UXAS DEVELOPERS

UXAS USER'S MANUAL

Contents

Introduction 7

Installation 9

Navigating 17

Services 19

Examples 51

Testing 71

Contributing 73

List of Figures

1.1 Notional Block Diagram 7

3.1 Open UxAS Folder 17
3.2 Open UxAS Folder Contents 17
3.3 Open UxAS Documentation Folder 17
3.4 Open UxAS Examples Folder 17
3.5 Open UxAS Message Data Models Folder 18
3.6 Open UxAS Resources Folder 18
3.7 Open UxAS Source Folder 18
3.8 Open UxAS Test Folder 18

4.1 UxAS Example Configuraion 20
4.2 Core Service Message Flow 22

5.1 AMASE simulation snapshot. 51
5.2 HelloWorld console messages. 55
5.3 The waterway to be monitored. 57
5.4 The message sequence flow diagram. 57
5.5 The Initialization message sequence flow diagram. 59
5.6 The LineSearchTask representing the waterway. 59
5.7 The Assignment message sequence flow diagram. 59
5.8 The Implementation message sequence flow diagram. 61
5.9 The complete set of assigned waypoints. 61
5.10 The Execution message sequence flow diagram. 61
5.11 Start executing the plan. 61
5.12 Pointing the sensor along the waterway. 61
5.13 Assigned UAV plans. 63
5.14 Initial UAV positions, requested tasks, and keep-out boundaries. 63
5.15 Distributed cooperation example message flow. 64

Introduction

1.1 Vision

The vision of UxAS is to become the Government standard for verifiable autonomy of multi-vehicle cooperative systems.

1.2 Purpose/Goals

The purpose of UxAS is to enable unmanned vehicle (UxV) autonomy by developing both a library of automated behaviors and the architecture to connect them. The goals are to develop services and tasks that enhance UxV autonomy, provide evidence of correct and safe system behavior, and to demonstrate modularity and rapid integration of capabilities.

1.3 Notional block diagram

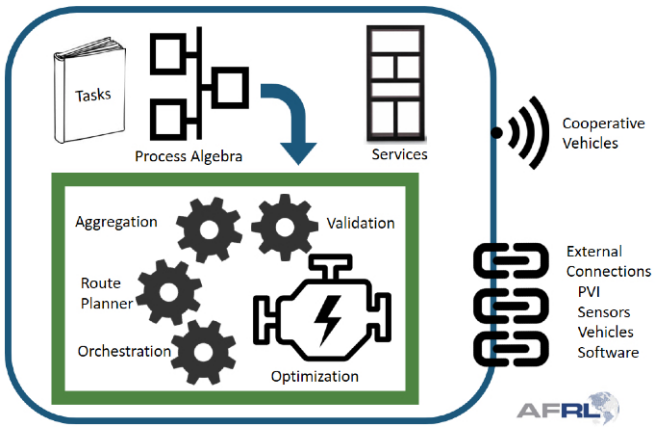


Figure 1.1: Notional Block Diagram

1.4 Feature list

1.5 *Unmanned Systems Autonomy Services*

UxAS is a software architecture that provides a framework to construct and deploy software services that are used to enable autonomous capabilities on-board unmanned systems. Along with the services, UxAS provides a means for defining, implementing, and exchanging inter/intra service messages. Based on many years of design, implementation, and flight testing of collaborative algorithms for UAVs, UxAS was designed, implemented, and flight tested for AFRL's ICE-T program. UxAS relies on two software systems to provide services, the Lightweight Message Construction Protocol (LMCP), and ZeroMQ.

LMCP was constructed by researchers at AFRL to provide a 'structure for common structured data and a process for serializing objects based on those types' and 'a method for encapsulating objects for transmission between applications'[@Duquette:2010]. The ability for LMCP to use Message Data Modules (MDM) makes it possible to define custom set of messages for each target system. An example MDM is the Common Mission Automation Services Interface (CMASI) which was designed to define data relevant to mission planning and UAV autonomy.

UxAS uses ZeroMQ to implement intra-process, inter-process, vehicle to vehicle, and vehicle to ground messaging. The authors describe ZeroMQ this way: 'looks like an embeddable networking library but acts like a concurrency framework'[@ZeroMQBook:2013]. Inside each UxAS process ZeroMQ publish/subscribe and push/pull nodes are used to implement a data bus that is accessible to every in-process service. The services are configured to subscribe to the LMCP messages that they require and push any generated LMCP messages to the data bus.

UxAS is implemented in C++ and has been used with various operating systems including, LINUX, Windows, OS X, and embedded LINUX. Services inherit capabilities from a parent class that implements the core functionality, i.e. messaging, configuration, and execution. UxAS has been used for applications such as implementing collaboration algorithms onboard UAVs as well as implementing the core functionality of Unmanned Ground Sensors (UGS) with communication links to the UAVs.

Installation

2.1 Dependencies

The primary tools and dependencies to obtain, build, document, and simulate UxAS are:

- Git
- OpenGL
- UUID library
- Boost
- Python 3
- Meson
- Ninja
- LmcpGen
- OpenAMASE (optional, simulation)
- NetBeans with Java JDK (optional, simulation)
- Doxygen (optional, documentation)
- LaTeX (optional, documentation)

The UxAS build system will download and compile the following external libraries

- Google Test (gTest)
- ZeroMQ (zeromq, czmq, cppzmq, zyre)
- SQLite
- Zlib
- Minizip

- Serial

Libraries for XML and GPS message parsing have numerous forks without centralized repository control. Code to build the following libraries is included with UxAS.

- PugiXML
- TinyGPS

2.2 *Supported platforms*

For an Ubuntu 16.04 or Mac OS X system with the listed prerequisite tools installed, UxAS should build from source without issue. Support for Windows is available on Windows 7 and 10 using Visual Studio.

2.2.1 *Installing Prerequisite Tools on Ubuntu Linux -or- Mac OS X (Partially-Automated)*

The following is a bash script that helps to partially-automation the "installing prerequisite tools" processes that are documented in this README.md file below.

This has been tested-working on Ubuntu 16.04, as of 2017-05-23

1. Download the script from the OpenUxAS repository (install_most_deps.sh) OR cd to your git clone d OpenUxAS directory
2. Run the script at the terminal: ./install_most_deps.sh
3. Follow the on-screen instructions

Note that the most up-to-date instructions on the dependencies-needed for UxAS are available below.

2.2.2 *Installing Prerequisite Tools on Ubuntu Linux*

1. Install git: in terminal
   - sudo apt-get install git
   - sudo apt-get install gitk
2. Install OpenGL development headers: in terminal
   - sudo apt-get install libglu1-mesa-dev
3. Install unique ID creation library: in terminal
   - sudo apt-get install uuid-dev

4. Install Boostlibraries (**optional but recommended**; see external dependencies section): in terminal
   - sudo apt-get install libboost-filesystem-dev libboost-regex-dev libboost-system-dev
5. Install doxygen and related packages **optional**: in terminal
   - sudo apt-get install doxygen
   - sudo apt-get install graphviz
   - sudo apt-get install texlive
   - sudo apt-get install texlive-latex-extra
6. Install pip3: in terminal
   - sudo apt install python3-pip
   - sudo -H pip3 install –upgrade pip
7. Install ninja build system: in terminal
   - sudo -H pip3 install ninja
8. Install meson build configuration: in terminal
   - sudo -H pip3 install meson
9. Ensure dependency search for meson is supported: in terminal
   - sudo apt-get install pkg-config
10. Install python plotting capabilities (**optional**): in terminal
    - sudo apt install python3-tk
    - sudo -H pip3 install matplotlib
    - sudo -H pip3 install pandas
11. Install NetBeans and Oracle Java JDK (**optional**)
    - Download the Linux x64 version
    - Run downloaded install script. In terminal: cd Downloads; sh jdk-9u131-nb-8_w-linux-x64.sh
    - Click Next three times, then Install
12. Enable C/C++ plug-in in NetBeans (**optional**)
    - Open NetBeans (in Ubuntu search, type NetBeans)
    - Choose Tools->Plugins from the top menu

- In the Available Plugins tab, search for C++
- Select C/C++ and click Install

13. Install Oracle Java run-time (required from LmcpGen): in terminal

- sudo add-apt-repository ppa:webupd8team/java
- sudo apt update; sudo apt install oracle-java8-installer
- sudo apt install oracle-java8-set-default

2.2.3 *Installing Prerequisites on Mac OS X*

1. Install XCode
2. Enable commandline tools. In terminal: xcode-select –install
3. Install homebrew (must be administrator). In terminal: sudo ruby -e "$(curl5 -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
4. Add homebrew to path. In terminal: echo $(export Path="/usr/local/bin:$PATH") >> /.bash_profile
5. Install git. In terminoal: brew install git
6. Install unique ID library. In terminal: brew install ossp-uuid
7. Install Boost library and configure it in a fresh shell. In terminal:
   - brew install boost
   - echo 'export BOOST_ROOT=/usr/local' >> /.bash_profile
   - bash
8. Install doxygen and related packages (**optional**). In terminal:
   - brew install doxygen
   - brew install graphviz
   - brew cask install mactex
9. Install pip3. In terminal:
   - brew install python3
10. Install nonja build system. In terminal:
   - brew install cmake
   - brew install pkg-config
   - sudo -H pip3 install scikit-build

- sudo -H pip3 install ninja

11. Install meson build configuration. In terminal:

- sudo -H pip3 install meson

12. Install python plotting capabilities (**optional**). In terminal:

- sudo -H pip3 install matplotlib
- sudo -H pip3 install pandas

13. Install Oracle Java run-time(required for LmcpGen)

14. Install NetBeans and Oracle JAva JDK (**optional**)

- Download the Mac OSX version
- Install .dmg

15. Enable C++ plug-in in NetBeans (**optional**)

- Open NetBeans
- Choose Tools->Plugins from the top menu
- In the Available Plugins tab, search for C++
- Select C/C++ and click Install

2.2.4 *Prep and Build on Native Windows*

1. Install Visual Studio 2017 Community Edition

- Ensure C++ selected in `Workloads` tab
- Ensure `Git for Windows` is selected in `Individual components` tab

2. Install Git with Bash shell

3. Install Python 3

- Make sure to check `Add Python 3.6 to PATH`
- Choose standard install (`Install Now`, requires admin)
- Verify installation by: `python --version` in `cmd` prompt
- Verify *pip* is also installed: `pip --version` in `cmd` prompt
- If unable to get python on path, follow this answer using location `C:\Users\[user]\AppData\Local\Programs\Python\Python36-32\`

4. Install *meson*

- In `cmd` prompt **with admin priviledges**: `pip install meson`

5. Install Boost

   - Note: the above link is for VS2017 pre-compiled libraries. To compile from source, you must install at the location: `C:\local\boost_1_64_0`

6. Pull UxAS repositories (from Git Bash shell)

   - `git -c http.sslVerify=false clone https://github.com/afrl-rq/OpenUxAS.git`
   - `git -c http.sslVerify=false clone https://github.com/afrl-rq/LmcpGen.git`
   - `git -c https://github.com/afrl-rq/OpenAMASE.git`

7. Build OpenAMASE

   - Load the OpenAMASE project in NetBeans and click `Build`

8. Auto-create the UxAS messaging library

   - Download released executable from GitHub
   - Place `LmcpGen.jar` in `LmcpGen/dist` folder
   - From the Git Bash shell in the root UxAS directory, run `sh RunLmcpGen.sh`
   - Note: For simplicity, make sure the LMCPGen, OpenUxAS, and OpenAMASE repositories follow the folder structure labeled in the Configure UxAS and Related Projects section.

9. Prepare build

   - Open VS command prompt (Tools -> Visual Studio Command Prompt)
   - Note: If the Visual Studio Command Prompt is absent from Visual Studio, it is also possible to perform the following actions by searching for the `Developer Command Prompt for VS 2017` application and switching the working directory to the root OpenUxAS directory
   - `python prepare`
   - `meson.py build --backend=vs` This should create a Visual Studio solution in the build folder.

10. Set UxAS as the Startup Project

    - Open the OpenUxAS.sln with Visual Studio, right-click the UxAS project found in the Solution Explorer
    - Select Set as StartUp Project

11. Add the boost library to the Library Directories for the dependent projects

    - With the OpenUxAS solution open in Visaul Studio, right-click the uxas project from the Solution Explorer and select `Properties` from the context menu.
    - Select `VC++ Directories` located within the `Configuration Properties` node in the `uxas Properties Pages` Pop Up
    - In under the general tab, there will be a `Library Directories` option. Add the absolute path of the boost libraries here. Given boost was setup with the instruction above, this path should be `C:\local\boost_1_64_0\lib32-msvc-14.1`

12. Build project with Visual Studio

    - Open project file `OpenUxAS.sln` in the `OpenUxAS/build` directory
    - In the `Solution Explorer`, right-click the `uxas` project, and select `Build` from the context menu

Caveats

- The Visual Studio backend for Meson mostly works, but will fail when regenerating build files. If you modify one of the `meson.build` files, delete the `build` directory and run `meson.py build --backend=vs` again. The steps following the `meson.build` command must also be performed.
- The UxAS test suite uses some hardcoded POSIX-style paths, and so does not currently work on Windows.

