.1 Trying 192.168.1.1... Connected to 192.168.1.1. Escape character is '^]'. BusyBox v1.14.0 () built-in shell (ash) Enter 'help' for a list of built-in commands.

La creación del archivo, se realiza a través del editor VI (Tabla 3-28).

Tabla 3-28. Creación archivo wifi.sh. Elaboración propia, 2016.

```
antizun@PRIME# telnet 192.168.1.1
Trying 192.168.1.1... Connected to 192.168.1.1. Escape character is '^]'.
BusyBox v1.14.0 () built-in shell (ash) Enter 'help' for a list of built-in commands.
vi /data/wi_.sh
```

Luego, en el archivo, se asigna la ESSID del *router*, y la IP que utilizará el drone (Tabla3-29).

Tabla 3-29. Asignación ESSID ARDRONE_NET. Elaboración propia, 2016.

```
killall udhcpd
ifcon_g ath0 down
iwcon_g ath0 mode managed essid ARDRONE NET
ifcon_g ath0 192.168.1.10 netmask 255.255.255.0 up
```

Una vez terminado que culmina el proceso de edición del archivo, se cambian los permisos para que sea ejecutable (Tabla 3-30).

Tabla 3-30. Modificaciones credenciales archivo WIFI.SH. Elaboración propia, 2016.

```
antizun@PRIME# telnet 192.168.1.1
Trying 192.168.1.1... Connected to 192.168.1.1. Escape character is '^]'.
BusyBox v1.14.0 () built-in shell (ash) Enter 'help' for a list of built-in commands.

chmod +x /data/wifi.sh
```

Finalmente, se cierra la conexión telnet (Tabla 3-31).

Tabla 3-31. Cierre conexión telnet AR.DRONE 2.0. Elaboración propia, 2016.

```
antizun@PRIME# telnet 192.168.1.1
Trying 192.168.1.1... Connected to 192.168.1.1. Escape character is '^]'.
BusyBox v1.14.0 () built-in shell (ash) Enter 'help' for a list of built-in commands.

exit
```

3.3.3 Ejecución archivo WIFI.SH desde estación cliente

Para ejecutar el wifi.sh, se establece conexión telnet desde estación cliente (Tabla 3-32).

Tabla 3-32. Ejecución archivo WIFI.SH desde estación cliente. Elaboración propia, 2016.

antizun@PRIME# echo "./data/wi_.sh"| telnet 192.168.1.1

Para probar conexión punto a punto entre la estación cliente y el AR. Drone, se valida a través de un comando *ping* a la dirección IP del robot (Tabla 3-33).

Tabla 3-33. Validación conexión punto a punto AR.Drone. Elaboración propia, 2016.

antizun@PRIME# ping 192.168.1.1

CAPÍTULO 4: DISEÑO ARQUITECTURA DE COMUNICACIONES ROS

El presente capítulo, muestra el desarrollo de la arquitectura de comunicaciones bajo la plataforma ROS, en él se describe el controlador utilizado para el paso de mensajes los AR. Drone y el servidor ROS, los nodos de la plataforma.

4.1. CONTROLADOR ARDRONE AUTONOMY PARA ROS

Ardrone autonomy, es un controlador ROS para Parrot AR-Drone 1.0 y 2.0, que fue desarrollado por *Autonomy Lab* de la Universidad Simon Frazer.

El controlador, permite la comunicación y paso de mensajes entre la máquina y la plataforma ROS.

La instalación, se puede realizar a través de línea de comandos utilizando *apt-get* (Tabla 4-34).

Tabla 4-34. Validación conexión punto a punto AR.Drone. Elaboración propia, 2016.

antizun@PRIME# apt-get install ros-*-ardrone-autonomy

Una vez integrado a la plataforma ROS, es posible iniciar los nodos principales de la plataforma.

4.2. LECTURA DATOS GPS FLIGTH RECORDER UTILIZANDO EL CONTROLADOR DE VUELO ARDRONE-AUTONOMY

Para la lectura de datos, se utiliza el dispositivo *Fligth Recorder*, que se integra al AR. Drone mediante puerto USB (Figura 4-18).

