S.N.A.P.A

"SUPERVISION, NAVIGATION AND PLANNING ARCHITECTURE"

ARQUITECTURA DE NAVEGACIÓN, PLANIFICACIÓN Y NAVEGACIÓN PARA UN DIRIGIBLE NO TRIPULADO

LUIS IGNACIO LOPERA GONZÁLEZ

UNIVERSIDAD DE LOS ANDES
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA
BOGOTÁ
2005

S.N.A.P.A

"SUPERVISION, NAVIGATION AND PLANNING ARCHITECTURE"

ARQUITECTURA DE NAVEGACIÓN, PLANIFICACIÓN Y NAVEGACIÓN PARA UN DIRIGIBLE NO TRIPULADO

LUIS IGNACIO LOPERA GONZÁLEZ
Código: 200327289

ASESOR:

ALAIN GAUTHIER SALAS, Ph.D.

COASESOR:

LEONARDO SOLAQUE, MSc.

UNIVERSIDAD DE LOS ANDES
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA
BOGOTÁ
2005

TABLA DE CONTENIDOS

LISTA DE FIGURAS

AGRADECIMIENTOS

Este es la culminación de una etapa más de mi vida académica. Agradezco a la Universidad la posibilidad de haber trabajado con ellos en diferentes aspectos ayudándome a mejorar mis calidades como persona y como ingeniero, al profesor Alain Gauthier por su profunda confianza en mi y a mis locas ideas aunque estoy seguro de que más de una no me la entendió.

A los compañeros de proyecto a quienes sé que mis ideas molestaron, a veces.

A mis padres que siempre han sido un apoyo importante y se aguantaron la segunda tesis.

1 INTRODUCCIÓN

Los problemas de planificación, navegación y supervisión han sido tratados y resueltos para una gran variedad de aplicaciones. Sin embargo, no se han hecho muchos esfuerzos por tratar de una manera sistémica los tres problemas.

Estas tres tareas son fundamentales en la implementación de vehículos autónomos, en la actualidad se resuelve el problema para unas condiciones o restricciones específicas; acarreando una serie de problemas en el momento en el que el autómata se ve forzado a abandonar el terreno para el cual fue programado,

Este trabajo hace una primera aproximación para tratar de reunir los tres aspectos en una sola arquitectura compacta, dinámica y fácilmente replicable.

El trabajo se desarrolla con las restricciones y características de un dirigible, sin embargo se diseña de tal manera que pueda ser fácilmente exportable a otras plataformas móviles.

2 ANÁLISIS DEL PROBLEMA

Este capitulo presenta los aspectos considerados para realizar el diseño de la arquitectura.

2.1 LA PLATAFORMA

La aeronave objetivo de este proyecto es un dirigible de 10 m de largo con un diámetro máximo de 2 m. Posee dos motores vectorizables a gasolina, un motor eléctrico en el timón, una CPU de abordo, un GPS y una brújula electrónica. Estas capacidades hacen que el dirigible sea no-holonómico ya que le es imposible desplazarse lateralmente y en reversa.

El dirigible es un sistema MIMO y tiene un modelo no lineal bastante complejo el cuál no hace fácil la tarea de desarrollar controladores, sin embargo este se puede reducir a 3 modelos [2]: vuelo aerostático, vuelo aerodinámico y vuelo de crucero; dependiendo el punto de operación en el que se encuentre el dirigible.

2.2 LAS ETAPAS DE VUELO

El vuelo de una aeronave puede ser divido en tres etapas principales:

- Despegue.
- Vuelo crucero.
- Aterrizaje.

El despegue va desde el momento en el que el dirigible está con velocidad 0 m/s y alcanza su velocidad de crucero. El vuelo crucero comprende las traslaciones del

dirigible a una velocidad constante lo suficientemente alta para generar elevación aerodinámica. El aterrizaje es la desaceleración del dirigible desde la velocidad crucero hasta 0 m/s, también comprende la cuidadosa ubicación espacial del dirigible para evitar estrellarlo contra el piso u objetos aledaños al punto de aterrizaje.

Por las razones que se exponen en el numeral 2.4, aparecen 3 etapas adicionales las cuales son: la etapa de aproximación, de ejecución y de regreso. Estas etapas son divisiones de la etapa de vuelo crucero, aunque la etapa de ejecución puede hacer que el dirigible entre en diferentes modos de vuelo.

La etapa de aproximación es la etapa comprendida entre el momento que se termina el despegue y el sitio donde se inicia la misión. La etapa de Regreso es la complementaria a la etapa de aproximación, va desde el momento en el que se determina que se ha completado la etapa de ejecución hasta el momento del aterrizaje. La etapa de ejecución es donde se lleva a cabo la misión, cada vuelo debe ser organizado como una consecución de objetivos, y cada vuelo debe tener una actividad específica a realizar, la misión.

2.3 LA ESTRATEGIA GENERAL DE CONTROL

A cada etapa principal se le puede asociar un modelo simplificado del dirigible. Si utilizamos estos modelos, se pueden desarrollar controladores más fácilmente, específicos para el estado de operación. La utilización de controladores por etapas hace inminente la utilización de la estrategia de control programad *"Scheduled*

Control". Esta estrategia nos permite darle más libertad al sistema ya que un control que sea diseñado para todos los puntos de operación del dirigible, tiende a ser muy conservador [6].

2.4 LAS GENERALIDADES DE LA PLANIFICACIÓN Y LA SUPERVISIÓN

La forma de planificar cambia dependiendo del problema, por ejemplo no es lo mismo planificar para un espacio abierto sin obstáculos que para un bosque. Así la supervisión también puede tener varios niveles de exigencia [1]. Esta idea da origen a las etapas de aproximación, ejecución y retorno.

Para lograr la verdadera autonomía hay que lograr que un sistema sea capaz de cambiar de ambiente, esto no puede ser más cierto que para vehículos aéreos no tripulados. Dependiendo de la aplicación que se le quiera dar al dirigible, este puede encontrar la necesidad de poder reaccionar a objetos fijos o móviles. La manera de reaccionar está determinada por el supervisor. Pero así como puede necesitar esta capacidad, puede que esté volando en un ambiente despejado de objetos donde no es crucial la precisión de lo puntos de navegación intermedios si no el final de la trayectoria, para esto un supervisor liviano puede ser el más indicado.

Esta variedad de planificadores y de supervisores para los diferentes momentos del vuelo hace necesario poder tener varios algoritmos que solucionen los diferentes problemas.