Navigating

3.1 File structure description with explanations of functionality of each part

This section outlines the file structure of UxAS with descriptions and explanations of each part. Figure 3.1 shows the UxAS folder and the folders housing its dependent projects. It is recommended that the OpenAMASE, LmcpGen, and OpenUxAS projects are placed in the same directory. The OpenAMASE project is used to run simulations with simulated aircraft. The LmcpGen project is used to generate LMCP messages.

Within OpenUxAS there is a series of folders. These folders include 3rd, doc, examples, mdms, resources, src, and tests. These folders can be seen in figure 3.2. The 3rd folder contains UxAS's dependent libraries. The doc folder contains UxAS documentation. The examples folder holds a series of configurations that are necessary to execute a simulation with OpenUxAS and OpenAMASE. The MDMs folder contains Message Data Modules that are used by LMCP to define custom sets of messages for each target system. The resources folder contains a series of scripts that can be used to analyze a simulation. The src folder contains the source code for OpenUxAS. The tests folder contains the unit tests.

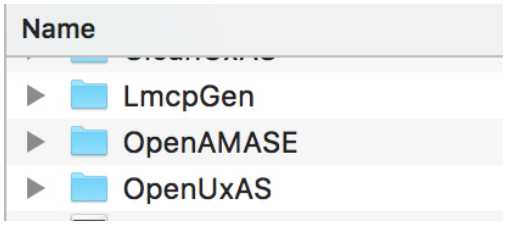


Figure 3.1: Open UxAS Folder

The contents of the doc folder can be seen in figure 3.3. This folder contains additional folders named doxygen, LMCP, and reference. The doxygen folder holds the files required to run doxygen and perform code inspection with the doxygen output. The LMCP folder contains files that describe the Message Data Modules. The reference folder contains UxAS flow charts, sequence diagrams, and the UxAS user manual.

The examples folder contains folders that contain configuration files required to execute a UxAS simulation. This folder can be seen in figure 3.4. Most of the simulations cannot be executed without OpenAMASE.

The mdms folder contains the message data modules. This folder can be seen in figure 3.5. These configuration files are used by LMCP

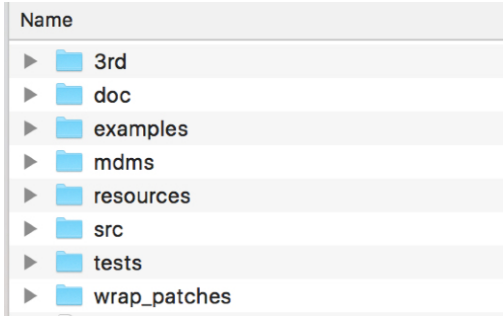


Figure 3.2: Open UxAS Folder Contents

to define custom sets of messages for each target system.

The resources folder contains a series of scripts that can be used to analyze a simulation. The resources folder can be seen in figure 3.6.

The src folder contains the OpenUxAS source code. This includes all of the header and implementation files. The contents of this folder can be seen in figure 3.7.

The test folder contains all the unit tests that can be run to assess the messages that UxAS sends. This folder can be seen in figure 3.8.

3.2 *Architecture and Messaging*

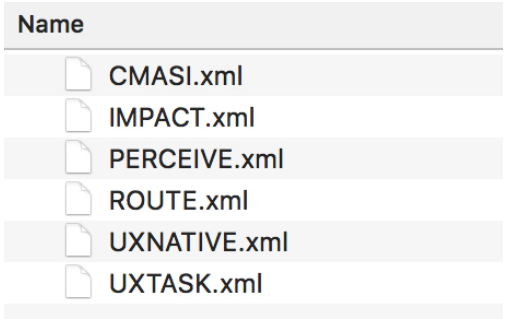


Figure 3.5: Open UxAS Message Data Models Folder

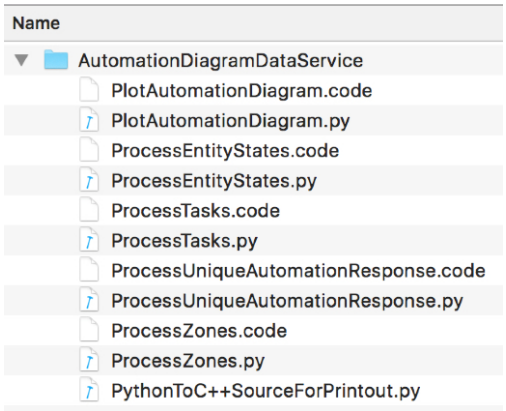


Figure 3.6: Open UxAS Resources Folder

Services

4.1 Adding New Services: Step-By-Step

1. Make copies of the service template source and header files:

   OpenUxAS/src/Services/oo_ServiceTemplate.cpp

   OpenUxAS/src/Services/oo_ServiceTemplate.h

2. Change the name of the copied files to reflect the name of the new service.

3. In the new files, search for the string *coo_ServiceTemplate* and Replace it with the new service name.

4. Change the unique include guard entries in the header file, i.e. *UXAS_oo_SERVICE_TEMPLATE_H*, to match the new service name.

5. Edit the file:

   OpenUxAS/src/Services/oo_ServiceList.h

   to add entries for the new service:

   1. include the header file for the new service in the section labeled *SERVICE HEADER FILES SECTION*, e.g.:

      #include “CmasiAreaSearchTaskService.h”

   2. add a service registration string in the section labeled *SERVICE REGISTRATION SECTION* e.g.:

      auto svc = uxas::stduxas::make_unique<afrl::cmasi::AreaSearchTask>();

   3. if the new service is a task, include the header file of the corresponding task message in the section labeled *INCLUDE TASK MESSAGES SECTION*, e.g.:

      #include “afrl/cmasi/AreaSearchTask.h”

   4. if the new service is a task, add a subscription string in the section labeled *SUBSCRIBE TO TASKS SECTION*, e.g.:

      addSubscriptionAddress(afrl::cmasi::AreaSearchTask::AREASEARCHTASK_FULL_LMCP_TYPE_NAME);

- NOTE: Before recompiling OpenUxAS, the new service's file name should be added to the srcs_services array within the meson.build file. This file is located at OpenUxAS/src/services/meson.build

4.2 *Configuring Services*

Service are configured using a global configuration file written in XML. The configuration file is selected either by using the default configuration file name: ***cfg.xml*** or passing in the path/filename when starting UxAS:

uxas_main **-cfgPath** *../PathToConfigurationFile/cfgFileName.xml*

The elements contained in an UxAS configuration file are, **XML Declaration**, **UxAS Element**, **Service Elements**, and **Bridge Elements**, see Figure 4.1.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<UxAS EntityID="1000" FormatVersion="1.0" EntityType="Aircraft">

    <Bridge Type="LmcpObjectNetworkZeroMqZyreBridge" NetworkDevice="enx281878b9065a">
        <SubscribeToExternalMessage MessageType="uxas.messages.task.AssignmentCoordination"/>
    </Bridge>

    <Service Type="TaskManagerService"/>

    <Service Type="SendMessagesService" PathToMessageFiles="../MessagesToSend/">
        <Message MessageFileName="AirVehicleConfiguration_V1000.xml" SendTime_ms="50"/>
        <Message MessageFileName="AirVehicleConfiguration_V2000.xml" SendTime_ms="60"/>
    </Service>