Figura 4-18. GPS *Fligth Recorder*. Elaboración propia (2016).

Luego, para integrar la lectura de datos, con la plataforma ROS, se ejecuta el siguiente comando (Tabla 4-35).

Tabla 4-35. Integración GPS con el controlador de vuelo *ardrone autonomy*. Elaboración propia, 2016.

antizun@PRIME# rosrun ardrone_autonomy ardrone_driver _enable_navdata_gps := True

4.3. INICIO DE NODOS PLATAFORMA ROS

Los nodos principales, permiten la comunicación entre las máquinas y la plataforma ROS, entre los cuales se cuentan, el nodo controlador de la máquina (*ardrone driver*), el nodo de control (*command control*) y el nodo de control por socket (*command socket*). Para el caso de este trabajo, el servicio de control directo, se realizará a través del nodo *command socket*, puesto que el cliente de comunicaciones es un *socket* que se comunicará a este servicio por el puerto 5204.

En la Tabla 4-36 se puede apreciar la instancia del archivo *command socket.py*, que permite al servicio escuchar por el puerto 5204 en la máquina local.

Tabla 4-36. Nodo *command socket*, servicio puerto 5204. Elaboración propia, 2016.

```
1 Skt = socket . socket ( socket . AF_INET , socket . SOCK_STREAM )
2 ip = '127.0.0.1 '; port = 5204
3 Skt . bind ((ip , port ))
4
5 while not rospy . is_shutdown ():
6 RECV = listen (Skt)
7 talk ( RECV )
```

Luego de establecer los puertos de comunicación, se inician los servicios a través del comando reservado *launch*, que ejecuta el nodo principal *command launch*, que a su vez deja operativo el nodo *command socket* (Tabla 3-37).

Tabla 4-37. Inicio de servicios plataforma ROS. Elaboración propia, 2016.

```
prime@prime - VirtualBox :~/ ros_commandloud$ roslaunch system_command_loud / launch /
command_launch . launch
... logging to / home / prime /. ros / log / c572ed8c -d283 -11e6 -a88a -0800272 e6b5d / roslaunch -
prime - VirtualBox -2975. log
Checking log directory for disk usage . This may take awhile .
Press Ctrl -C to interrupt
Done checking log file disk usage . Usage is <1GB.

started roslaunch server http :// prime - VirtualBox :35830/

SUMMARY
========

PARAMETERS
* / rosdistro
* / rosversion

NODES
/
ardrone_driver ( ardrone_driver / ardrone_driver )
command_control ( system_command_loud / command_control .py)
command_socket ( system_command_loud / command_socket .py)
ROS_MASTER_URI = http :// localhost :11311
core service [/ rosout ] found
process [ command_socket -1]: started with pid [2995]
process [ command_control -2]: started with pid [2996]
process [ ardrone_driver -3]: started with pid [2997]
```

Luego de iniciar los nodos, la plataforma se encuentra operativa para recibir las órdenes a través del cliente de comandos de la estación base.

CAPÍTULO 5: ESTACIÓN BASE CLIENTE SOCKET

En el presente capítulo, se presenta el diseño del cliente de comandos, el cual se comunicará con el servidor ROS a través de paso de mensajes por el puerto asignado, se describe además, las principales funciones utilizadas como son el despegue, el movimiento y el aterrizaje.

5.1. CLIENTE DE COMANDOS SOCKET STREAM

El cliente de comandos, es una aplicación desarrollada en lenguaje de programación C, que utiliza *socket stream*, para paso de mensajes entre máquinas.

El cliente, utiliza dos funciones base, que son:

- take of.
- move.
- land.

La Tabla 5-38 muestra la línea de comandos almacenada en el *buffer*, que hace referencia a la operación de despegue.

Tabla 5-38. Función *socket* TAKE OFF. Elaboración propia, 2016.

```
1 sockfd = socket ( AF_INET , SOCK_STREAM , 0);
2 if ( connect ( sockfd , ( struct sockaddr *) & serv_addr , sizeof ( serv_addr )) < 0) error (
" ERROR connecting ");
3 memset ( buffer ,0, sizeof ( buffer ));
4 strcpy ( buffer ," {\" STATE \":{\" OPC \":\" takeoff \", \" DELAY \":0} ,\" HEIGHT \":100}} ");
```

La Tabla 5-39 muestra la línea de comandos almacenada en el *buffer*, que hace referencia a la operación de despliegue en movimiento.