En términos generales la planificación es una tarea que se ejecuta off-line, y no tiene que ser realizada en la computadora del dirigible. Dependiendo del algoritmo seleccionado para planificar un sector determinado, puede suceder que se necesiten muchos más recurso que los que los que dispone la CPU del dirigible.

En comparación los algoritmos de supervisión necesariamente son on-line, esto los obliga a depender de los recursos del computador y deben ser muy eficiente en el manejote estos, ya que tienen que ser compartidos por diferentes componentes del dirigible.

La planificación debe calcular unas referencias para que el supervisor siga, y este a su ves las debe pasar a los controladores o tomar las medidas necesarias para que el plan de vuelo sea ejecutado. Sin embargo existen ocasiones donde todo sale mal y el supervisor no es capaz de hacer cumplir los objetivos planificados, en este momento el supervisor debe llamar nuevamente al planificador para tratar de solucionar el problema. Aún así el nuevo plan puede resultar malo, entonces el supervisor debe decidir que hacer en ese momento.

Como consecuencia de esta situación se crean los algoritmos de escape, diseñados para aterrizar el dirigible en caso de que algo o todo salga mal y es función de la arquitectura ver que dadas las circunstancias estos algoritmos sean ejecutados.

3 DISEÑO DE LA ARQUITECTURA

En este capitulo se muestra el diseño de la arquitectura con las ventajas ideológicas que este presenta. Las características que debe cumplir la arquitectura son:

- Dividir tareas eficientemente.
- Permitir realizar tareas en paralelo.
- Debe permitir versatilidad en el intercambio de tareas.
- Debe tener memoria de la forma en que se hacen las cosas.
- Debe permitir la fácil administración de los algoritmos y almacenar información relevante.

La arquitectura debe encapsular los niveles elevados del lazo de control, generando las referencias para los controladores. Gráficamente se vería así:

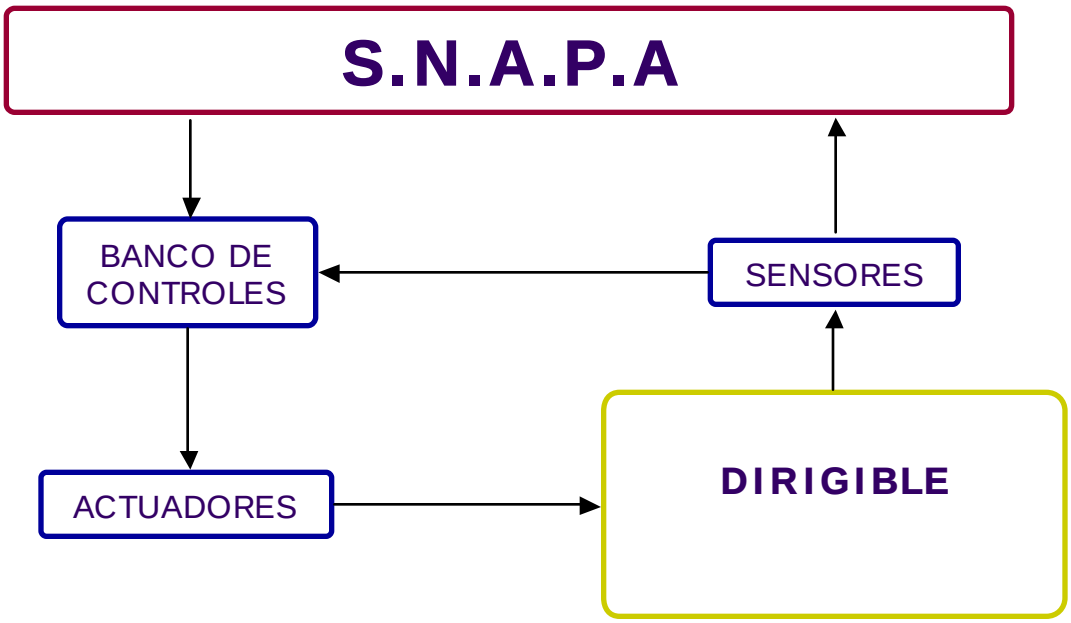

Figura 1: Esquema general

3.1 LA DEFINICIÓN DE MISIÓN

Para que la planeación la supervisión y la navegación se puedan estructurar de manera ordenada es crucial la definición de lo que es una misión.

Definición 1 Misión:

Una misión es el conjunto de objetivos divididos por etapas de vuelo del dirigible, expresados en posiciones espaciales, o requerimientos de tiempo, o de velocidad. Debe comenzar con la indicación de los objetivos del despegue y terminar con los del aterrizaje. También debe contemplar la existencia de objetivos adicionales a aquellos ya mencionados.

Esta definición es basada informalmente en la manera en la que los militares organizan sus misiones. Al terminar la creación de la misión cada etapa de vuelo debe tener asociado un algoritmo de planificación y uno de supervisión. Además se debe asignar el orden en el que se deben ejecutar las maniobras de escape (*Bail Out Maneuvers*).

3.2 LA DIVISIÓN DE TAREAS

Como se explicaba anteriormente la planificación se hace *off-line*, esto implica que no es necesario planificar dentro del dirigible. Hay que recordar que los recursos de la CPU de abordo son limitados. Teniendo esto en cuenta, y aprovechando la estructura utilizada en [1][7], se tiene que la CPU de la estación base es el lugar idóneo para realizar la planificación y la simulación.

La supervisión si es necesario hacerla abordo para no depender de un cordón umbilical (inclusive si es inalámbrico) entre la estación base y el dirigible. En caso de perder contacto con la estación base el supervisor debe estar en capacidad de operar. Esto genera la siguiente estructura:

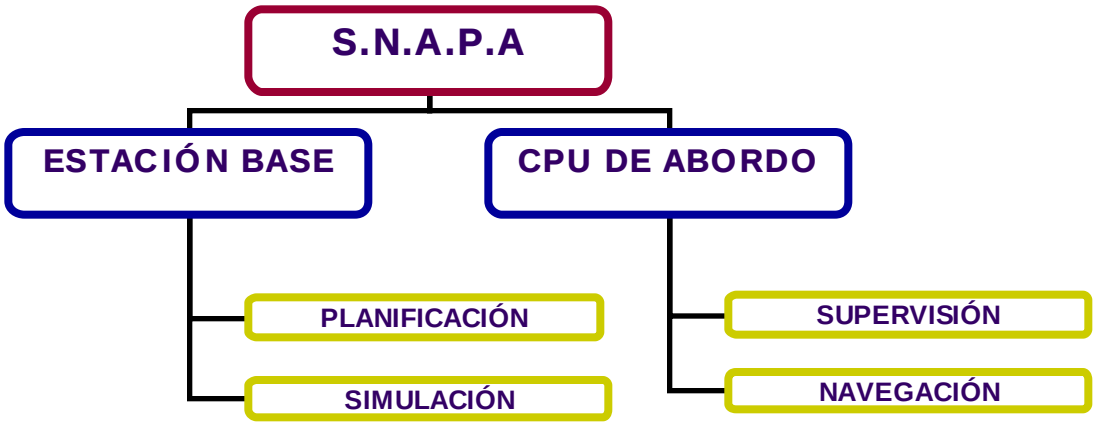

Figura 2: Esquema de la división de tareas

3.3 LA SIMULTANEIDAD DE TAREAS Y LA FACILIDAD DE INTERCAMBIARLAS

En el momento de la planificar es evidente que cada etapa es planificada independientemente. A cada etapa se le asigna un algoritmo distinto el cuál se encarga de cumplir los objetivos parciales y sobre todo los objetivos de comienzo y fin. Entonces no es necesario ejecutar los algoritmos consecutivamente, estos se deben poder ejecutar en paralelo para ahorrar tiempo. En el capitulo de resultados se demuestra el ahorro que se consigue al planificar en paralelo.

Para la supervisión no es correcto decir que se hacen todas las etapas al tiempo, porque el dirigible solo se encuentra en una etapa a la vez. Sin embargo el supervisor puede ejecutar varias tareas diferentes a la de supervisión simultáneamente, por ejemplo: puede ejecutar un controlador, puede extraer información de sensores diferentes a los necesarios para el control (imágenes de cámara). Y el supervisor estaría en la capacidad de detenerlos cuando estos ya no fueran requeridos para optimizar la utilización de los recursos.

Por lo anterior es evidente lo importante que es permitir la multiplicidad de tareas tanto en la CPU de abordo como en la estación base. La mejor forma de lograr esta multiplicidad es utilizando *threads* de ejecución.

Un *thread* es una porción del programa que es atendido como si fuera un proceso independiente. Como es una porción del programa es mucho más fácil de replicar que un proceso completo y consume menos recursos. El truco del asunto es lograr

que cada *thread* sea replicado con un algoritmo distinto para poder lograr la multiplicidad en las tareas. Para lograr esto sin necesidad de recompilar todo el programa, se hace que el *thread* llame a una función particular de una librería, así lo que cambia es la implementación de la función. Es decir solo hay que compilar la librería no todo el programa.

La utilización de threads también permite el fácil intercambio de tareas, cuando el administrador determina que una tarea está por culminar este puede generar un nuevo thread. Así cuando el primero termine, el nuevo ya está listo para iniciar su ejecución. Sin perder tiempo valioso en el intercambio de las tareas.

3.4 DE LA MEMORIA DE CÓMO HACER LAS COSAS, LA ADMINISTRACIÓN DE ALGORITMOS Y DE LA INFORMACIÓN RELEVANTE

Para tener una memoria de la forma de hacer las cosas solo se necesita que cada nueva librería se almacenada en lugar determinado del disco duro tanto para la CPU de abordo como la estación base. Sin embargo la administración de archivos de los sistemas operativos no permite almacenar información relevante adicional a la librería.

Se considera entonces la facilidad de almacenar en una base de datos. Estas por diseño están hechas para administrar eficientemente información de todo tipo. Como consecuencia, los algoritmos son almacenados en la base de datos, resolviendo el problema de la memoria, puesto que allí se pueden almacenar ordenadamente muchos algoritmos para las diferentes etapas de vuelo. Adicionalmente resuelve el problema de almacenar la información relevante para

cada uno de estos y permite una fácil administración de los diferentes algoritmos y su información.

La estructura de la base de datos para ambas plataformas es la siguiente:

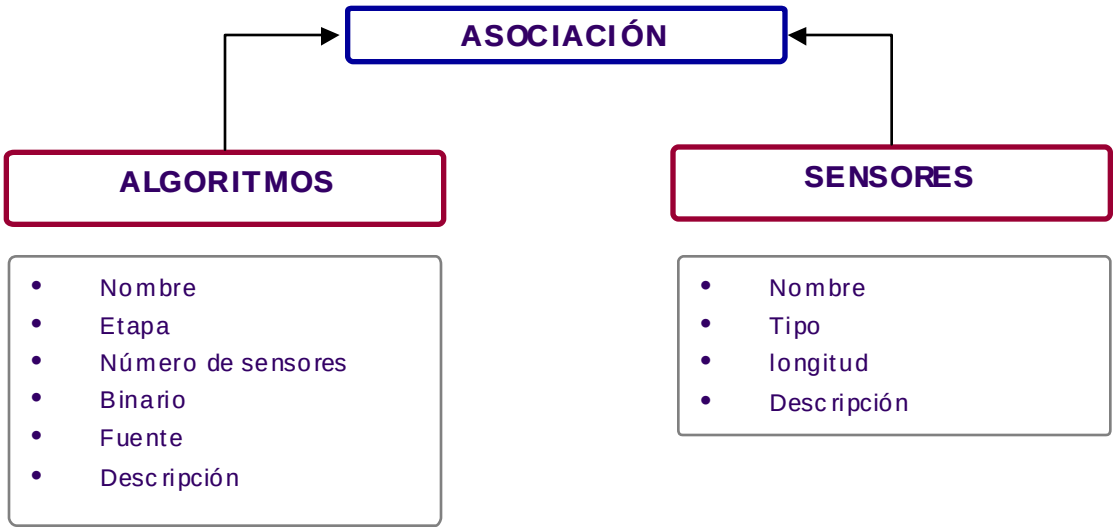

Figura 3: Estructura de tablas de la base de datos.

La base de datos relaciona la tabla de sensores que puede resultar relevante en el momento del diseño de las diferentes librerías. Crea un camino único de comunicación entre los sensores y los algoritmos, facilitando el desarrollo de las librerías porque no se deben preocupar por capturar la información de los sensores sino que estos estarán procesados y listos a ser utilizados por el algoritmo.

3.5 DEL ADMINISTRADOR, LAS COMUNICACIONES Y LOS ESQUEMAS PARCIALES

En la CPU de abordo existe un Administrador cuya función es controlar el momento en el que se cambia de estado y traer el algoritmo indicado. En otras palabras se

encarga de cumplir la misión programada por el usuario, tanto en simulación como en vuelo. En términos de la arquitectura de control el administrador está por encima del supervisor y se asegura de que sea el correcto para la etapa en la que se encuentra. En a estación base el administrador es controlado por el usuario para determinar la manera en la que se ejecuta la planeación (paralelo o secuencial).