    <Service Type="MessageLoggerDataService" FilesPerSubDirectory="10000" LogFileName="testfile">
        <LogMessage MessageType="uxas" NumberMessagesToSkip="0"/>
    </Service>
</UxAS>
```

Figure 4.1: UxAS Example Configuraion

Here is the **XML Declaration**:
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
The **UxAS Element**:
<UxAS EntityID="1000" FormatVersion="1.0" EntityType="Aircraft">
accepts the following attributes:

EntityID Identification number of the entity represented by this instance of UxAS

EntityType used to differentiate between entities such as, *Aircraft* and *UGS*. Entries are defined by the services that use them.

ConsoleLoggerSeverityLevel if this attribute is present, all log messages at or below the severity level are displayed in the console. Valid entries are: *DEBUG*, *INFO*, *WARN*, and *ERROR*

MainFileLoggerSeverityLevel if this attribute is present, all log messages at or below the severity level are save in log files. Valid entries are: *DEBUG*, *INFO*, *WARN*, and *ERROR*

RunDuration_s UxAS will run for *RunDuration_s* seconds before terminating.

Service Elements configure the services. There can be as many Service Elements as required. The attributes of the Service Elements are the options defined by each service. For example, the HelloWorld service can be configured with the following Service Element:

<Service Type="HelloWorld" StringToSend="Hello from #1" SendPeriod_ms="1000"/>

Bridge Elements configure bridges, which are services that create communication connections. For the Distributed Cooperation example, the zyre connection is configured with the following Bridge Element:

<Bridge Type="LmcpObjectNetworkZeroMqZyreBridge" NetworkDevice="enx281878b9065a"> </Bridge>

4.3 Core Services

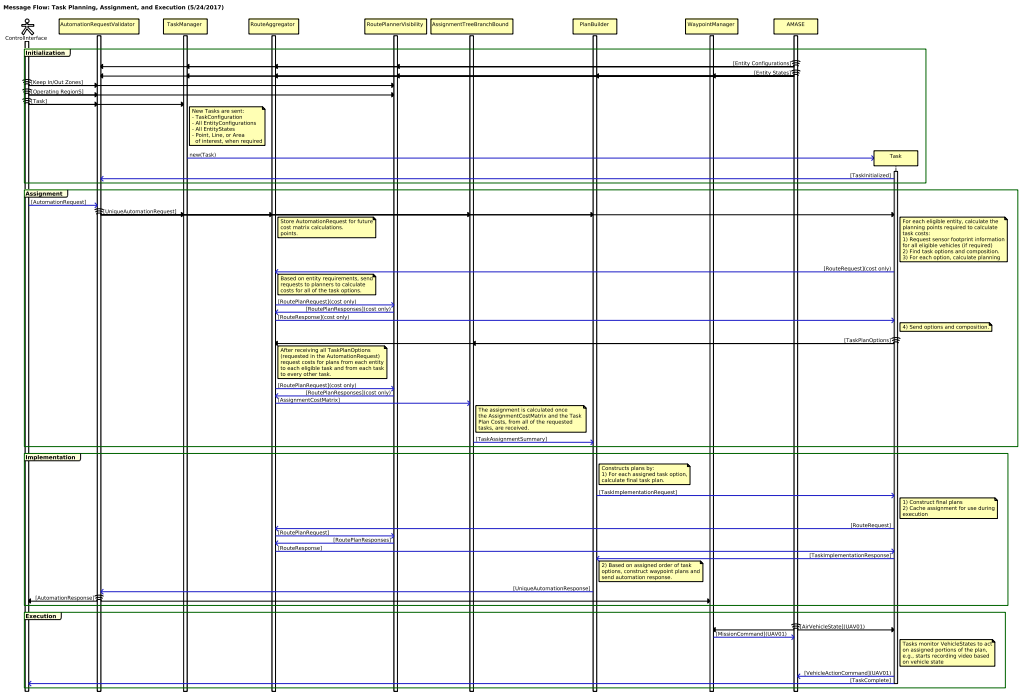
The core services of UxAS work in concert to carry out the *task assignment pipeline* which automates the calculation of the proper vehicle-task orderings for efficient overall mission completion. The design choice to seperate each of these services is motivated by consideration of computational complexity as well as availability in the literature for parts of the process (i.e. route planners and task assignment algorithms).

The message flow between the core services is captured in Figure 4.2 and indicates the temporal relationship between the core services in carrying out a complete automation request. Time proceeds from the top of the diagram to the bottom with each horizontal arrow representing a single LMCP message of the specified type.

AutomationRequestValidatorService - This service will check an outside automation request for feasibility. If any task, vehicle, or operating region in the request has not yet been described to the system, this service will return an error indicating that the request cannot be fulfilled. If all necessary parts of the request are available to the system, then this service will attach a unique identifier to the request and start the sequence of steps necessary to fulfill that request. If any other outside requests are made while the system is in the process of fulfilling an existing request, this service will queue those additional requests, ensuring that the system only handles one request at a time.

TaskManagerService - This service dynamically creates new task services as their descriptions are sent into the system. When a new *Task* is described, this service will send the proper configuration

Figure 4.2: Core Service Message Flow



message for construction of a new service that adheres to the general task interface specification for that particular *Task*. As part of the task service construction, this service will pass along all relevant details that a *Task* needs for calculation of its behavior (i.e. current vehicle states, defined areas of interest, etc).

Task - Tasks make up the atomic building blocks of automation requests. The composition of tasks and the assignment of particular vehicles to tasks comprises an automation response. When included as part of an automation request, a *Task* reports a set of possible *options* which represent the possible ways that the task can be completed. Each option is precisely a start and end location for applicable vehicles as well as the cost to complete the task using that option. The location and cost information for each option is used by the assignment service to select an option for each task that completes the overall mission efficiently. Once an option is selected by the assignment service, each *Task* also reports the set of waypoints that a vehicle should use to complete the task.

RoutePlannerVisibilityService - One of potentially many route planners, this service fulfills simple requests for routing in complex environments. For a given environment (described by sets of *KeepIn* and *KeepOut* polygons), a route planner will calculate the appropriate waypoints to guide a vehicle from a prescribed start location to a desired end location, ensuring that no waypoint is placed in an invalid area. This particular route planner uses a visibility graph as a basis to create distance-minimizing routes in two dimensions. Route planning services have a simple request-reply interface with each request corresponding to a single vehicle in a single environment.

RouteAggregatorService - This service works with a set of route planners, requesting routes from the appropriate planner based on the situation. Additionally, this service provides the capability to make large-scale route planning requests involving multiple vehicles and multiple start/end locations. A primary use of this service is to calculate routes for all vehicles to travel between task locations. By orchestrating the cost-to-go calculations between tasks, this service acts as the bookkeeper for the complete cost map required by assignment algorithms.

AssignmentTreeBranchBoundService - This service provides a resource allocation algorithm to determine an efficient use of the vehicle assets to fulfill tasks in the system. By design, the resource allocation is de-coupled from route planning and only uses the costs provided by the *RouteAggregatorService*. The ultimate output of this service is an ordered list of tasks for each vehicle. This ordering must account for the process algebra relationship between tasks. This particular assignment service builds a tree of possible assignment

orderings and prunes that tree based on the prescribed task relationships. The algorithm proceeds in two phases: 1) greedy, depth-first search to find a feasible plan; 2) backtracking up the tree in a branch-and-bound manner to discover plans that are lower cost than the initial greedy plan. Once a predetermined amount of the tree has been searched (to ensure worst-case execution time), the assignment service will return the most efficient task ordering discovered.

PlanBuilderService - With a task ordering determined by the assignment service, the *PlanBuilderService* will organize the final set of waypoints that each vehicle should follow to complete the assignment. Working through the task ordering, this service requests the appropriate *Tasks* in the assigned order to build a complete set of waypoints for each vehicle to carry out the mission. By stitching each part of the mission together in the proper order, this service provides the automation response that fulfills the original, outside automation request.

What follows are details for each of the core services and a rough description of the state machines that each follows to complete its part of the overall task assignment process. See the auto-generated LMCP documentation for precise details of each message.

4.3.1 *AutomationRequestValidatorService*

This service provides a simple sanity check on external automation requests. It also queues requests and tags them with unique identifiers, feeding them into the system one at a time.

This service has two states: **idle** and **busy**. In both states, when a non *AutomationRequest* message is received, a local memory store is updated to maintain a list of all available tasks, vehicle configurations, vehicle states, zones, and operating regions.

Upon reception of an *AutomationRequest* message, this service ensures that such a request can be carried out by checking its local memory store for existence of the requested vehicles, tasks, and operating region. If the request includes vehicles, tasks, or an operating region that has not previously been defined, this service will publish an error message.

Upon determination that the *AutomationRequest* includes only vehicles, tasks, and an operating region that have previously been defined, this service creates a *UniqueAutomationRequest* with a previously unused unique identifier. If in the **idle** state, this service will immediately publish the *UniqueAutomationRequest* message and transition to the **busy** state. If already in the **busy** state, the *UniqueAutomationRequest* will be added to the end of a queue.

When this service receives either an error message (indicating that

the *UniqueAutomationRequest* cannot be fulfilled or a corresponding *UniqueAutomationResponse*), it will publish the same message. If in the **idle** state, it will remain in the **idle** state. If in the **busy** state, it will remove from the queue the request that was just fulfilled and then send the next *UniqueAutomationRequest* in the queue. If the queue is empty, this service transitions back to the **idle** state.

This service also includes a parameter that allows an optional *timeout* value to be set. When a *UniqueAutomationRequest* is published, a timer begins. If the *timeout* has been reached before a *UniqueAutomationResponse* is received, an error is assumed to have occured and this service removes the pending *UniqueAutomationRequest* from the queue and attempts to send the next in the queue or transition to **idle** if the queue is empty.

Table 4.1: Table of messages that the *AutomationRequestValidatorService* receives and processes.

Message Subscription	Description
AutomationRequest	Primary message to request a set of Tasks to be completed by a set of vehicles in a particular airspace configuration (described by an *OperatingRegion*).
(2 ms work)	Determines validity of *AutomationRequest*. **idle** -> **busy**, emit *UniqueAutomationRequest* **busy** -> **busy**, add *UniqueAutomationRequest* to queue
EntityConfiguration	Vehicle capabilities (e.g. allowable speeds) are described by entity configuration messages. Any vehicle requested in an *AutomationRequest* must previously be described by an associated *EntityConfiguration*.
(0 ms work)	Add to internal storage for use in validation step. **idle** -> **idle**, add to storage **busy** -> **busy**, add to storage
EntityState	Describes the actual state of a vehicle in the system including position, speed, and fuel status. Each vehicle in an *AutomationRequest* must have reported its state.

Message Subscription	Description
(0 ms work)	Add to internal storage for use in validation step. **idle** -> **idle**, add to storage **busy** -> **busy**, add to storage
Task	Details a particular task that will be referenced (by ID) in an *AutomationRequest*.
(0 ms work)	Add to internal storage for use in validation step. **idle** -> **idle**, add to storage **busy** -> **busy**, add to storage
TaskInitialized	Indicates that a particular task is ready to proceed with the task assignment sequence. Each task requested in the *AutomationRequest* must be initialized before a *UniqueAutomationRequest* is published.
(0 ms work)	Add to internal storage for use in validation step **idle** -> **idle**, add to storage **busy** -> **busy**, add to storage
KeepOutZone	Polygon description of a region in which vehicles must not travel. If referenced by the *OperatingRegion* in the *AutomationRequest*, zone must exist for request to be valid.
(0 ms work)	Add to internal storage for use in validation step. **idle** -> **idle**, add to storage **busy** -> **busy**, add to storage
KeepInZone	Polygon description of a region in which vehicles must remain during travel. If referenced by the *OperatingRegion* in the *AutomationRequest*, zone must exist for request to be valid.
(0 ms work)	Add to internal storage for use in validation step. **idle** -> **idle**, add to storage **busy** -> **busy**, add to storage
OperatingRegion	Collection of *KeepIn* and *KeepOut* zones that describe the allowable space for vehicular travel. Must be defined for *AutomationRequest* to be valid.

Message Subscription	Description
(0 ms work)	Add to internal storage for use in validation step. **idle** -> **idle**, add to storage **busy** -> **busy**, add to storage
UniqueAutomationResponse	Completed response from the rest of the task assignment process. Indicates that the next *AutomationRequest* is ready to be processed.
(1 ms work)	If response ID does not match request ID at top of queue, ignore and remain in current state. Otherwise: **idle** -> **idle**, normal operation should preclude receiving this message in the **idle** state **busy**, emit corresponding *AutomationResponse* If queue is empty, **busy** -> **idle**, else **busy** -> **busy**, emit request message at top of queue

Table 4.2: Table of messages that the *AutomationRequestValidatorService* publishes.

Message Publication	Description
UniqueAutomationRequest	A duplicate message to an external *AutomationRequest* but only published if the request is determined to be valid. Also includes a unique identifier to match to the corresponding response.
ServiceStatus	Error message when a request is determined to be invalid. Includes human readable error message that highlights which portion of the *AutomationRequest* was invalid.
AutomationResponse	Upon reception of a completed *UniqueAutomationResponse*, this message is published as a response to the original request.

4.3.2 *TaskManagerService*

The *TaskManagerService* is a very straight-forward service. Upon reception of a Task message, it will send the appropriate *Create-*

NewService message. To do so, it catalogues all entity configurations and current states; areas, lines, and points of interest; and current waypoint paths for each vehicle. This information is stored in local memory and appended as part of the *CreateNewService* message which allows new Tasks to immediately be informed of all relevant information needed to carry out a Task.

When *TaskManagerService* receives a *RemoveTasks* message, it will form the appropriate *KillService* message to properly destroy the service that was created to fulfill the original Task.

Table 4.3: Table of messages that the *TaskManagerService* receives and processes.

Message Subscription	Description
Task	Primary message that describes a particular task. The task manager will make the appropriate service creation message to build a service that directly handles this requested Task.
(1 ms work)	Emit *CreateNewService* message
RemoveTasks	Indicates that Task is no longer needed and will not be included in future *AutomationRequest* messages. Task manager will send the proper *KillService* message to remove the service that was constructed to handle the requested Task.
(1 ms work)	Emit *KillService* message corresponding to *Task*
EntityConfiguration	Vehicle capabilities (e.g. allowable speeds) are described by entity configuration messages. New Tasks are informed of all known entities upon creation.
(0 ms work)	Store to report during *Task* creation
EntityState	Describes the actual state of a vehicle in the system including position, speed, and fuel status. New Tasks are informed of all known entity states upon creation.
(0 ms work)	Store to report during *Task* creation

Message Subscription	Description
AreaOfInterest LineOfInterest PointOfInterest	Describes known geometries of areas, lines, and points. New Tasks are informed of all such named areas upon creation.
(0 ms work)	Store to report during *Task* creation
MissionCommand	Describes current set of waypoints that a vehicle is following. New Tasks are informed of all known current waypoint routes upon creation.
(0 ms work)	Store to report during *Task* creation

Table 4.4: Table of messages that the *TaskManagerService* publishes.

Message Publication	Description
CreateNewService	Primary message published by the Task Manager to dynamically build a new Task from an outside description of such a Task.
KillService	When Tasks are no longer needed, the Task Manager will correctly clean up and destroy the service that was built to handle the original Task.

4.3.3 *Task*

A *Task* forms the core functionality of vehicle behavior. It is the point at which a vehicle (or set of vehicles) is dedicated to a singular goal. During *Task* execution, a wide spectrum of behavior is allowed, including updating waypoints and steering sensors. As part of the core services, this general *Task* description stands in for all *Tasks* running in the system.

The general *Task* interaction with the rest of the task assignment pipeline is complex. It is the aggregation of each *Task's* possibilities that defines the complexity of the overall mission assignment. These *Task* possibilities are called *options* and they describe the precise ways that a *Task* could unfold. For example, a *LineSearchTask* could present two options to the system: 1) search the line from East-to-West and 2) search the line from West-to-East. Either is valid and a selection of one of these options that optimizes overall mission efficiency is the role of the assignment service.

A general *Task* is comprised of up to nine states with each state corresponding to a place in the message sequence that carries out the task assignment pipeline. The states for a *Task* are:

- **Init**: This is the state that all *Tasks* start in and remain until all internal initialization is complete. For example, a *Task* may need to load complex terrain or weather data upon creation and will require some (possibly significant) start-up time. When a *Task* has completed its internal initialization, it must report transition from this state via the *TaskInitialized* message.

- **Idle**: This represents the state of a *Task* after initialization, but before any requests have been made that include the *Task*. *UniqueAutomationRequest* messages trigger a transition from this state into the **SensorRequest** state.

- **SensorRequest**: When a *Task* is notified of its inclusion (by noting the presence of its ID in the *Tasks* list of an *UniqueAutomationRequest* message), it can request calculations that pertain to the sensors onboard the vehicles that are also included in the *UniqueAutomationRequest* message. While waiting for a response from the *SensorManagerService*, a *Task* is in the **SensorRequest** state and will remain so until the response from the *SensorManagerService* is received.

- **OptionRoutes**: After the *SensorManagerService* has replied with the appropriate sensor calculations, the *Task* can request waypoints from the *RouteAggregatorService* that carry out the on-*Task* goals. For example, an *AreaSearchTask* can request routes from key surveillance positions that ensure sensor coverage of the entire area. The *Task* remains in the **OptionRoutes** state until the *RouteAggregatorService* replies.