Tabla 5-39. Función *socket* MOVE. Elaboración propia, 2016.

```
1 sockfd = socket ( AF_INET , SOCK_STREAM , 0);
2 if ( connect ( sockfd , ( struct sockaddr *) & serv_addr , sizeof ( serv_addr )) < 0) error (
" ERROR connecting ");
3 // waypoint i
4 memset ( buffer ,0, sizeof ( buffer ));
5 strcpy ( buffer ," {\" MOVE \":{\" DIR_GPS \":\" latitud i,N, longitud i,E\" ,\" VELOCITY \":\"
slow \"}} ");
6 n = send ( sockfd , buffer , strlen ( buffer ), 0);
7 printf (" %s\n", buffer );
```

La Tabla 5-40 muestra la línea de comandos almacenada en el *buffer*, que hace referencia a la operación de despliegue aterrizaje.

Tabla 5-40. Función *socket* LAND. Elaboración propia, 2016.

```
1 sockfd = socket ( AF_INET , SOCK_STREAM , 0);
2 if ( connect ( sockfd , ( struct sockaddr *) & serv_addr , sizeof ( serv_addr )) < 0) error (
" ERROR connecting ");
3 memset ( buffer ,0, sizeof ( buffer ));
4 strcpy ( buffer ," {\" STATE \":{\" OPC \":\" land \", \" DELAY \":0}} ");
5 n = send ( sockfd , buffer , strlen ( buffer ), 0);
6 printf (" %s\n", buffer );
```

En base a lo visto en el Capítulo 2.6, se desarrolla el cliente socket, el cual recibe como argumento la IP del servidor y el puerto de escucha, para posteriormente, ejecutar las órdenes *take off, move y land*, las cuales son base para realizar un recorrido de exploración a través de los *waypoints*. En el ejemplo siguiente, se puede apreciar que en principio, hay un despegue a través de la función *take off*, luego se avanza hacia los objetivos a través de la función *move*, luego el último *waypoint* es un retorno a la estación base, para luego finalizar con el aterrizaje a través de *land*.

```
7 #include <sys/types.h>
8 #include <sys/socket.h>
9 #include <netinet/in.h>
10 #include <netdb.h>
11
12 void error(const char *msg) {
13     perror(msg);
14     exit(0);
15 }
16
17 int main(int argc, char *argv[]) {
18     int sockfd, portno, n;
19     struct sockaddr_in serv_addr;
20     struct hostent *server;
21
22     char buffer[256];
23     if (argc < 3) {
24         //fprintf(stderr, "usage %s hostname port\n", argv[0]);
25     printf("Ingresar ip y puerto");
26         exit(0);
27     }
28     portno = atoi(argv[2]);
29     sockfd = socket(AF_INET, SOCK_STREAM, 0);
30
```

```
31
32      if (sockfd < 0)
33          error("ERROR opening socket");
34      server = gethostbyname(argv[1]);
35      if (server == NULL) {
36          fprintf(stderr, "ERROR, no such host\n");
37          exit(0);
38      }
39      bzero((char *) &serv_addr, sizeof (serv_addr));
40      serv_addr.sin_family = AF_INET;
41      bcopy((char *) server->h_addr,
42              (char *) &serv_addr.sin_addr.s_addr,
43              server->h_length);
44      serv_addr.sin_port = htons(portno);
45
46
47        while (1) {
48
49          sockfd = socket(AF_INET, SOCK_STREAM, 0);
50          if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof (serv_addr
                ))) < 0) error("ERROR connecting");
51      memset(buffer,0,sizeof(buffer));
52      strcpy(buffer,"{\"STATE\":{\"OPC\":\"takeoff\", \"DELAY\":0},\"HEIGHT
            \":100}}");
53          n = send(sockfd, buffer, strlen(buffer), 0);
54          printf("%s\n", buffer);
```

```
55
56      /*En esta parte se configura el resto de la mision*/
57
58
59      close(sockfd);
60      break;
61
62
63
64
65      }
66      return 0
67  }
```

5.2. RESULTADOS EXPERIMENTALES

En base a lo visto en el capítulo anterior, se procede a experimentar con parámetros previamente definidos, como son la IP del servidor y el puerto de comunicación, para posteriormente, ejecutar las órdenes de comando *take off*, *move* y *land* a través del cliente de comandos.