En ambas plataformas existe un módulo de comunicación, el cuál se encarga de llevar los datos al lugar indicado. En el caso de la estación base el módulo se encarga de llevar los resultados del planificador al simulador y viceversa. En el caso de la CPU de abordo este módulo se encarga de enviar la información a los controladores o a los actuadores y leer la información de los sensores.

A continuación se ilustran los esquemas parciales:

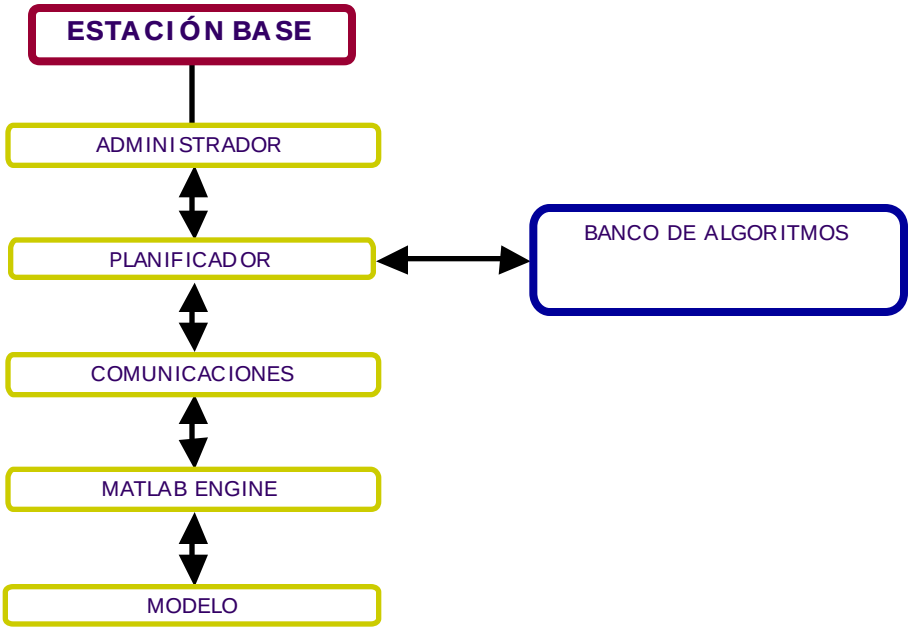

Figura 4: Esquema parcial estación base.

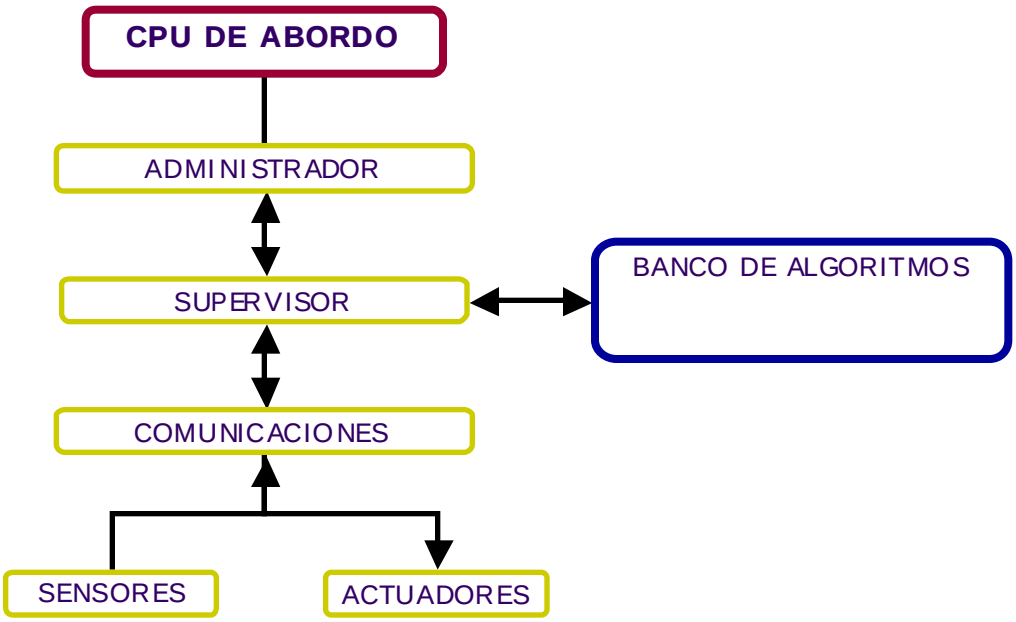

Figura 5: Esquema parcial CPU de abordo

4 IMPLEMENTACIÓN DE LA ARQUITECTURA

En este capítulo se explica la forma en la que se implementó la arquitectura anteriormente descrita.

La arquitectura está desarrollada en C++, se busca sacar provecho de todas las herramientas de desarrollo de software y de todas las construcciones lógicas y funcionales que, hasta el momento, no han sido explotadas fuertemente en el desarrollo de sistemas autónomos.

Tras un análisis de la posibles maneras en las que se podría implementar la arquitectura, aparece de manera evidente, que ambas plataformas estructuralmente hablando son idénticas. Es decir que si se programa el *thread* de tal manera que no sea necesario recompilar el programa cada vez que se desea cambiar de función, y se tiene una manera de saber si la función es on-line o off-line, el mismo *thread* sirve para supervisión y para planificación.

4.1 DE LA ESTRUCTURA DEL THREAD

El *thread* se encapsuló en una clase la cual se encarga de facilitar la creación de instancias de la misma. Adicionalmente se programaron una serie de funcione que permiten cargar los algoritmos de la base de datos.