- **OptionsPublished**: When routes are returned to the *Task*, it will utilize all route and sensor information to identify and publish the applicable *TaskOptions*. The determination of *TaskOptions* is key to overall mission performance and vehicle behavior. It is from this list of options that the assignment will select in order to perform this particular *Task*. After publication of the options, a *Task* waits in the **OptionsPublished** state until the *TaskImplementationRequest* message is received, whereupon it switches to **FinalRoutes**.

- **FinalRoutes**: Upon reception of a *TaskImplementationRequest*, a *Task* is informed of the option that was selected by the assignment service. At this point, a *Task* must create the final set of waypoints that include both *enroute* and *on-task* waypoints from the specified

vehicle location. The *Task* is required to create the *enroute* waypoints since a route refinement is possible, taking advantage of the concrete prior position of the selected vehicle. The *Task* remains in the **FinalRoutes** state until the route request is fulfilled by the *RouteAggregatorService*.

- **OptionSelected**: When the final waypoints are returned from the *RouteAggregatorService*, the *Task* publishes a complete *TaskImplementationResponse* message. A *Task* will remain in this state until an *EntityState* message includes this *Task* ID in its *AssociatedTaskList*. If during this state, a subsequent *UniqueAutomationRequest* is made, the *Task* returns to the **SensorRequest** state and immediately attempts to fulfill the requirements of the new *UniqueAutomationRequest*. This behavior implies that a *Task* can only be part of a single *AutomationRequest* and subsequent requests always override previous requests.
- **Active**: If the *Task* is in the **OptionSelected** state and an *EntityState* message is received which includes the *Task* ID in the *AssociatedTaskList*, then the *Task* switches to the **Active** state and is allowed to publish new waypoints and sensor commands at will. A *Task* remains in the **Active** state until a subsequent *EntityState* message does *not* list the *Task* ID in its *AssociatedTaskList*. At which point, a transition to **Completed** is made. Note that a *Task* can reliquish control indirectly by sending the vehicle to a waypoint not tagged with its own ID. Likewise, it can maintain control indefinitely by ensuring that the vehicle only ever go to a waypoint that includes its ID. If a *UniqueAutomationRequest* message that includes this *Task* ID is received in the **Active** state, it transitions to the **Completed** state.
- **Completed**: In this state, the *Task* publishes a *TaskComplete* message and then immediately transitions to the **Idle** state.

Table 4.5: Table of messages that a general *Task* receives and processes.

Message Subscription	Description
UniqueAutomationRequest	Indicates which *Tasks* are to be considered as well as the set of vehicles that can be used to fulfill those *Tasks*. Upon reception of this message, if a *Task* ID is included, it will publish *TaskPlanOptions*.

Message Subscription	Description
(2 ms work)	If included in the request, begin process of calculating task options and costs by emitting *SensorFootprintRequests* **Idle** -> **SensorRequest** in normal operation **OptionSelected** -> **SensorRequest**, when interrupted
TaskImplementationRequest	After an assignment has been made, each *Task* involved is requested to build the final set of waypoints that complete the *Task* and corresponding selected option. A *Task* must build the route **to** the *Task* as well as waypoints that implement the *Task*. For each on-task waypoint, the *AssociatedTaskList* must include the *Task* ID.
(2 ms work)	**OptionsPublished** -> **FinalRoutes**, emit *RouteRequest* to determine final waypoint routes needed for implementation
EntityConfiguration	Vehicle capabilities (e.g. allowable speeds) are described by entity configuration messages. *Tasks* can reason over sensor and vehicle capabilites to present the proper options to other parts of the system. If a vehicle does not have the capability to fulfill the *Task* (e.g. does not have a proper sensor), then the *Task* shall not include that vehicle ID in the list of eligible entities reported as part of an option.
(0 ms work)	Add to internal storage for use in calculating options No state change

Message Subscription	Description
EntityState	Describes the actual state of a vehicle in the system including position, speed, and fuel status. This message is primary feedback mechanism used for *Tasks* to switch to an **Active** state. When a *Task* ID is listed in the *AssociatedTaskList* of an *EntityState* message, the *Task* is allowed to update waypoints and sensor commands at will.
(0 ms work)	**OptionSelected**: if *Task* ID is listed in *AssociatedTaskList*, then -> **Active** **Active**: if *Task* ID is NOT listed in *AssociatedTaskList*, then -> **Completed**
RouteResponse	Collection of route plans that fulfill a set of requests for navigation through an *OperatingRegion*. A *Task* must request the waypoints to route a vehicle from its last to the start of the *Task*. Additionally, this message can be used to obtain on-task waypoints.
(1 ms work)	**OptionRoutes** -> **OptionsPublished**, emit *TaskPlanOptions* **FinalRoutes** -> **OptionSelected**, emit *TaskImplementationResponse*
SensorFootprintResponse	Collection of sensor information at different conditions corresponding to a *SensorFootprintRequests* message. Used to determine if a particular entity with known sensor payloads can meet the sensor resolution constraints required to fulfill this *Task*.
(1 ms work)	**SensorRequest** -> **OptionRoutes**, emit *RouteRequest* to determine on-*Task* waypoints

Table 4.6: Table of messages that a general *Task* publishes.

Message Publication	Description
TaskPlanOptions	Primary message published by a *Task* to indicate the potential different ways a *Task* could be completed. Each possible way to fulfill a *Task* is listed as an *option*. *TaskOptions* can also be related to each other via Process Algebra.
TaskImplementationResponse	Primary message published by a *Task* that reports the final set of waypoints to both navigate the selected vehicle to the *Task* as well as the waypoints necessary to complete the *Task* using the selected option.
RouteRequest	Collection of route plan requests to leverage the route planner capability of constructing waypoints that adhere to the designated *OperatingRegion*. This request is made for waypoints en-route to the *Task* as well as on-task waypoints.
SensorFootprintRequests	Collection of requests to the *SensorManagerService* to determine ground-sample distances possible for each potential entity. Uses camera and gimbal information from the cached *EntityConfiguration* messages.
VehicleActionCommand	When a *Task* is **Active**, it is allowed to update sensor navigation commands to on-task vehicles. This message is used to directly command the vehicle to use the updated behaviors calculated by the *Task*.
TaskComplete	Once a *Task* has met its goal or if a vehicle reports that it is no longer on-task, a previously **Active** *Task* must send a *TaskComplete* message to inform the system of this change.

4·3·4 *RoutePlannerVisibilityService*

The *RoutePlannerVisibilityService* is a service that provides route planning using a visibility heuristic. One of the fundamental architectural decisions in UxAS is separation of route planning from task assignment. This service is an example of a route planning service for aircraft. Ground vehicle route planning (based on Open Street Maps data) can be found in the *OsmPlannerService*.

The design of the *RoutePlannerVisibilityService* message interface is intended to be as simple as possible: a route planning service considers routes only in fixed environments for known vehicles and handles requests for single vehicles. The logic necessary to plan for multiple (possibly heterogeneous) vehicles is handled in the *RouteAggregatorService*.

In two dimensional environments composed of polygons, the shortest distance between points lies on the visibility graph. The *RoutePlannerVisibilityService* creates such a graph and, upon request, adds desired start/end locations to quickly approximate a distance-optimal route through the environment. With the straight-line route created by the searching the visibility graph, a smoothing operation is applied to ensure that minimum turn rate constraints of vehicles are satisfied. Note, this smoothing operation can violate the prescribed keep-out zones and is not guaranteed to smooth arbitrary straight-line routes (in particular, path segments shorter than the minimum turn radius can be problematic).

Due to the need to search over many possible orderings of *Tasks* during an assignment calculation, the route planner must very quickly compute routes. Even for small problems, hundreds of routes must be calculated before the assignment algorithm can start searching over the possible ordering. For this reason it is imperative that the route planner be responsive and efficient.

Table 4.7: Table of messages that the *RoutePlannerVisibilityService* receives and processes.

Message Subscription	Description
RoutePlanRequest "AircraftPathPlanner"	Primary message that describes a route plan request. A request considers only a single vehicle in a single *OperatingRegion* although it can request multiple pairs of start and end locations with a single message. In addition to subscribing to *RoutePlanRequest*, this service also subscribes to the group mailbox "AircraftPathPlanner". Upon reception of a message on this channel, the service will process it as if it came over the broadcast channel. The return message always uses return-to-sender addressing.
(10 ms work)	For each start/end pair, this service will compute a path that respects the geometric constraints imposed by the corresponding *OperatingRegion*. For each start/end pair, the route planner could reasonably work for 10ms to complete the calculation. Once all have been calculated, emits *RoutePlanResponse*.
KeepOutZone	Polygon description of a region in which vehicles must not travel. This service will track all *KeepOutZones* to compose them upon reception of an *OperatingRegion*.
(0 ms work)	Store for use during calculation of operating region map
KeepInZone	Polygon description of a region in which vehicles must remain during travel. This service will track all *KeepInZones* to compose them upon reception of an *OperatingRegion*.
(0 ms work)	Store for use during calculation of operating region map

Message Subscription	Description
OperatingRegion	Collection of *KeepIn* and *KeepOut* zones that describe the allowable space for vehicular travel. When received, this service creates a visibility graph considering the zones referenced by this *OperatingRegion*. Upon *RoutePlanRequest* the visibility graph corresponding to the *OperatingRegion* ID is retreived and manipulated to add start/end locations and perform the shortest path search.
(20 ms work)	To respond quickly to *RoutePlanRequest* messages, the *RoutePlannerVisibilityService* will create a pre-processed map using the geometric constraints from the *OperatingRegion*. The result is stored for later requests.
EntityConfiguration	Vehicle capabilities (e.g. allowable speeds) are described by entity configuration messages. This service calculates the minimum turn radius of the entity by using the max bank angle and nominal speed. Requested routes are then returned at the nominal speed and with turns approximating the minimum turn radius.
(20 ms work)	Stored for use during *RoutePlanRequest*s. Also used to update operating region maps based on the capability of the entity.

Table 4.8: Table of messages that the *RoutePlannerVisibilityService* publishes.

Message Publication	Description
RoutePlanResponse	This message contains the waypoints and time cost that fulfills the route request. This message is the only one published by the *RoutePlannerVisibilityService* and is always sent using the return-to-sender addressing which ensures that only the original requester receives the response.

4.3.5 *RouteAggregatorService*

The *RouteAggregatorService* fills two primary roles: 1) it acts as a helper service to make route requests for large numbers of heterogenous vehicles; and 2) it constructs the task-to-task route-cost table that is used by the assignment service to order the tasks as efficiently as possible. Each functional role acts independently and can be modeled as two different state machines.

The *Aggregator* role orchestrates large numbers of route requests (possibly to multiple route planners). This allows other services in the system (such as *Tasks*) to make a single request for routes and receive a single reply with the complete set of routes for numerous vehicles.

For every aggregate route request (specified by a *RouteRequest* message), the *Aggregator* makes a series of *RoutePlanRequests* to the appropriate route planners (i.e. sending route plan requests for ground vehicles to the ground vehicle planner and route plan requests for aircraft to the aircraft planner). Each request is marked with a request ID and a list of all request IDs that must have matching replies is created. The *Aggregator* then enters a **pending** state in which all received plan replies are stored and then checked off the list of expected replies. When all of the expected replies have been received, the *Aggregator* publishes the completed *RouteResponse* and returns to the **idle** state.

Note that every aggregate route request corresponds to a separate internal checklist of expected responses that will fulfill the original aggregate request. The *Aggregator* is designed to service each aggregate route request even if a previous one is in the process of being fulfilled. When the *Aggregator* receives any response from a route

planner, it checks each of the many checklists to determine if all expected responses for a particuarl list have been met. In this way, the *Aggregator* is in a different **pending** state for each aggregate request made to it.

Table 4.9: Table of messages that the *RouteAggregatorService* receives and processes in its *Aggregator* role.

Message Subscription	Description
RouteRequest	Primary message that requests a large number of routes for potentially heterogeneous vehicles. The *Aggregator* will make a series of *RoutePlanRequests* to the appropriate planners to fulfill this request.
(1 ms work)	**idle** -> **pending**, indexed by *RouteRequest* request ID, create a checklist of expected responses. Emit a number of *RoutePlanRequest* messages equal to the number of vehicles in the `VehicleID` field of the original *RouteRequest*
EntityConfiguration	Vehicle capabilities (e.g. allowable speeds) are described by entity configuration messages. This service uses the *EntityConfiguration* to determine which type of vehicle corresponds to a specific ID so that ground planners are used for ground vehicles and air planners are used for aircraft.
(0 ms work)	No state change. Store to identify appropriate route planner by vehicle ID.
RoutePlanResponse	This message is the fulfillment of a single vehicle route plan request which the *Aggregator* catalogues until the complete set of expected responses is received.

Message Subscription	Description
(1 ms work)	Store response and check to see if this message completes any checklist. If a checklist is complete, use the corresponding request ID to create a complete *RouteResponse* message. Emit *RouteResponse* and **pending** -> **idle**.

Table 4.10: Table of messages that the *RouteAggregatorService* publishes in its *Aggregator* role.

Message Publication	Description
RouteResponse	Once the *Aggregator* has a complete set of responses collected from the route planners, the message is built as a reply to the original *RouteRequest*.
RoutePlanRequest	The *Aggregator* publishes a series of these requests in order to fulfill an aggregate route request. These messages are published in batch, without waiting for a reply. It is expected that eventually all requests made will be fulfilled.

The *RouteAggregatorService* also acts in the role of creating the *AssignmentCostMatrix* which is a key input to the assignment service. For simplicity, this role will be labeled as the *Collector* role. This role is triggered by the *UniqueAutomationRequest* message and begins the process of collecting a complete set of on-task and between-task costs.

The *Collector* starts in the **Idle** state and upon reception of a *UniqueAutomationRequest* message, it creates a list of *Task* IDs that are involved in the request and then moves to the **OptionsWait** state. In this state, the *Collector* stores all *TaskPlanOptions* and matches them to the IDs of the *Task* IDs that were requested in the *UniqueAutomationRequest*. When the expected list of *Tasks* is associated with a corresponding *TaskPlanOptions*, the *Collector* moves to the **RoutePending** state. In this state, the *Collector* makes a series of route plan requests from 1) initial conditions of all vehicles to all tasks and 2) route plans between the end of each *Task* and start of all other *Tasks*. Similar to the *Aggregator*, the *Collector* creates a checklist of expected route plan responses and uses that checklist to determine when the

complete set of routes has been returned from the route planners. The *Collector* remains in the **RoutePending** state until all route requests have been fulfilled, at which point it collates the responses into a complete *AssignmentCostMatrix*. The *AssignmentCostMatrix* message is published and the *Collector* returns to the **Idle** state.

Note that the *AutomationValidatorService* ensures that only a single *UniqueAutomationRequest* is handled by the system at a time. However, the design of the *Collector* does allow for multiple simultaneous requests as all checklists (for pending route and task option messages) are associated with the unique ID from each *UniqueAutomationRequest*.

Table 4.11: Table of messages that the *RouteAggregatorService* receives and processes in its *Collector* role.

Message Subscription	Description
UniqueAutomationRequest	Primary message that initiates the collection of options sent from each *Task* via the *TaskPlanOptions* message. A list of all *Tasks* included in the *UniqueAutomationRequest* is made upon reception of this message and later used to ensure that all included *Tasks* have responded.
(1 ms work)	**Idle** -> **OptionsWait**, create checklist of expected task options.
TaskPlanOptions	Primary message from *Tasks* that prescribe available start and end locations for each option as well as cost to complete the option. Once all expected *TaskPlanOptions* have been received, the *Collector* will use the current locations of the vehicles to request paths from each vehicle to each task option and from each task option to every other task option.