Posteriormente, la experimentación se basa en una misión de exploración, para lo cual se define los *waypoints*, en una zona específica de la Región de Magallanes:

La Tabla 5-41, indica los *waypoints* utilizados para la realización de las pruebas en terreno.

Tabla 5-41. *Waypoints* pruebas en terreno Región de Magallanes y Antártica Chilena. Elaboración propia, 2016.

W1: 53°09'40.2"S 70°55'12.6"W	-53.161160, -70.920164
W2: 53°09'41.6"S 70°55'09.0"W	-53.161567, -70.919158
W3: 53°09'38.4"S 70°55'05.3"W	-53.160666, -70.918149
W4: 53°09'36.2"S 70°55'10.7"W	-53.160048, -70.919651
W5: 53°09'39.6"S 70°55'14.3"W	-53.160994, -70.920638

En el ejemplo siguiente, se puede apreciar que en principio, hay un despegue a través de la función *take off*, luego se avanza hacia los objetivos a través de la función *move*, luego el último *waypoint* es un retorno a la base estación, para luego finalizar con el aterrizaje a través de *land*.

Para probar la efectividad del protocolo, se define los parámetros y los *waypoints* en el código fuente del socket:

```c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(const char *msg) {
    perror(msg);
    exit(0);
}

int main(int argc, char *argv[]) {
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];
    if (argc < 3) {
        //fprintf(stderr, "usage %s hostname port\n", argv[0]);
    printf("Ingresar ip y puerto");
        exit(0);
    }
    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);


    if (sockfd < 0)
        error("ERROR opening socket");
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr, "ERROR, no such host\n");
```

```
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *) server->h_addr,
          (char *) &serv_addr.sin_addr.s_addr,
          server->h_length);
    serv_addr.sin_port = htons(portno);


      while (1) {

        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr
           )) < 0) error("ERROR connecting");
    memset(buffer,0,sizeof(buffer));
    strcpy(buffer,"{\"STATE\":{\"OPC\":\"takeoff\", \"DELAY\":0},\"HEIGHT
       \":100}}");
        n = send(sockfd, buffer, strlen(buffer), 0);
        printf("%s\n", buffer);


    sleep(5);

        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)
           ) < 0) error("ERROR connecting");

        //waypoint 1
        memset(buffer,0,sizeof(buffer));
    strcpy(buffer,"{\"MOVE\":{\"DIR_GPS\":\"-53.161160, -70.920164\",\"
       VELOCITY\":\"slow\"}}");
        n = send(sockfd, buffer, strlen(buffer), 0);
    printf("%s\n", buffer);

        //waypoint 2
        memset(buffer,0,sizeof(buffer));
    strcpy(buffer,"{\"MOVE\":{\"DIR_GPS\":\"-53.161567, -70.919158\",\"
       VELOCITY\":\"slow\"}}");
        n = send(sockfd, buffer, strlen(buffer), 0);
    printf("%s\n", buffer);


        //waypoint 3
        memset(buffer,0,sizeof(buffer));
    strcpy(buffer,"{\"MOVE\":{\"DIR_GPS\":\"-53.160666, -70.918149\",\"
       VELOCITY\":\"slow\"}}");
        n = send(sockfd, buffer, strlen(buffer), 0);
    printf("%s\n", buffer);
```

```
80          //waypoint 4
81          memset(buffer,0,sizeof(buffer));
82      strcpy(buffer,"{\"MOVE\":{\"DIR_GPS\":\"-53.160048, -70.919651\",\"
           VELOCITY\":\"slow\"}}");
83          n = send(sockfd, buffer, strlen(buffer), 0);
84      printf("%s\n", buffer);
85
86
87          //waypoint 5
88          memset(buffer,0,sizeof(buffer));
89      strcpy(buffer,"{\"MOVE\":{\"DIR_GPS\":\"-53.160994, -70.920638\",\"
           VELOCITY\":\"slow\"}}");
90          n = send(sockfd, buffer, strlen(buffer), 0);
91      printf("%s\n", buffer);
92
93
94          //waypoint 6, retorno
95          memset(buffer,0,sizeof(buffer));
96      strcpy(buffer,"{\"MOVE\":{\"DIR_GPS\":\"-53.161160, -70.920164\",\"
           VELOCITY\":\"slow\"}}");
97          n = send(sockfd, buffer, strlen(buffer), 0);
98      printf("%s\n", buffer);
99
100         //aterrizaje
101
102         sockfd = socket(AF_INET, SOCK_STREAM, 0);
103         if (connect(sockfd, (struct sockaddr *) &serv_addr, sizeof (serv_addr
               )) < 0) error("ERROR connecting");
104     memset(buffer,0,sizeof(buffer));
105     strcpy(buffer,"{\"STATE\":{\"OPC\":\"land\", \"DELAY\":0}}");
106         n = send(sockfd, buffer, strlen(buffer), 0);
107         printf("%s\n", buffer);
108
109
110
111
112     close(sockfd);
113     break;
114
115
116
117
118     }
119     return 0
120 }
```