Para que no sea necesario volver a compilar todo el programa se estandarizó la función que llama el thread cada vez que se quiere ejecutar el algoritmo. Y para solucionar de una vez el problema de saber si la función es on-line u off-line se escogieron dos funciones :

- **void** _ PlanningAlgorithm(…)
- **bool** _ PlanningAlgorithmOL(…)

El *thread* cuando carga la librería dinámicamente busca las funciones si encuentra la primera función determina que el algoritmo off-line y si encuentra la segunda determina que es on-line. Es importante que en una misma librería no se encuentren ambas versiones ya que solo se ejecutará la versión en línea debido al orden de búsqueda de la funciones. El encabezado del *thread* luce así:

```
class SPThread : public TThread
{
private:

protected:
    void __fastcall Execute();
    double start_time;
    double end_time;
    int state;
    double wind_dir_ant[6];
    double slope[3];
    double m[3];
    void update_win_dir_accel();
    int accel_type;
public:

    double exeTime(){return end_time-start_time;};
    double StartTime(){return start_time;};
    double EndTime(){return end_time;};
    AnsiString OutPut_File;
    AnsiString GetState();
    int Stage;
    bool OnLine;
    bool needsmap;
    bool isMyLibrary;          //did I created the library
    bool Shared;               //share the matlab engine
    TMission    *mission;
    DBelement   *DB_connection;
    void        *data;
```

```
    MatEngine  *ep;
    PlanningAlgorithm Alg;          // Función de algoritmo cargada dinámicamente offline
    PlanningAlgorithmOL AlgOL; // Función de algoritmo cargada dinámicamente on line
    double wind_dir[6];             //[u,v,w,u',v',w']
    double wind_dir_exp[3];         //Final value of win dir;
    void SetAccelType(int acc);
    double wind_accel_window;   //Time it takes to change from wind condition A to B
    bool exceptioned;
    double InitialState[12];
    double InitialPos[9];
    double SampleTime;
    HMODULE CurrentLibrary;
    AnsiString CurrentDllname;
    bool completeLoadDll();
    bool isTerminated(){return Terminated;};
    __fastcall ~SPThread();
    __fastcall SPThread(bool CreateSuspended);
};
```

Como se puede ver en el encabezado hay una serie de variables que sirven para calcular las estadísticas de operación, cuanto se demoró ejecutando el *thread.* También hay variables de estado que permiten saber al administrador en que etapa está el thread (cargando, simulando, escribiendo resultados, detenida, o finalizada). Se incorporó las variables de dirección y velocidad del viento para que desde el GUI el usuario pueda cambiar sus condiciones en tiempo de simulación y así pueda realizarse verificaciones más reales. Se incorporó unos parámetros de aceleración para crear variaciones en la velocidad del viento de diferentes maneras.

La función completeLoadDll() busca en la misión almacenada en la variable *mission* el *Id* del algoritmo asociado a la etapa almacenada en la variable *Stage.* Crea un archivo con el nombre del algoritmo y descarga el binario de la base de datos a este archivo. Luego lo carga en memoria y asocia la función _ PlanningAlgorithm(..) o _PlanningAlgorithmOL(…) a las variable Alg o AlgOL respectivamente y

dependiendo de la que encuentre se algo llega a fallar retorna falso y pone la bandera de exceptioned en verdadero.

La variable ep es la variable que maneja el motor de Matlab este será explicado más adelante. Esta variable en la versión on broard del thread desaparece y es remplazada por un apuntador a las instancias GeN^{om} de los sensores y actuadores las cuales son conectadas dinámicamente dependiendo los sensores que el algoritmo necesite. Esta información se encuentra en la base de datos.

4.2 DE LA ESTRUCTURA DE LA MISIÓN

La misión está encapsulada en la clase TMission. Esta clase presenta toda la estructura necesaria para almacenar, de acuerdo a la definición 1, todos los componentes de la misión. El encabezado de la clase es el siguiente:

```
class TMission
{
private:
   void SetBailoutOrder(int Index,int );
   int* GetBailoutOrder(int Index);
   int msBailoutOrder[MAX_BAILOUT_MANUVERS];
   int FindPos(int Value);
   AnsiString mName;
public:
   double (*Stage1Obj)[8];
   double (*Stage2Obj)[8];
   double (*Stage3Obj)[8];
   double (*Stage4Obj)[8];
   double (*Stage5Obj)[8];
   int NumOfObjSt1;
   int NumOfObjSt2;
   int NumOfObjSt3;
   int NumOfObjSt4;
   int NumOfObjSt5;
   double (* getStObjs(int Stage))[8];
   int getNumStObjs(int Stage);
   int Stage1AlgId;
   int Stage2AlgId;
   int Stage3AlgId;
   int Stage4AlgId;
```

```
    int Stage5AlgId;
    int getStageAlgId(int stage);
    int Units;
    int FrameofReference;
    TStringList *FligthObjs;
    __property AnsiString Name={read=mName,write=mName};
    __property int BailoutMMDOrder[int Index]={write=SetBailoutOrder,read=GetBailoutOrder};
    TMission();
    ~TMission(){};
    bool ConvertObjs();
    void SaveMission();
    void LoadMission(AnsiString filename);
};
```

Se puede observar que la clase posee un campo para asociar un nombre a la misión. Da la posibilidad de guardar la misión en un archivo, lo cuál, en esta versión, es necesario para poder pasar la información al CPU del dirigible. Asigna el orden de maniobras de escape. Indica para cada etapa de la misión cual algoritmo utilizar y posee unas variables para manejar los objetivos como cadenas de texto o como vectores numéricos. Adicional a esto presenta la variable Units para indicar las unidades de los objetivos y FrameofReference que es el marco de referencia con respecto al cuál fueron ingresados.

4.3 DEL SIMULADOR

El simulador está montado utilizando el motor de MATLAB, este se utiliza llamando a una serie de librerías. Para que los desarrolladores no tuvieran que lidiar con este problema cada vez que quisieran utilizar MATLAB como parte de alguno de sus algoritmos se desarrollaron dos clases para encapsular el motor y facilitar su manejo.

La primera clase es la clase MatExeptions creada para manejar las excepciones generadas por errores en la clase principal, MatEngine. Esta última es la clase que

encapsula el motor y provee funciones que facilitan la importación y exportación de variables a MATLAB, y la ejecución de comandos en el motor. Los encabezados de las clases son los siguientes:

```
class MatExeptions {
private:
    AnsiString errstr;
public:
    MatExeptions(AnsiString err){errstr=err;};
    ~MatExeptions(){};
    AnsiString GetLastError(){return errstr;};
};

class MatEngine {
private:
    Engine * ep;
    AnsiString workspace;
    mxArray *matarray;
    int dims;               //number of dimensions
    const int *lengths;         //size of each diimension
public:
    AnsiString OutPut_Buffer;
    AnsiString Command_Buffer;
    AnsiString GetWorkspace();
    void SetWorkspace(AnsiString wrkspace);
    Engine *GetEngine();
    bool StepByStep;
    void executeCmd();
    void executeCmd(AnsiString cmd);
    MatEngine(bool Shared);
    ~MatEngine();
    bool getVariable(AnsiString name,void *value);
    bool getVariable(AnsiString name,void **value);
    bool getVariable(AnsiString name,void *value,int edims,int *dimSize);
    bool setVariable(AnsiString name,void *value,int edims,int *dimSize);
    int GetLength(int dim);

    bool equalslengths(int *);
    int calOffset(int *,int *);
    void incrementIT(int *,int pos);
    bool isOnvec(int *,int);
};
```

La ventaja de utilizar estas clases es que se puede utilizar el modelo del dirigible creado en MATLAB, sin necesidad de volver codificar todo el modelo en lenguajes menos amigables como C o C++, además si se necesitan hacer cambios se hacen en el .m y no es necesario volver a compilar todo el programa o las librerías que utilicen instancias del MatEngine.