Message Subscription	Description
(1 ms work)	Store task options and check to see if this message completes the checklist. If the checklist is complete, create a series of *RoutePlanRequest* messages to find routes from the current locations of vehicles to each task and from each task to every other task. Emit this series of *RoutePlanRequest* messages, **OptionsWait** -> **RoutePending**.
EntityState	Describes the actual state of a vehicle in the system including position, speed, and fuel status. This message is used to create routes and cost estimates from the associated vehicle position and heading to the task option start locations.
(0 ms work)	No state change. Store for use in requesting routes from vehicle positions to task start locations.
RoutePlanResponse	This message is the fulfillment of a single vehicle route plan request which the *Collector* catalogues until the complete set of expected responses is received.
(1 ms work)	Store response and check to see if this message completes the cost matrix. If so, emit *AssignmentCostMatrix* and **RoutePending** -> **Idle**.

Table 4.12: Table of messages that the *RouteAggregatorService* publishes in its *Collector* role.

Message Publication	Description
AssignmentCostMatrix	Once the *Collector* has a complete set of *TaskPlanOptions* as well as routes between tasks and vehicles, this message is built to inform the next step in the task assignment pipeline: the *AssignmentTreeBranchBoundService*.

Message Publication	Description
RoutePlanRequest	The *Collector* publishes a series of these requests in order to compute the vehicle-to-task and task-to-task route costs. These messages are published in batch, without waiting for a reply. It is expected that eventually all requests made will be fulfilled.

4.3.6 *AssignmentTreeBranchBoundService*

The *AssignmentTreeBranchBoundService* is a service that does the primary computation to determine an efficient ordering and assignment of all *Tasks* to the available vehicles. The assignment algorithm reasons only at the cost level; in other words, the assignment itself does not directly consider vehicle motion but rather it uses estimates of that motion cost. The cost estimates are provided by the *Tasks* (for on-task costs) and by the *RouteAggregatorService* for task-to-task travel costs.

The *AssignmentTreeBranchBoundService* can be configured to optimize based on cumulative team cost (i.e. sum total of time required from each vehicle) or the maximium time of final task completion (i.e. only the final time of total mission completion is minimized). For either optimization type, this service will first find a feasible solution by executing a depth-first, greedy search. Although it is possible to request a mission for which **no** feasible solution exists, the vast majority of missions are underconstrained and have an exponential (relative to numbers of vehicles and tasks) number of solutions from which an efficient one must be discovered.

After the *AssignmentTreeBranchBoundService* obtains a greedy solution to the assignment problem, it will continue to search the space of possibilities via backtracking up the tree of possibilities and *branching* at descision points. The cost of the greedy solution acts as a *bound* beyond which no solution is be considered. In other words, as more efficient solutions are discovered, any partial solution that exeeds the cost of the current best solution will immediately be abandoned (cut) to focus search effort in the part of the space that could possibly lead to better solutions. In this way, solution search progresses until all possibilities have been exhausted or a pre-determined tree size has been searched. By placing an upper limit on the size of the tree to search, worst-case bounds on computation time can be made to ensure desired responsiveness from the *AssignmentTreeBranchBoundService*.

General assignment problems do not normally allow for specification of *Task* relationships. However, the *AssignmentTreeBranchBoundService* relies on the ability to specify *Task* relationships via Process Algebra constraints. This enables creation of moderately complex missions from simple atomic *Tasks*. Adherence to Process Algebra constraints also allows *Tasks* to describe their *option* relationships. The Process Algebra relationships of a particular *Task* option are directly substituted into and replace the original *Task* in the mission-level Process Algebra specification. Due to the heavy reliance on Process Algebra specifications, any assignment service that replaces *AssignmentTreeBranchBoundService* must also guarantee satisfaction of such specifications.

The behavior of the *AssignmentTreeBranchBoundService* is straightfoward. Upon reception of a *UniqueAutomationRequest*, this service enters the **wait** state and remains in this state until a complete set of *TaskPlanOptions* and an *AssignmentCostMatrix* message have been received. In the **wait** state, a running list of the expected *TaskPlanOptions* is maintained and checked off when received. Upon receiving the *AssignmentCostMatrix* (which should be received strictly after the *TaskPlanOptions* due to the behavior of the *RouteAggregatorService*), this service conducts the branch-and-bound search to determine the proper ordering and assignment of *Tasks* to vehicles. The results of the optimization are packaged into the *TaskAssignmentSummary* and published, at which point this service returns to the **idle** state.

Table 4.13: Table of messages that the *AssignmentTreeBranchBoundService* receives and processes.

Message Subscription	Description
UniqueAutomationRequest	Sentinel message that initiates the collection of options sent from each *Task* via the *TaskPlanOptions* message. A list of all *Tasks* included in the *UniqueAutomationRequest* is made upon reception of this message and later used to ensure that all included *Tasks* have responded.
(0 ms work)	**idle** -> **wait**, store request ID for identification of corresponding *TaskPlanOptions* and *AssignmentCostMatrix*.

Message Subscription	Description
TaskPlanOptions	Primary message from *Tasks* that prescribe available start and end locations for each option as well as cost to complete the option. In the **wait** state, this service will store all reported options for use in calculating mission cost for vehicles when considering possible assignments.
(0 ms work)	No state change. Store cost of each task option for look-up during optimization.
AssignmentCostMatrix	Primary message that initiates the task assignment optimization. This message contains the task-to-task routing cost estimates and is a key factor in determining which vehicle could most efficiently reach a *Task*. Coupled with the on-task costs captured in the *TaskPlanOptions*, a complete reasoning over both traveling to and completing a *Task* can be looked up during the search over possible *Task* orderings.
(1500 ms work)	Using the cost of each task option (from the stored *TaskPlanOptions* messages) and the cost for each vehicle to reach each option (from *AssignmentCostMatrix*), perform an optimization attempting to find the minimal cost mission that adheres to the Process Algebra contraints. Upon completion, emit *TaskAssignmentSummary* and **wait** -> **idle**.

Table 4.14: Table of messages that the *AssignmentTreeBranchBoundService* publishes.

Message Publication	Description
TaskAssignmentSummary	The singular message published by this service which precisely describes the proper ordering of *Tasks* and the vehicles that are assigned to complete each *Task*.

4·3·7 *PlanBuilderService*

The final step in the task assignment pipeline is converting the decisions made by the *AssignmentTreeBranchBoundService* into waypoint paths that can be sent to each of the vehicles. Using the ordering of *Tasks* and the assigned vehicle(s) for each *Task*, the *PlanBuilderService* will query each *Task* in turn to construct enroute and on-task waypoints to complete the mission.

Similar to both the *RouteAggregator* and the *AssignmentTreeBranchBoundService*, the *PlanBuilderService* utilizes a received *UniqueAutomationRequest* to detect that a new mission request has been made to the system. The *UniqueAutomationRequest* is stored until a *TaskAssignmentSummary* that corresponds to the unique ID is received. At this point, the *PlanBuilderService* transitions from the **idle** state to the **busy** state.

Using the list of ordered *Tasks* dictated by the *TaskAssignmentSummary*, the *PlanBuilderService* sends a *TaskImplementationRequest* to each *Task* in order and waits for a *TaskImplementationResponse* from each *Task* before moving to the next. This is necessary as the ending location of a previous *Task* becomes the starting location for a subsequent *Task*. Since each *Task* is allowed to refine its final waypoint plan at this stage, the exact ending location may be different that was was originally indicated during the *TaskPlanOptions* phase. By working through the *Task* list in assignment order, all uncertainty about timing and location is elminated and each *Task* is allowed to make a final determination on the waypoints to be used.

Once all *Tasks* have reponded with a *TaskImplementationResponse*, the *PlanBuilderService* links all waypoints for each vehicle into a complete *MissionCommand*. The total set of *MissionCommands* are collected into the *UniqueAutomationResponse* which is broadcast to the system and represents a complete solution to the original *AutomationRequest*. At this point, the *PlanBuilderService* returns to the **idle** state.

Table 4.15: Table of messages that the *PlanBuilderService* receives and processes.

Message Subscription	Description
TaskAssignmentSummary	Primary message that dictates the proper order and vehicle assignment to efficiently carry out the requested mission. Upon reception of this messsage, the *PlanBuilderService* queries each *Task* in order for the final waypoint paths.
(2 ms work)	**idle** -> **busy**, create a queue of ordered *TaskImplementationRequest* messages in the order prescribed by the *TaskAssignmentSummary*. Emit request at top of queue.
EntityState	Describes the actual state of a vehicle in the system including position, speed, and fuel status. This message is used to inform the first *Task* of the location of the vehicles. Subsequent *Tasks* use the predicted positions and headings of vehicles after previous *Tasks* have reported waypoints earlier in the mission.
(0 ms work)	No state change. Store for use in creating *TaskImplementationRequest* messages.
TaskImplementationResponse	Primary message that each *Task* reports to inform this service of the precise waypoints that need to be followed to reach the *Task* and carry it out correctly. The ordered collection of these messages are used to build the final *UniqueAutomationResponse*.
(2 ms work)	Remove top of task request queue and update predicted locations of vehicles. If task request queue is not empty, configure request at top of queue with predicted vehicle positions and emit the corresponding *TaskImplementationRequest* message. If queue is empty, **busy** -> **idle**.

Message Subscription	Description
UniqueAutomationRequest	Informs this service of a new mission request in the system. Contains the desired starting locations and headings of the vehicles that are to be considered as part of the solution.
(0 ms work)	No state change. Store for use in creating *TaskImplementationRequest* messages. Note, positions in *UniqueAutomationRequest* override reported state positions stored when *EntityState* messages are received.

Table 4.16: Table of messages that the *PlanBuilderService* publishes.

Message Publication	Description
TaskImplementationRequest	The primary message used to query each *Task* for the proper waypoints that both reach and carry out the *Task*. Once the *PlanBuilderService* receives a corresponding response from each *Task*, it can construct a final set of waypoints for each vehicle.
UniqueAutomationResponse	This message contains a list of waypoints for each vehicle that was considered during the automation request. This collection of complete waypoints for the team fulfills the original request.

4.4 Additional Services

SensorManagerService - A service that constructs sensor footprints, calculates GSDs, determine sensor settings.

WaypointPlanManagerService - Serves waypoint plans to the vehicle interface.

00_ServiceTemplate - This is a basic service that can be used as a template when constructing new services.

01_HelloWorld - This is a basic example of a UxAS service that sends/receives KeyValuePair messages and prints out the results.

BatchSummaryService -

MessageLoggerDataService - This service logs messages received from other UxAS services to a SQLite database. Logging can be configured to log either all or a subset of service messages.

OperatingRegionStateService -

OsmPlannerService- loads an Open Street Map file and constructs ground plans/costs to be used for assignments.

SendMessageService - sends out messages, loaded from files, at a given time or periodically.

ServiceBase - the base class for all UxAS service classes. Service class constructors are registered in the *ServiceBase* creation registry.

ServiceManager - a singleton class that inherits from the *ServiceBase* class. It performs initial service creation for the UxAS entity at startup. After entity startup, it creates services per requests received from other services via messaging. The ServiceManager exclusively uses the *ServiceBase* creation registry to create services.

4.5 Task Services

oo_TaskTemplate - This is a basic task that can be used as a template when constructing new tasks

AngledAreaSearchTaskService - Area search task with specified direction

BlockadeTaskService - Task for using multiple vehicles to surround an entity, for example, multiple surface vehicles surrounding incoming enemy ship.

CmasiAreaSearchTaskService - Area search task

CmasiLineSearchTaskService - Defines a line search task. A line search is a list of points that forms a polyline. The ViewAngleList determines from which direction the line may be viewed. View angles are specified using the Wedge type. If the UseInertialViewAngles option is true, then wedges are defined in terms of North-East coordinates, otherwise wedges are defined relative to the line segment currently being viewed (a vector from point i through point i+1). To be a valid look angle, the line segment must be viewed from an angle within the bounds of the wedge.

CmasiPointSearchTaskService - Point search task

CommRelayTaskService - Task for providing comm relay support

CordonTaskService - Task for using multiple ground vehicles to block access to an area. Given a point to secure and a standoff distance, task identifies number (K) routes that must be blocked to successfully deny access to the area. If there are not enough eligible vehicles, then this task will use the maximum number of eligible vehicles in a best effort strategy which attempts to maximize radial

coverage.

EscortTaskService - Task for targeting surveillance at an offset of a moving entity, for example to scout ahead of a convoy.

ImpactLineSearchTaskService - Defines a line search task. A line search is a list of points that forms a polyline. The ViewAngleList determines from which direction the line may be viewed. View angles are specified using the Wedge type. If the UseInertialViewAngles option is true, then wedges are defined in terms of North-East coordinates, otherwise wedges are defined relative to the line segment currently being viewed (a vector from point i through point i+1). To be a valid look angle, the line segment must be viewed from an angle within the bounds of the wedge.

ImpactPointSearchTaskService - Impact Point Search Task

OverwatchTaskService - Multi vehicle overwatch task

PatternSearchTaskService - Search task with specified search pattern

TaskManagerService - A service that constructs/destroys tasks.

TaskServiceBase - A base service that implements storage/functions common to all tasks.

4.6 Connection Services

LmcpObjectNetworkPublishPullBridge -

LmcpObjectNetworkSerialBridge -

LmcpObjectNetworkSubscribePushBridge - connects an external entity to the internal message bus using *ZMQ_SUB* and *ZMQ_PUSH* sockets.

LmcpObjectNetworkTcpBridge - connects an external TCP/IP stream to the internal message bus.

LmcpObjectNetworkZeroMqZyreBridge - provides network discovery and communications. Dynamically discovers and bridges with zero-many Zyre-enabled systems.

ZeroMqZyreBridge - provides network discovery and communications. Dynamically discovers and bridges with zero-many Zyre-enabled systems.

4.7 Logging and Data Capture Services

MessageLoggerDataService - logs messages received from other UxAS services to files in a directory. Logging can be configured to log a subset of service messages, or all service messages.

Examples

5.1 AMASE simulation guide

The AMASE simulation is used to simulate the UxAS scenarios, see Figure 5.1. AMASE was developed by members of AFRL's Aerospace Vehicles Technology Assessment & Simulation Branch (AVTAS) to provide a common framework for research and development of cooperative control algorithms. AMASE was extended for UxAS in order to simulate *communications* and *on-board data processing*. AMASE includes the capability to connect to external programs using TCP/IP connections and LMCP messages. LMCP was developed by AFRL researchers to allow communication between cooperative control algorithms, ground stations, and simulations.

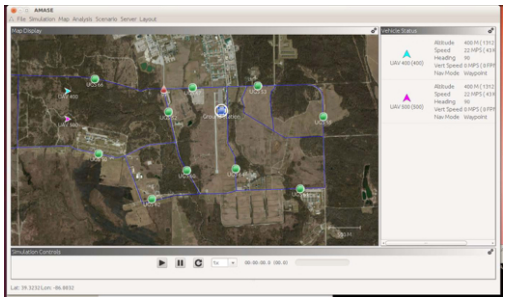


Figure 5.1: AMASE simulation snapshot.

AMASE is written in Java to make it cross-platform compatible. It consists of a graphical front-end that displays simulated objects, a user interface to control the simulation, simulated air and ground vehicles, and simulated communications. It has built-in, configurable, vehicle simulation models that are capable of following waypoint paths. AMASE uses the same LMCP messages as UxAS which makes it easy to communicate between the two.

Communication between AMASE and other programs is implemented through TCP/IP connections. When a program (UxAS) connects to AMASE's general connection (port 55555), the program sends LMCP messages, such as *MissionCommand*, to command the simulated vehicle's actions. AMASE sends all generated messages, such as *AirVehicleStates*, to all connected programs (UxAS). In this manner, every connected program communicates with all of the entities in the simulation. There are also customization connections that make it possible to connect directly to simulated entities. These direct connections pass, only the messages that are available to the simulated entity, from AMASE to the external program (UXAS) . This makes it possible to connect multiple copies of UxAS to AMASE as if they are installed on-board the entities. This facilitates software-in-the-loop simulations.

5.1.1 *Working With AMASE*

The code for AMASE is located in the folder:

```
uxas/tools/amase/
```

This folder contains three folders:

AMASE_DK/ contains the AMASE Development Kit, which is the core executable software and documentation for AMASE.

amase-icet/ contains code that implements new functionality for UxAS-type projects. UxAS examples execute AMASE from this folder.

amase-icet-lmcp/ contains the LMCP Java api generated to match the MDMs used by UxAS.

An AMASE simulation is configured by reading in a scenario file. These files use xml to define elements such as vehicle configurations, initial vehicle states, and initial vehicle commands. The file:

```
uxas/tools/amase/amase-icet/Scenario_WaterwaySearch.xml
```

is an example of a scenario file that configures two simulated UAVs. The scenario file can be loaded at run-time or after starting AMASE, from the menu:

```
file->OpenScenario
```

Many elements of the simulation are optional and are configured using files in the folder:

```
uxas/tools/amase/amase-icet/config/communications/
```

These files include:

EntityControl.xml - defines which models are used to make up the entity simulations. The port that are used to communicate directly to an entity are configure in this file.

EntityIcons.xml maps type of entity to the icon used to represent it.

Plugins.xml contains entries for configuring plug-ins. For example, the different layers used in the maps are configured in this file.

WindowService.xml configure layout and functionality of the windows in the AMASE user interface.

Configuring Communication Ports

By default the AMASE TCP/IP server is configured to connect on port **5555**. This is configured in the *Plugins.xml* file. Any client that

connects to the *localhost*, or the local IP address, on port **5555** will receive all message generated in AMASE. Any messages sent to this port will be available to all entities in AMASE.

In order to run UxAS as if it is on-board the simulated UAVs, new connection types were implemented that form a connection from UxAS to the simulated UAVs. AMASE sends only information available to the connected UAV to UxAS. Any messages sent from UxAS are delivered only to the connected UAV.

These connections are configured in the file *EntityControl.xml*. Here is a sample entry:

```
<TcpConnection Id="100" Port="9100" />
```

The value of *Id* is the AMASE entity ID and the value of ***Port*** is the port address of the connection.

Using NASA Worldwind Maps

5.1.2 *Running AMASE Step-by-Step*

1. open a terminal window in the folder:

   uxas/tools/amase/

2. start the bash script:

   ./runAMASE_Sample.sh

3. the AMASE user interface will open

4. push the "play" button to start the simulation

This starts the AMASE simulation with the scenario file:

Scenario_Sample.xml

The command line in the bash script is:

```
java -Xmx2048m -splash:./data/amase_splash.png -classpath
./dist/*:./dist/lib/* avtas.app.Application -config config/communications
-scenario "Scenario_Sample.xml";
```

In order to facilitate running AMASE from the same folders as UxAS, the following bash script was developed to start AMSE:

```
here=$PWD;
cd ../../tools/amase/amase-icet;
java -Xmx2048m -splash:./data/amase_splash.png -classpath ./dist/*:./dist/lib/* avtas.app.Application
-config config/communications -scenario "Scenario_WaterwaySearch.xml";
cd "$here";
```

5.1.3 *Connecting UxAS to AMASE*

Connections from UxAS to AMASE entities requires UxAS to connect as a TCP/IP client. To accomplish this, a ***LmcpObjectNetworkTcpBridge*** bridge (service) entry is needed in the UxAS config file. As an example, the following is the entry from the Waterway Search example:

```
<Bridge Type="LmcpObjectNetworkTcpBridge" TcpAddress="tcp://127.0.0.1:5555" Server="FALSE">
  <SubscribeToMessage MessageType="afrl.cmasi.MissionCommand" />
  <SubscribeToMessage MessageType="afrl.cmasi.LineSearchTask" />
  <SubscribeToMessage MessageType="afrl.cmasi.VehicleActionCommand" />
</Bridge>
```

In this case UxAS is connecting to the general **5555** connection. Any *MissionCommand*, *LineSerachTask*, or *VehicleActionCommand* messages generated in UxAS will be sent to all of the entities in AMASE.

Simulation Time

AMASE is a simulation that generates its own time. UxAS time is based on the real-time clock. This can pose problems for complex UxAS services that rely on temporal synchronization. In order to address this problem, the service ***Test_SimulationTime*** was implemented. It is configured in the following manner:

```
<Service Type="Test_SimulationTime"/>
```

The ***Test_SimulationTime*** service switches UxAS to *discrete-time* mode. It subscribes to *AirVehicleState* messages and when it receives one, it sets UxAS time equal to the time contained in the *AirVehicleState*. As AMASE runs it sends out *AirVehicleState* messages with the current simulation time which makes it possible to synchronize UxAS.

5.2 *Hello World Example*

(Demonstration of Basic UxAS configuration and service to service communication)

This is an introductory example to UxAS. The UxAS service ***HelloWorld*** is designed to perodically send and receive *KeyValuePair* messages. Parameters for the service are *StringToSend* and *SendPeriod_ms*. As the names imply, *StringToSend* is the string sent from the service and *SendPeriod_ms* is how often, in milliseconds, the string will be sent. Each instance of the ***HelloWorld*** service subscribes to *KeyValuePair* messages, which means that it will receive any of these messages sent from other services. The configuration file for this example,

uxas/examples/01_HelloWorld/cfg_HelloWorld.xml

contains entries for two instances of ***HelloWorld***:

```
<Service Type="HelloWorld" StringToSend="Hello from #1" SendPeriod_ms="1000"/>
<Service Type="HelloWorld" StringToSend="Hello from #2" SendPeriod_ms="5001"/>
```

As the example runs, both instances send *KeyValuePair* messages and each instance receives these messages from the other. Figure 5.2 shows a screen capture of the resulting console output.

5.2.1 Coding the Hello World Example

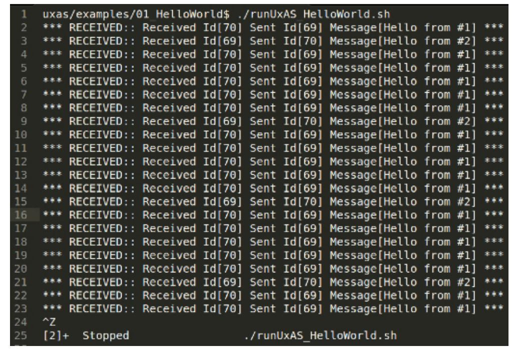


Figure 5.2: HelloWorld console messages.

The two files:

```
uxas/code/src/Services/01_HelloWorld.cpp
uxas/code/src/Services/01_HelloWorld.h
```

contain the source code for the ***HelloWorld*** service. These files were constructed using copies of the service template files:

```
uxas/code/src/Services/00_ServiceTemplate.cpp
uxas/code/src/Services/00_ServiceTemplate.h
```

Significant Modifications to the file *00_ServiceTemplate.h* consisted of:

a callback function for the timer :

```
void OnSendMessage();
```

a string to store messages to send :

```
std::string m_stringToSend = std::string("Hello World!");
```

an int64 to store the send period :

```
int64_t m_sendPeriod_ms{1000};
```

a uint64 to store the ID of the timer :