Posteriormente se inician los nodos principales de la plataforma ROS (Tabla 5-42):

Tabla 5-42. Inicio nodos plataforma ROS para pruebas en terreno. Elaboración propia, 2016.

```
prime@prime□Vi r tualBox : ~ / ros commandloud$ ros launch
system command loud/ launch /command launch . launch
. . . l o g g ing to /home/prime / . r o s / l o g / c572ed8c□d283□11e6□a88a□0800272 e6b5d/
ros launch□prime□VirtualBox □2975. l o g
Checking l o g d i r e c t o r y for di sk usage . This may take awhi l e .
Pr e s s Ct r l□C to i n t e r r u p t
Done che cking l o g f i l e di sk usage . Usage i s <1GB.
s t a r t e d ros launch s e r v e r ht tp : //prime□Vi r tualBox :35830/
SUMMARY
========
PARAMETERS
* / r o s d i s t r o
* / r o s v e r s i o n

NODES
/
a r d r o n e d r i v e r ( a r d r o n e d r i v e r / a r d r o n e d r i v e r )
command control ( system command loud/ command control . py )
command socket ( system command loud/ command socket . py )

ROS MASTER URI=ht tp : // l o c a l h o s t :11311
cor e s e r v i c e [ / rosout ] found
pr o c e s s [ command socket □1] : s t a r t e d with pid [ 2 9 9 5 ]
pr o c e s s [ command control □2] : s t a r t e d with pid [ 2 9 9 6 ]
pr o c e s s [ a rdr one dr i v e r □3] : s t a r t e d with pid [ 2 9 9 7 ]
```

Se verifica que el nodo *command socket*, tiene el pid 2995, esto quiere decir, que el servicio se encuentra operativo para el paso de mensajes.

Luego se compila el socket cliente a través de gcc, y se ejecuta la orden:

- ./command_socket 127.0.0.1 5204

Al ejecutar esta orden, los AR. Drone comienza su recorrido sobre el cuadrante, estableciendo comunicación con ROS.

Posteriormente se genera un fallo arbitrario de corte de comunicación con servidor a través de telnet, a la IP del AR. Drone 192.168.1.101, lo permite por las características propias del dispositivo *Fligth Recorder*, que la máquina retorne a la estación base, de acuerdo a lo establecido en el protocolo.

CAPÍTULO 6: CONCLUSIONES

En este último capítulo, se abordan los resultados obtenidas al crear el protocolo de navegación entre UAVs, bajo arquitectura de comunicación ROS, para exploraciones de reconocimiento.