4.4 DE LA BASE DE DATOS

La base de datos encapsula el API de MySQL para C++ adaptandolo a las necesidades de la arquitectura, implementa funciones para cargar y actualizar los algoritmos en la base de datos, copiarlos a disco duro, y asociar los sensores respectivos. Las clases para poder lograr esto son las siguientes:

```
class Sensor {
  friend class SensorList;
  private:
    Sensor *next;
    Sensor *prev;
  public:
    int id;
    string name;
    string type;
    int length;
    string description;
    Sensor(){next=NULL;prev=NULL;};
    ~Sensor(){};
};
```

Esta clase se encarga de manejar la información de un sensor, y replica la estructura de la tabla Sensors en la base de datos. Está preparada para que los sensores puedan ser asociados en lista de sensores de acuerdo a la tabla de asociación.

```
class Alg {
  public:
    int id;
    string name;
    string stage;
    int num_sens;
    string file_name;
    string source;
    string description;
    SensorList *sens_list;
    Alg(){sens_list=NULL;id=-1;};
    ~Alg(){};
};
```

Esta clase se encarga de manejar la información de un algoritmo y replica la estructura de la tabla Algorithms en la base de datos, el campo file_name se utiliza para saber el nombre del archivo que se va a cargar en el campo ALg_ Binary de la

base de datos. sens_list es un apuntador a la lista de sensores asociados al algoritmo.

```
//*****************************************************************
class AlgTree {
private:
    AlgTree *head;
    AlgTree *next;
    AlgTree *prev;
    AlgTree *traveler;

    AlgTree *findlast();
    void updatecount(int offset);
public:
    string text;
    int id;
    string stage;
    int count;
    int Number_of_children;
    AlgTree *children;
    AlgTree * operator [] (int i)
    {
        int j;
        traveler=this;
        for(j=0;j<i && j<this->count ; j++)
        {
            traveler=traveler->next;
        }
        return this->traveler;
    };
    void add(AlgTree *brother);
    void add(string brother,int id1,int st);
    void addchild(AlgTree *child);
    void addchild(string child,int id1,int st);
    void remove_children();
    AlgTree();
    ~AlgTree();
};
```

Esta clase se construyó con el único propósito de poder visualizar en el administrador de la base de datos de la aplicación todos los algoritmos y su información relevante.

```
//*****************************************************************
class DBelement {
private:
    Connection *con;

    Query *query;

    //Sensor sens1;
    int num_rows;
    int num_columns;
```

```
    bool connect();
public:
    DBelement();
    ~DBelement(){};
    Alg algorithm;
    Sensor sens;
    int rows() {return num_rows;};
    int columns() {return num_columns;};
    string Server;
    string Port;
    string Database;
    string User;
    string Password;
    bool connected();
    bool connected(bool action);
    bool getAlg(int alg_id);
    bool getAlg(const string alg_name);
    Sensor *  getSensor(int sens_id);
    Sensor *  getSensor(const string sens_name);
    bool UpdateAlg();
    bool UpdateSens(int sens_id);
    bool InsertAlg();
    bool InsertSens();
    bool WriteDll(int alg_id,string file_name);
    SensorList * getSensorList();
    AlgTree * filltree();
};
```

Esta clase es la clase que encapsula la comunicación con la base de datos, se puede ver que contiene la información relevante para la conexión y las funciones de manejo de la información necesarias para que la arquitectura funcione correctamente.

```
//*********************************************************************
class SensorList {
private:
  Sensor *sens;
  Sensor *traveler;
  int num_sens;
  Sensor *findlast();
public:
    Sensor *operator [](int i)
    {
        traveler=sens;
        for(int j=0;j<i && j<num_sens;j++)
        {
            traveler=traveler->next;
        }
        return traveler;
    };
    void    add(int sens_id,DBelement * BLIMP_ALG_DB);
    void    add(int sens_id,string name, int type, int length);
    void    remove(int sens_id);
    Sensor* search(int sens_id);
```

```
    Sensor* next();
    const int     count(){return num_sens;};
    SensorList();
    ~SensorList(){
       if (sens==NULL)
          return;
       while((traveler=findlast())!=sens)
       {
          traveler->prev->next=NULL;
          delete traveler;
       }
       delete sens;
    };
};
```

Esta clase se construyó con el propósito de poder administrar la asociación de sensores a un algoritmo específico. También permite listar todos los sensores almacenados en la base de datos.

4.5 DE LAS FUNCIONES _PlanningAlgorithm Y _PlanningAlgorithmOL

Estas funciones determinan el comportamiento del thread. Este mira el estado de la variable OnLine y ejecuta la función específica. Si es verdadero, el thread entra en un ciclo hasta que se le indica que se termine o hasta que la función PlanningAlgorithmOL retorna verdadero indicando que ya completo su ciclo, de lo contrario ejecuta la función PlanningAlgorithm una sola vez y cuando esta retorna pasa los datos al lugar indicado, en el caso de la estación base pasa el vector generado a el motor de MATLAB para que sean simulados.

Estas funciones deben entenderce como la función main() de un programa en DOS, e lsistema operativo busca esta función para ejecutar el programa, si el programa no posee dicha función no ejecuta. De manera similar pasa con las librerías desarrolladas para la arquitectura, si no poseen la función PlanningAlgorithmOL o la PlanningAlgorithm la librería no ejecuta y el algoritmo no corre.