```
uint64_t m_sendMessageTimerId{0};
```

The ***HelloWorld*** service was implemented by:

1. Handling the parameters *StringToSend* and *SendPeriod_ms* in the *HelloWorld::configure* virtual method.
2. Instantiating a *uxas::common::TimerManager* timer in the *HelloWorld::initialize()* virtual method.
3. Starting the timer in the *HelloWorld::start()* virtual method.
4. Destroying the timer in the *HelloWorld::terminate()* virtual method.
5. Printing out the string for any received messages in the *HelloWorld::processReceivedLmcpMessage* virtual method.
6. Sending the message when timer times out in the *HelloWorld::OnSendMessage()* method.

5.2.2 *Hello World Example - Specifics*

This is a basic example of a UxAS service that sends/receives *KeyValuePair* messages and prints out the results.

Files

runUxAS_HelloWorld.sh - This is a bash shell script used to execute the example

cfg_HelloWorld.xml - This is the file used to configure the example for UxAS

01_HelloWorld.cpp - the C++ source code for the example. Note: this file is located in the following directory: code/src/Services/

01_HelloWorld.h - the C++ header file for the example. Note: this file is located in the following directory: code/src/Services/

Running the Example

1. open a ternimal window in the directory: "examples/01_HelloWorld/"
2. enter the command: ‘./runUxAS_HelloWorld.sh‘

What Happens?

- Two copies of the HelloWorld service start up and begin sending messages. One copy sends a message once a second and the other sends a message every 5 seconds. Each service receives the messsages sent by the other service. When messages are received the services print them out.

Things to Try

- Change the rate that messages are sent out. This is done by editing the ‘cfg_HelloWorld.xml‘ file and changing the value of ‘SendPeriod_ms‘. Note: the time is entered in milliseconds, i.e. 1000 milliseconds == 1 second.
- Change the message sent by each of the services. This is done by editing the ‘cfg_HelloWorld.xml‘ file and changing the value of ‘StringToSend‘.

5.3 Waterway Monitoring Example

(Path Plannning and Assignment Message Flow Example)

In order to explain how UxAS uses LMCP messages to generate path plans and assign Unmanned Air Vehicles (UAV) to perform tasks, a waterway monitoring scenario has been constructed. In this example, the challenge is to generate the commands required to cause one of two UAVs to fly alongside and record imagery of a given section of the waterway. Figure 5.3 shows the waterway of interest, as well as the initial position of the two UAVs. Here are the rules for this scenario:

Figure 5.3: The waterway to be monitored.

- The plans/assignments are generated by UxAS on the ground.
- A ground control station manages the message traffic between UxAS and the UAVs.
- The UAVs can start monitoring from either end of a waterway section.
- Each waterway section is to be monitored only one time.
- The UAVs must fly alongside the waterway section so that the sensor angle with respect to the waterway does not change
- The path plan/assignment should seek to minimize the maximum distance traveled by either of the UAVs.

Figure 5.4 depicts the flow of messages between services required to create plans, assign a UAV, and point the UAV's sensor as the plans are being executed. This figure will be used as a guide to explain how UxAS uses messages to plan and implement the waterway monitoring scenario.

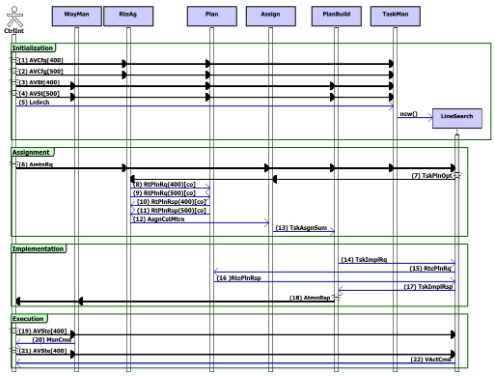


Figure 5.4: The message sequence flow diagram.

- The **rectangles** across the top of the diagram, and the one on the right side, represent UxAS services, which are described in the next section.
- The **stick figure** labeled 'CtrlInt' is the interface to a ground control station that is connected to the UAVs via radio.
- The **horizontal lines** represent LMCP messages and the arrowheads show where messages are received. Note: messages are sent in order from top to bottom.

5.3.1 UxAS Services in the Message Flow

WayMan The **WaypointManager** is used during the execution phase to send small sets of waypoints to the UAVs via the ground station. It partitions the mission waypoints into overlaping sets that are

small enough that they can be sent to the UAVs given communication and autopilot constraints.

RteAg The **RouteAggregator** perfoms two functions it manages *Route Requests* and constructs the *Assignment Matrix*. It receives ***RouteRequest*** messages, then sends ***RoutePlanRequest*** messages to an appropriate planner for the vehicle type, e.g. UAV, ground vehicle, surface ship, and then send out the resulting ***RouteResponse*** messages. Once it receives ***TaskOption*** messages from all of the tasks specified in the ***AutomationRequest***, it sends out the messages necessary to build a matrix of costs, ***AssignmentCostMatrix***, that is used by the **Assignment** service.

Plan This **Planner** service is specialized to generate waypoints plans for a UAV to fly from a given position, with a given heading, to a second position, with a second heading.

Assign The **Assignment** service minimizing a cost function uses costs generated from the ***AssignmentCostMatrix*** and the ***TaskOption*** messages to assign vehicles to tasks.

PlanBuild The **PlanBuilder** service oversees the construction of waypoint plans for the mission, based on the ***TaskAssignmentSummary***, using ***TaskImplmentation*** messages to the tasks.

TaskMan The **TaskManager** service receives task messages and manages creation and deletion of corresponding task services.

LinSearch The **LineSearchTask** service manages the functionality required to build plans to cause the UAVs to follow a given line while pointing it sensor. Note: UxAS Tasks are special services that can be instantiated by sending task messages. Tasks can be used in a persistent fashion, i.e. they can be instantiated early and then assigned many times. Each task service is responsible for constructing task implementation plans for every eligible vehicle, calculating task costs and communicating them to the other services, and commanding task activities, such as sensor pointing, during task execution.

5.3.2 *Message Flow Description*

The following sections describe the message flow and how it is used to implement the Waterway Monitoring scenario. The message flow is broken into four phases: Initialization, Assignment, Implementation, and Execution. Each is described below. Note that in order to utilize the processor to best advantage, all processing is accomplished when the required information becomes available.

INITIALIZATION In this phase all the information required to calculate plans and assignments is passed into UxAS via messages, see Figure 5.5.

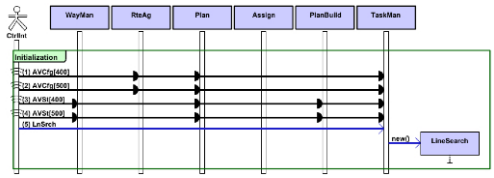


Figure 5.5: The Initialization message sequence flow diagram.

(1) *and* **(2)** *AVCfg[400]* and *AVCfg[500]* are ***AirVehicleConfiguration*** Messages for UAV 400 and 500, respectively. These messages are sent from the ground control station and provide UAV specific parameters used in planning and execution such as UAV operating speed, altitude, and maximum bank angle, as well as, sensor configurations.

(3) *and* **(4)** *AVSt[400]* and *AVSt[500]* are ***AirVehicleState*** Messages for UAV 400 and 500, respectively. These messages are sent to UxAS from the ground control station whenever the ground control station receives them from the UAVs. They provide state information from the UAVs such as, position, actual altitude, actual speed, current waypoint, current active task, and state of the sensors.

(5) *LnSrch* is a ***LineSearch*** message. It defines the Waterway and defines options such as starting the search of the waterway from the given start point or permitting the search to start from either end, see Figure 5.6.

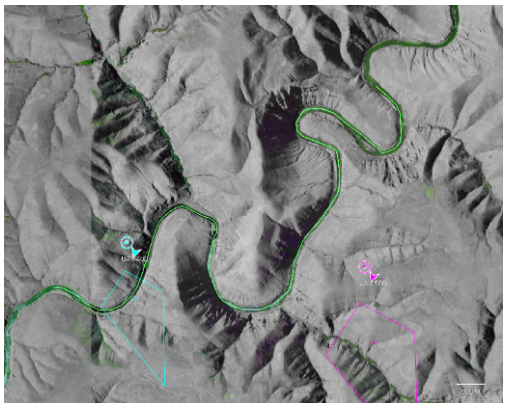


Figure 5.6: The LineSearchTask representing the waterway.

new() Based on the ***LineSearch*** message, the TaskManager service send a ***CreateNewService*** message to the ServiceManager, not shown, to instantiate a new *LineSearch*task service. The ***CreateNewService*** message contains a ***XmlConfiguration*** field that the TaskManager uses to supply the new task with all existing ***AirVehicleConfiguration*** and ***AirVehicleState*** messages. Note: At this point the task can calculate plans for both of the UAVs, but this is delayed until they are informed of the set of candidate UAVs in the ***AutomationRequest***.

ASSIGNMENT In this phase the costs associated with assigning UAVs to tasks are calculated and then UAV to task assignments are determined, see Figure 5.7.

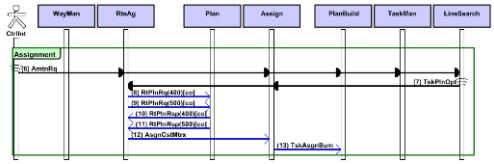


Figure 5.7: The Assignment message sequence flow diagram.

(6) *AmtnRq* is a ***AutomationRequest*** message. It is sent from the ground control station and initiates the assignment process. The ***AutomationRequest*** is a ***AutomationRequest*** includes IDs of the UAVs and tasks to be considered for assignment. If the ID of a task is included, the task is require to generate plans for each eligible UAVs to perform the tasks.

(7) *TskPlnOpt* is a ***TaskPlanOptions*** message. Each task can have optional methods of implementation. In the waterway search

example, the ***LineSearchTask*** has two optional implementations, start from one end or start from the other end. Therefore the task must compute plans for each UAV to peform the task two different ways. This produces four ***TaskOption*** messages and the algebra string: $(V_{1_s} or V_{1_e} or V_{2_s} or V_{2_e})$, i.e. UAV 1 performs the task starting at point s or UAV 1 performs the task starting at point e or UAV 2 perform the task from either end. The ***TaskOption*** messages and the algebra sting are sent out with a ***TaskPlanOptions*** message.

(8) *RtPlnRq[co]* is a ***RoutPlanRequest*** (cost only) message. Once the **RouteAggregator** determines that is has received ***TaskPlanOptions*** from all of the tasks contained in the ***AutomationRequest***, it constructs an assignment cost matrix by sending ***RoutPlanRequest*** messages to the planner.

(10) *RtPlnRsp[co]* is a ***RoutPlanResponse*** (cost only) message. When the planner has calculated costs all of the plans in the ***RoutPlanRequest***, it returns them in a ***RoutPlanResponse*** message.

(12) *AsgnCstMtrx* is a ***AssignmentCostMatrix*** message. When the route aggregator has received all required costs, it constructs and sends out the ***AssignmentCostMatrix*** message.

[(13)] *TskAsgnSum* is a ***TaskAssignmentSummary*** message. When the **Assignment** service receives the ***AssignmentCostMatrix*** message, it runs an optimization algorithm using the ***AssignmentCostMatrix*** and ***TaskPlanOptions*** to assign tasks to UAVs. This results in a ***TaskAssignmentSummary*** message.

Implementation During this phase waypoint plans are constructed based on the assignmnets, see Figure 5.8.