Al comienzo de este trabajo, se define la problemática de utilizar la tecnología para misiones de exploración, con los objetivos de retornar con información relevante ante cualquier tipo de contingencia, y de esta manera de evitar riesgos de accidentes de personal humano.

A continuación, se desarrollan las conclusiones para el protocolo de navegación entre UAVs que ha sido formalizado en el presente trabajo:

- Definir las características para exploraciones de reconocimiento, da una visión importante de lo significativo que puede ser la utilización de la tecnología en misiones de exploración arriesgadas, se expone el caso de expedición *Endurance* dirigida por Ernest Shacleton en 1914, su objetivo era atravesar la Antártica hasta Polo Sur, las pésimas condiciones hicieron que la embarcación quedara encallada en el hielo, no obstante los planes de contingencia y la experiencia de la tripulación ayudaron a que todos salieran con vida y fueran rescatados el 30 de agosto de 1916.

- Crear protocolo de navegación para UAVs, utilizando georreferenciación, con características propias de exploraciones reconocimiento. En este punto, una vez definidas las características de una exploración, se procede a crear un protocolo de navegación el cual se basa en explorar un territorio desconocido donde no se conocen las condiciones ni las variaciones del clima. Se definen cuatro etapas; apreciación física del territorio; planeación de vuelo; ejecución del vuelo. En esta etapa, si existiese pérdida de comunicación entre alguno de los exploradores, se da curso al protocolo secundario de retorno, donde los exploradores vuelven a la estación base principal; por último se define la etapa procesamiento de la información. Con la definición de estas cuatro etapas, se asegura retornar con la información relevante y evitar riesgo de personal humano.

  Por otra parte, pare efectos de este trabajo se utiliza un UAV Parrot 2.0, el cual utiliza un dispositivo GPS de tipo *Fligth Recorder*, también para efectos de pruebas, se utiliza

un *router* WIFI HUAWEI E5338. Con estos elementos se consigue interoperabilidad entre más de un UAV, modificando la seguridad el *router* y accesando vía ftp al UAV.

- Crear servidor de paso de mensajes utilizando plataforma ROS. El servidor de paso de mensajes permite la comunicación con las máquinas, se utiliza ROS, puesto que es una tecnología relativamente nueva, donde al igual que Linux, posee una gran comunidad de desarrolladores, que permiten cada vez más, generar más controladores para diferentes tipos de robots.

- Crear arquitectura de comunicación entre UAVs Parrot AR.DRONE 2.0 y plataforma de comunicación ROS. A través del controlador de vuelo *Ardrone Autonomy*, la plataforma ROS y el GPS *Fligth Recorder*, se crea una arquitectura de comunicación que permite el paso de mensajes entre los nodos plataforma descritos en el Capítulo 4.

- Crear cliente de comandos *socket stream*. El cliente de comandos, nos permite comunicar con el servidor ROS a través de un puerto asignado, luego el servidor replica las órdenes a las máquinas para establecer la misión. El cliente de comandos del presente trabajo se desarrolla en lenguaje C, donde se instancian las órdenes de *take off*, *move* y *land*, las cuales son base para realizar un recorrido de exploración a través de los *waypoints*.

- Validar protocolo de navegación, mediante pruebas de simulación y en terreno para vuelos de exploración. Para validar el protocolo de navegación, se realizan pruebas que permiten experimentar con parámetros previamente definidos como la IP del servidor, puerto de comunicación y órdenes de comandos. Posterior a eso se definen *waypoints* en una zona específica.

Para validar el protocolo de navegación, en primer lugar se inicializan los nodos plataforma que permiten dejar operativo el servidor de paso de mensajes (Tabla 6-43):