Los encabezados de las funciones son los siguientes:

```
void PlanningAlgorithm (double Objectives[][8],
                        long ObjSize,
                        double *(Output[][9]),
                        long &Output_size,
                        double *SampleTime,
                        double *ControlSignal[5],
                        long &CntSize,
                        double **windir,
                        double    *InitialPos);

bool PlanningAlgorithmOL(double Objectives[][8],
                        long ObjSize,
                        int CurrentObj,
                        double Output[9],
                        double *SampleTime,
                        double *ControlSignal[5],
                        double *windir,
                        double    *CurrentPos,
                        double *CurrentState,
                        double elapsedtime); //hasta aquí versión estación base
                        void **sensors,
                        int sensorsdims[],
                        int sensorstypes[]);    // estos tres aparecen en la CPU de abordo
                                                   únicamente
```

Dada la facilidad de cargar algoritmos que brinda el thread y la facilidad de replicarse se pensó en la posibilidad de cargar algoritmos de control o algoritmos que planificaran no en referencias sino en las señales de control del dirigible. En consecuencia, adicional a la verificación del estado de la variable OnLine, el sistema verifica si ControlSignal==NULL para saber si el algoritmo genera señales de control o de referencia. Esto hace vital garantizar que la variable sea NULL cuando se generen referencias.

Hay que observar las sutiles diferencias de los algoritmos, hay que tener en cuenta que la versión Online solo ejecuta un paso y retorna para simular o ejecutar la acción determinada recibe los resultados del simulador o de los sensores y vuelve y ejecuta, por eso es importante manejar de manera adecuada el valor de retorno de la función, para que el thread de ejecución no sea terminado prematuramente.

El que el thread pueda determinar si el algoritmo sea on-line o off-line, y el hecho de que estructuralmente sean semejantes permite que la estación base sea capaz de simular algoritmos de supervisión y que la cpu de abordo sea capaz de ejecutar algoritmos de planificación. El interés de poder hacer esto en la cpu del dirigible es que el supervisoor puede determinar que le es imposible cumplir con el plan de vuelo trazado inicialmente, así que replica un thread con un planificador con la localización corregida y las condiciones de vuelo actualizadas para poder alcanzar los objetivos primarios.

4.6 LA FICHA TECNICA

Banco de algoritmos:	**MySQL**
Replicación de algoritmos:	**Threads**
Motor de simulación:	**MATLAB Engine**
Generador de módulos:	**GeN^{om}**
Sistema Operativo Estación Base:	**Windows XP**
Sistema Operativo de abordo:	**Linux Fedora**

Figura 6: Ficha Técnica

Se escogió Windows XP para la estación base ya que la interfaz gráfica es más familiar para los usuarios. Se escogió Linux para la CPU de abordo porque permite correr en modos operacionales completos pero reducidos, economizando en recursos. Esto implica que la versión de la arquitectura de abordo no posee interfaz gráfica.

5 RESULTADOS Y APLICACIONES

Aunque el rendimiento particular de un algoritmo cualquiera, no debe ser mejorado, si hay una gran mejoría del tiempo total de planificación, y en la administración de los diferentes recursos.

Asumamos por un momento que cada algoritmo tiene un tiempo de ejecución Ti, supongamos que este tiempo de ejecución se distribuye de la siguiente manera:

k_i:=Número de operaciones de procesador

l_i:=Número de accesos a memoria en unidades de operaciones del procesador

m_i:=Número de accesos a disco duro en unidades de operaciones del procesador

$$T_i = k_i + l_i + m_i$$

Ahora, un elemento de k utiliza un ciclo de procesamiento[1], un elemento de l utiliza 10 operaciones del procesador y un elemento de m utiliza 100 operaciones del procesador. Ahora, la tecnología *multithreading* permite esperar por la información de un *thread* mientras procesa las instrucciones de otro. Sea

$$T = T_1 + T_2 + T_3 + T_4 + T_5$$

$$T_{max} = 5 \max(T_i) \quad i := 1..5$$

[1] Valores promedio, simplemente por tener una idea de las magnitudes.

T_{max} es una cota superior del tiempo de ejecución, si asumimos que dada la distribución apropiada de k, l y m se puede ejecutar las 5 threads en $max(T_i)$ ciclos se tiene que en el mejor de los casos tendríamos :

$$T_{planeación} = max(T_i) = T_{max}/5$$

Y en el peor de los casos:

$$T_{planeación} = T$$

Esto muestra un ahorro sustancial de tiempo en el momento de planificar, lo que ratifica la importancia y los beneficios de utilizar Schedule control y *threads*.

El tiempo de desarrollo de algoritmos de planificación y supervisión se ve reducido por las herramientas que la arquitectura presenta al desarrollador, es decir que no se tiene que preocupar de tareas específicas, por ejemplo, la adquisición de los datos, ya que estos estarán a su disposición para cuando los necesite. Como consecuencia, trabajos futuros sobre el proyecto pueden ser desarrollados mucho más rápido, de tal manera que se pueda avanzar en los temas de interés.

La arquitectura permite implementar de manera similar cosas que en la teoría son totalmente diferentes, como por ejemplo: El supervisor pueda cargar los controladores de la base de datos, lo único que tiene que hacer es replicarse así mismo esta vez con el algoritmo de control, el seguirá supervisando y su copia

tomará el control del dirigible. Filosóficamente son dos entidades distintas y que no se pueden mezclar, pero la manera de implementar esa capacidad es igual, tanto para el supervisor como para el controlador, lo que hace la arquitectura supremamente flexible.

La arquitectura permite correr algoritmos de identificación, en este caso, se planea primero una serie de señales de control para aplicar al dirigible, luego el supervisor aplica esas señales y almacena los datos recogidos por los diferentes sensores, luego se toman los datos y se aplican en la estación base para realizar la identificación. Una versión alterna realiza el ajuste en vuelo, pero hay que tener cuidado con no ir a consumir todos los recursos del sistema causando una falla del sistema.

6 COSAS PARA MEJORAR LA ARQUITECTURA

La arquitectura da lugar para mejoras, adicional a la optimización de las funciones construidas, y la ampliación de las capacidades de las clases actuales, las siguientes son algunas tareas que podrían mejorar sustancialmente la arquitectura:

- Creación y codificación de un canal al nivel de la arquitectura par comunicar la estación base con la cpu de abordo.
- Un sistema inteligente para la selección del algoritmo más conveniente para la etapa o situación específica (evitar o complementar la entrada del usuario).

7 SISTEMA DE RECICLAJE DE HELIO PARA EL DIRIGIBLE

(Asesorado por MsC. Rafael Beltrán e Ing. Daniel Beltrán)

Debido a una escasez de recurso económicos del proyecto y una serie de dificultades técnicas y administrativas, no se pudo construir un hangar para el dirigible en los terrenos de la hacienda el noviciado, propiedad de la Universidad.

La imposibilidad de construir el hangar, trajo la necesidad de desinflar la carena del dirigible para su almacenamiento. El helio es un gas costoso y sus 30 m^3 hacían dispendioso su almacenamiento.

Se cotizaron varios compresores industriales los cuales venían a una presión estándar de 175 PSI, a esta presión se obtiene la siguiente capacidad en volumen.

$$VoPoTi = ViPiTo$$
$$To = Ti$$
$$(Vo + Vi)Po = ViPi$$
$$Vi = \frac{VoPo}{-Po + Pi}$$
$$Vi = \frac{30m^3 \times 10.2864}{175 - 10.2864}$$
$$Vi = 1.87m^3 \approx 2m^3$$

El costo del tanque de 2 m^3 se sale del presupuesto. Se propuso entonces el siguiente sistema cuyo valor asciende a los 5'000.000 de pesos, y consiste en: 2 compresores de 2HP y una presión máxima de 115 psi conectados en serie con un radiador entre ellos para alcanzar una presión de 390 psi con un tanque de 240 galones. El problema de esta solución es que demora alrededor de 4 horas para

desinflar el dirigible; Por esta razón se incorporó al diseño una serie de válvulas de paso para iniciar con los dos compresores en paralelo, y una vez alcanzados los 100 psi hacer la conexión en serie. Se estima que esto reduce el tiempo en una hora.

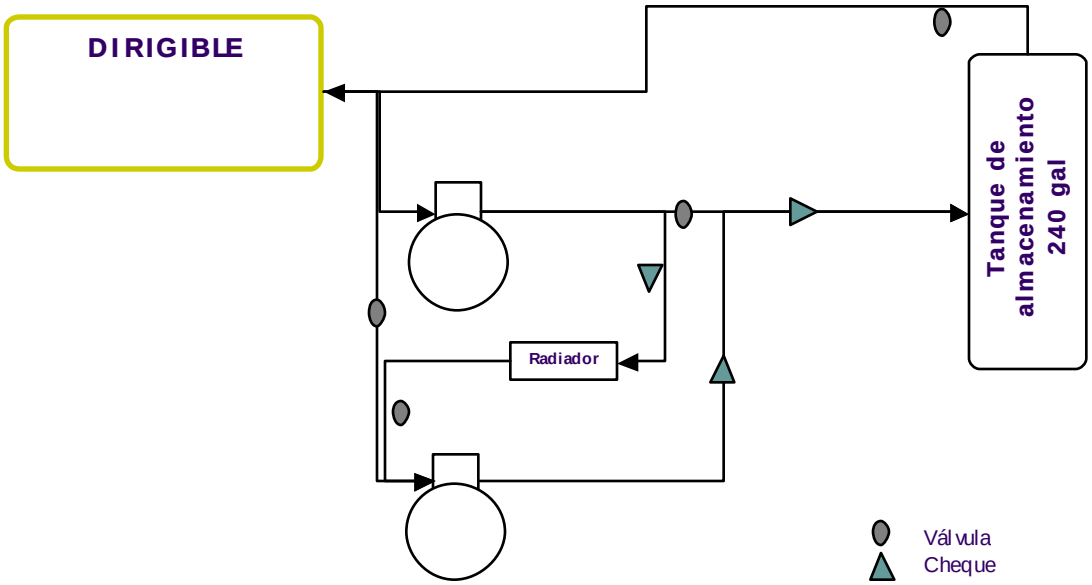

Figura 7: Esquema sistema de reciclaje de Helio

8 CONCLUSIONES

Aunque el rendimiento particular de un algoritmo cualquiera, no debe ser mejorado, si hay una gran mejoría del tiempo total de planificación, y en la administración de los diferentes recursos.

La arquitectura está diseñada para cumplir las siguientes tres tareas: planificar, supervisar y navegar. Dada su flexibilidad, se puede expandir a que tome el control completo del dirigible o utilice cualquier capacidad adicional que se le instale. Se espera que sea fácilmente transportable a otras plataformas.

Algunos experimentos realizados en simulación, muestran que el cambio de algoritmos no afecta la estabilidad del sistema, ya que la dinámica del dirigible es muy lenta y su inercia es considerable. Pero sin embargo es bueno tomar las directivas mencionadas en [6]

El manejo de la simultaneidad de las threads, resulta crucial para el correcto funcionamiento del sistema. Una mala replicación implica el desgaste de recursos y pérdida de resultados. Baja la confiabilidad de los datos obtenidos.

Si los futuros desarrolladores utilizan las interfaces construidas para trabajar con el motor de Matlab, MySQL y las clases construidas para la aplicación, obtendrán excelentes resultados en muy poco tiempo.

Hacer planificación por etapas sigue minimizando el tiempo de ejecución y la flexibilidad de resolver diferentes problemas que brinda la base de datos no se ve restringido aunque se implemente un único control lo suficientemente robusto para que no haya necesidad de cambiar de controladores y que este sea lo suficientemente flexible para no limitar la operabilidad del dirigible.

La arquitectura es un excelente complemento al sistema GeNom, porque permite sacar ventaja de la modularidad que este presta y al igual que GeNom la arquitectura mejora la etapa de desarrollo y facilita la reutilización de algoritmos (componentes).

BIBLIOGRAFÍA

[1] Raja Chatila, Simon Lacroix, Jérémie Gancet., EDEN PROJECT, LAAS, http://www.laas.fr/~simon/eden/index.php

[2] María Carolina Patiño Reyes, Diseño de un sistema de control de vuelo de un blimp. Universidad de los Andes, Febrero 2005.

[3] An Architecture for Autonomy, R. Alami, R. Chatila, S. Fleury, M. Ghallab, F. Ingrand LAAS-CNR http://www.laas.fr/~felix/publis/ijrr97/node9.html

[4] Software Architecture for Autonomous Perception Systems, Anthony Mallet, LAAS-CNRS, January 2003, http://www.laas.fr/~mallet/home//mallet-cnrs.pdf

[5] Linux Threads Frequently Asked Questions (FAQ)
by Sean Walton, KB7rfa walton@oclc.org (Last revised 21 Jan 1997) http://www.tldp.org/FAQ/Threads-FAQ/

[6] F. Wang and V. Balakrishnan, `` Improved Stability Analysis and Gain-Scheduled Controller Synthesis for Parameter-Dependent Systems". Submitted to the IEEE Trans. AC, September 1999

[7] Fusión sensorial y simulación para el direccionamiento de un dirigible. Ing. Diego Patiño, Trabajo de Tesis de Maestría, Universidad de los Andes, Febrero de 2005