Tabla 6-43. Pruebas de Inicio nodos plataforma ROS . Elaboración propia, 2016.

```
prime@prime□Vi r tualBox : ~ / ros commandloud$ ros launch
system command loud/ launch /command launch . launch
. . . l o g g ing to /home/prime / . r o s / l o g / c572ed8c□d283□11e6□a88a□0800272 e6b5d/
ros launch□prime□VirtualBox □2975. l o g
Checking l o g d i r e c t o r y for di sk usage . This may take awhi l e .
Pr e s s Ct r l□C to i n t e r r u p t
Done che cking l o g f i l e di sk usage . Usage i s <1GB.
s t a r t e d ros launch s e r v e r ht tp : //prime□Vi r tualBox :35830/
SUMMARY
========
PARAMETERS
* / r o s d i s t r o
* / r o s v e r s i o n

NODES
/
a r d r o n e d r i v e r ( a r d r o n e d r i v e r / a r d r o n e d r i v e r )
command control ( system command loud/ command control . py )
command socket ( system command loud/ command socket . py )

ROS MASTER URI=ht tp : // l o c a l h o s t :11311
cor e s e r v i c e [ / rosout ] found
pr o c e s s [ command socket □1] : s t a r t e d with pid [ 2 9 9 5 ]
pr o c e s s [ command control □2] : s t a r t e d with pid [ 2 9 9 6 ]
pr o c e s s [ a rdr one dr i v e r □3] : s t a r t e d with pid [ 2 9 9 7 ]
```

Ejecución del socket desde la *Shell:*

- ./command_socket 127.0.0.1 5204

Previamente se encuentran definidos los *waypoints* (Tabla 6-44)*:*

Tabla 6-44. *Waypoints* definidos par validación de protocolo. Elaboración propia, 2016.

W1: 53°09'40.2"S 70°55'12.6"W	-53.161160, -70.920164
W2: 53°09'41.6"S 70°55'09.0"W	-53.161567, -70.919158
W3: 53°09'38.4"S 70°55'05.3"W	-53.160666, -70.918149
W4: 53°09'36.2"S 70°55'10.7"W	-53.160048, -70.919651
W5: 53°09'39.6"S 70°55'14.3"W	-53.160994, -70.920638

Posteriormente se genera un fallo arbitrario de corte de comunicación con el servidor a través de telnet, a la IP de AR. Drone 192.168.1.101 (Tabla 6-45):

Tabla 6-45. Pruebas fallo arbitrario a través de telnet AR.Drone 2.0. Elaboración propia, 2016.

```
antizun@PRIMEn# t e l n e t 1 9 2 . 1 6 8 . 1 . 1
Trying 1 9 2 . 1 6 8 . 1 . 1 . . .
Connected to 1 9 2 . 1 6 8 . 1 . 1 .
Escape cha r a c t e r i s '\^]' .
BusyBox v1 . 1 4 . 0 ( ) bui l t□in s h e l l ( ash )
Enter 'help' for a l i s t o f bui l t□in commands .
k i l l a l l udhcpd
i f c o n f i g ath0 down
```

Por las características del dispositivo *Fligth Recorder*, de acuerdo a lo establecido se retorna a la estación base, definida en el *waypoints* primario.

Finalmente, se cumple el protocolo de navegación entre UAVs, para una misión de exploración específica utilizando los fundamentos planteados en los objetivos, como lo son la utilización de la plataforma ROS y el cliente de comandos *socket stream*.

REFERENCIAS BIBLIOGRÁFICAS

Austin, R. (2010). UNMANNED AIRCRAFT SYSTEMS UAVS DESIGN, DEVELOPMENT AND DEPLOYMENT.

BracaMan (2009). Programación en socket. http://es.tldp.org/Tutoriales/PROG SOCKETS/prog-sockets.html.

De la Nuez, D. (2014). Exploración de la Antártica. www.smithsonianmag.com/history/reliving-shackletons-epic-endurance-expedition-102707360/.

Flyability (2014). Proyecto Elios. http://www.yability.com/elios/.

Martínez, Q. (2015). Generación de Mapas de Alta Resolución con Drones. sg.com.mx/revista/47/generacion-mapas-alta-resolucion-drones.WGfxrnUrI8o.

Piskorski, S. (2012). AR.Drone Developer Guide.

Qgroundcontrol (2010). MAVLink is a very lightweight, header-only message marshalling library for micro air vehicles. . http://qgroundcontrol.org/mavlink/start.

ros.org (2012). The Robot Operating System. http://www.ros.org/is-ros-for-me/.

Sisar (2014). Indices de vigor. http://ruas.cl/sisar/.