(10) *TskImplRq* is a ***TaskImplementationRequest*** message. When the **PlanBuilder** service receives a ***TaskAssignmentSummary*** it implements the assignments by sending ***TaskImplementationRequest*** messages to the assigned tasks. It sends these requests based on the assignment order in the ***TaskAssignmentSummary***. Each task is required to construct a waypoint plan that will cause the UAV to move from its last position, and heading, to the start of the task and then append any task waypoints.

(15) *RtePlnRq* is a ***RoutePlanRequest*** message. Tasks use the ***RoutePlanRequest*** message to obtain the waypoint plans.

(16) *RtePlnRsp* is a ***RoutPlanResponse*** message. When the planner has constructed all the plans requested in the ***RoutePlanRequest*** message, it send then out in a ***RoutPlanResponse*** message.

(17) *TskImplRsp* is a ***TaskImplementationResponse*** message. The task send back the waypoint plan in ***TaskImplementationResponse*** message. Note: if there are multiple tasks assigned to a UAV, the **PlanBuilder** uses the UAV's last position and heading from the ***TaskImplementationResponse*** as the starting point and heading for the next ***TaskImplementationRequest***.

(18) *AtmnResp* is a ***AutomationResponse*** message. Once the **PlanBuilder** has constructed plans for all of the task assignments, it sends them out in a ***AutomationResponse***, Figure 5.9 shows the final plan to monitor the waterway.

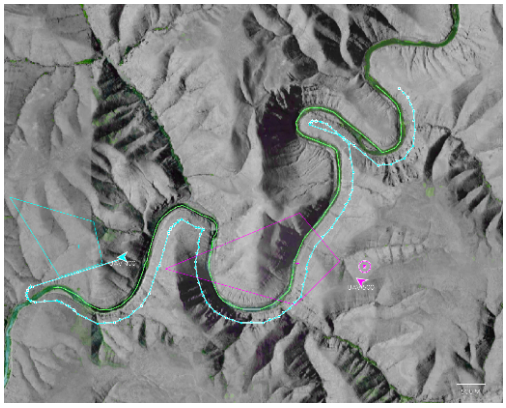


Figure 5.9: The complete set of assigned waypoints.

5.3.3 *Execution*

During this phase, UxAS manages plans and, directs sensor pointing, as the plans are carried out.

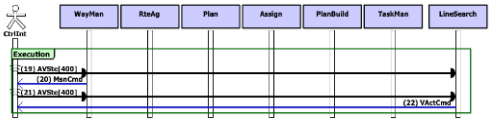


Figure 5.10: The Execution message sequence flow diagram.

(19) *AVSt[400]* is a ***AirVehicleState*** message, from UAV 400. The UAV reports its state to the ground control station which, in turn, send a ***AirVehicleState*** message to UxAS.

(20) *MsnCmd* is a ***MissionCommand*** message. When the **MissionManager** receives an ***AirVehicleState*** message it determines which small set of waypoints the UAV is flying. If the UAV is a given number of waypoints from the end of the small set, the WaypointManager sends out the next small set of waypoints in a ***MissionCommand*** message, see Figure 5.11.

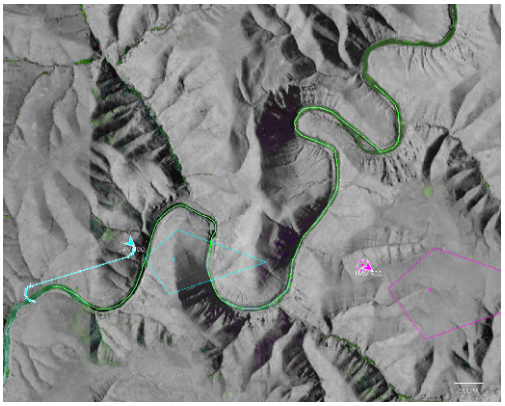


Figure 5.11: Start executing the plan.

(21) *AVSt[400]* is a ***AirVehicleState*** message, from UAV 400. The UAV reports its state to the ground control station which, in turn, send a ***AirVehicleState*** message to UxAS.

(22) *VActCmd* is a ***VehicleActionCommand*** message that implements a ***GimbalStareAction***. The task is responsible for sending any messages necessary to implement its functionality during the execution phase. When a task receives an ***AirVehicleState*** message it checks the *AssociatedTasks* list for its ID. If the task's ID is contained in the *AssociatedTasks* list, the task is *active* and, in this example, based on the UAV's position, calculates a location on the waterway to point the sensor. The UAV is commanded to point the sensor using the ***GimbalStareAction*** message, see Figure 5.12.

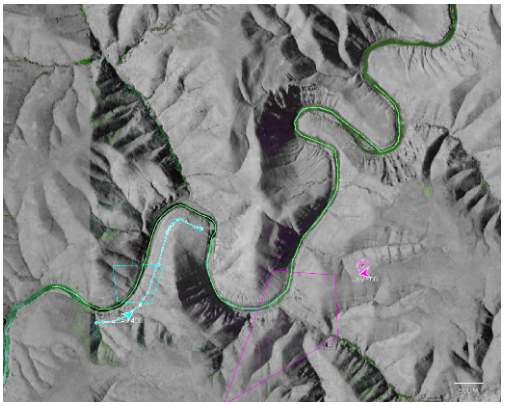


Figure 5.12: Pointing the sensor along the waterway.

5.4 *Distributed Cooperation Example*

(Demonstration of synchronizing planning states to coordinate vehicle/task assignments)

Given a set of Unmanned Air Vehicles (UAVs) and a set of tasks, planning trajectories and assigning UAVs to perform tasks is a difficult problem. If a Ground Control Station (GCS) is able to communicate to all of the UAVs (*fully connected*), then the trajectory/assignment problem can be solved *centrally* and the resulting assignment plans can be sent to the UAVs for execution. However, there are scenarios where complete communication channels do n exist and it is desirable to have a team of UAVs that can decide on trajectory/assignments (*cooperate*) by communicating with each oth in an ad-hoc/distributed manner.

This example demonstrates two copies of UxAS that represent twoUAVs. They communicate with each other using a Zyre bridge, in order to synchronize the UAVs positions and headings (*planning states*) that will be used to determine the trajectories and assignments for the UAVs. Since both copies of UxAS are running exactly the same trajectory/assignment algorithms, synchronizing the inputs to the algorithms will produce the same solutions on both. In this manner, each copy of UxAS generates trajectories/assignments for all UAVs and then implements the trajectories/assignment generated for itself. Both copies of UxAS are running the same algorithms as used in the fully connected case, therefore, this method of cooperation is termed "Centralized Assignment/Distributed Implementation". Figure 5.13 is a plot of the resulting assigned plans for both UAVs, obtained by both copies of UxAS,

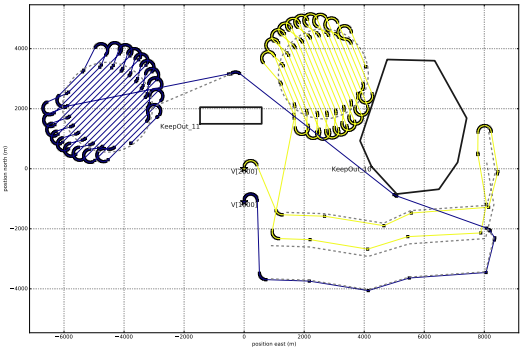


Figure 5.13: Assigned UAV plans.

The following LMCP messages are used to implement cooperation in this example:

*uxas.messages.task.**AssignmentCoordinatorTask*** - task that coordinates trajectories/assignments

*uxas.messages.task.**CoordinatedAutomationRequest*** - sent to all UAVs, to start the coordinated assignment.

*uxas.messages.task.**PlanningState*** - contains the entity state that will be used for planning.

*uxas.messages.task.**AssignmentCoordination*** - sent to synchronize *PlanningState*s with *CoordinatedAutomationRequest*s

In this example there are two UAVs, two keep-out boundaries, and five tasks, see Figure 5.14. In the figure UAVs 1000 and 2000 are represented by stars, keep-out boundaries 10 and 11 are polygons with solid lines, and tasks are represented by dashed lines. There are two area and three line search tasks. The example is designed to bring up two copies of UxAS, allow them to communicate to each other

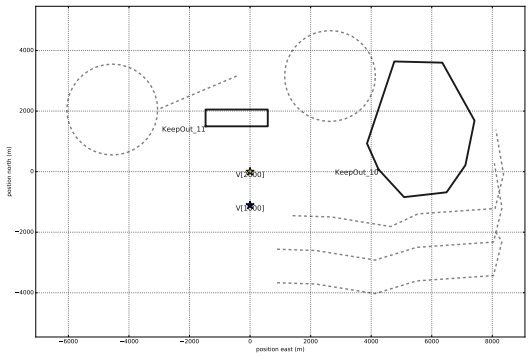


Figure 5.14: Initial UAV positions, requested tasks, and keep-out boundaries.

over a Zyre interface, and exercise the operation of the *Assignment-Coordination* task and associated messages. After the copies of UxAS are initialized a "CoordinatedAutomationRequest", that specifies they coordinate with *three* UAVs, is sent to each. *AssignmentCoordination* tasks will exchange "*AssignmentCoordination*" messages, and since there is no third UAV, wait until the specified *MaximumResponseTime* to construct and send TaskAutomationRequest messages to build plans and assignments for the two UAVs to perform all five tasks. The TaskAutomationRequest messages are constructed using *AssignmentCoordination* messages, which were contained in the *AssignmentCoordination* messages. The *AssignmentCoordination* message contain the state that each UAV will be used for planning. Using these messages, which are synchronized on the *CoordinatedAutomationRequest* ID, makes it possible to synchronize the inputs to the planning/assignment algorithms, and hence obtain a coordinated task assignment/plans for each of the vehicles.

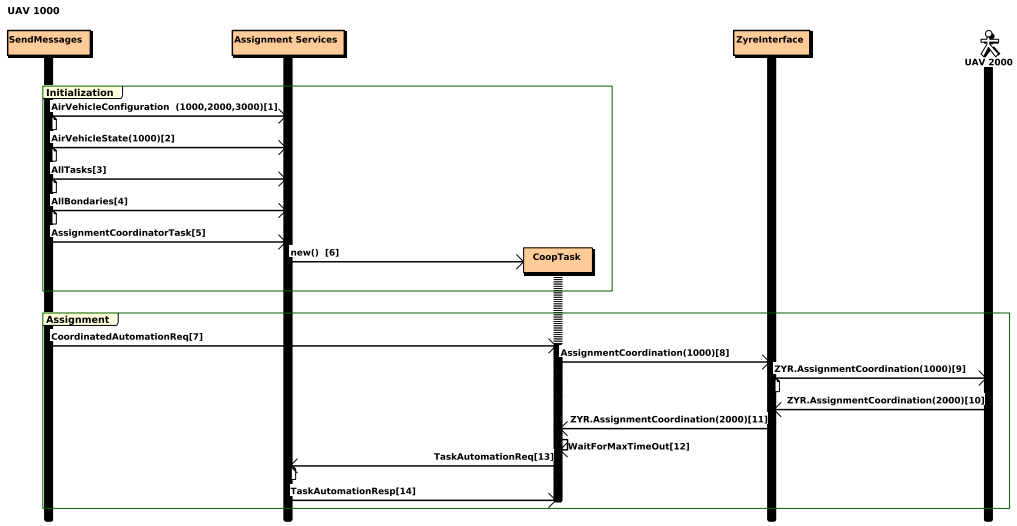


Figure 5.15: Distributed cooperation example message flow.

A an message flow diagram, see Figure 5.15, was created to better explain the events that occur during this example. This diagram depicts the messages important to this scenario. For a more in-depth look a the message flow for assignments see the *Waterway Search* example. The elements of the diagram are:

- The **rectangles** across the top of the diagram, and the one in the center, represent UxAS services, which are described in the next section.
- The **stick figure** labeled 'UAV 2000' is the second UAV and is connected to the current UAV via a network connection.
- The **horizontal lines** represent LMCP messages and the arrowheads show where messages are received. Note: messages are sent in order from top to bottom.

UxAS Services in the Message Flow

SendMessages This service reads messages from files and then sends them out to the other UxAS services at given times. It is used to "load" initial message into UxAS, as well as, time sensitive messages such as the *CoordinatedAutomationRequest*.

Assignment Services These are all the services, discussed in the Waterway example, required to plan and assign UAVs to perform tasks.

CoordinationTask This task (service) manages/generates the messages required to coordinate the assignments.

Zyre Interface Zyre is a ZeroMQ library that finds other entities on the local network and then makes a TCP/IP connection with them.

UAV 2000 This is the second UAV in the example. It has the same services as this one.

5.4.1 *Message Flow Description*

The following sections describe the message flow and how it is used to implement the Distributed Cooperation Monitoring scenario. The message flow is broken into two phases: Initialization and Assignment. Each is described below. Note: this example ends after the assignments have been calculated.

Initialization In this phase all the information required to calculate plans and assignments is passed into UxAS via messages.

[1] AirVehicleConfiguration(1000, 2000, 3000) are the set of ***AirVehicleConfiguration*** Messages for UAVs 1000, 2000, and 3000. These messages provide UAV specific parameters used in planning and execution such as UAV nominal operating speed, altitude, and maximum bank angle, as well as, sensor configurations. All UAVs considered in planning must have an AirVehicleConfiguration.

*[2] **AirVehicleState(1000)*** is ***AirVehicleState*** message for UAV 1000. This is the state that UAV 1000 will used to calculate plans and assignments. States for other vehicle are exchanged in the coordination process.

*[3] **AllTasks*** is the set of messages that define the tasks used in the example. This example demonstrates planning assignment for five tasks, two types of area search and two types of line search (one used twice). The messages in the set are: **AngledAreaSearchTask(ID51)**, **AreaOfInterest(ID100)**, **AreaSearchTask(ID50)**, **ImpactLineSearchTask(ID21)**, **LineOfInterest(ID101)**, **LineSearchTask(ID20)**, **LineSearchTask(ID30)**.

*[4]**AllBoundaries*** is the set of messages that define the boundaries used in the example. The messages in the set define two keep-out boundaries, **KeepOutZone(ID10)** and **KeepOutZone(ID11)**, and the operating region, **OperatingRegion(ID100)**. Note, the operating region defines which boudary ID are active.

*[5] **AssignmentCoordinatorTask*** causes the *TaskManager*, contained in the **AssignmentServices**, to create a new instance of the coordination task.

ASSIGNMENT During this phase, a special automation request is sent in which causes the assignment coordination task to manage cooperation with the other vehicles. Since there is no UAV 3000, the assignment coordination task will delay starting the assignment until expiration of the *MaximumResponseTime*, which is specified in the CoordinatedAutomationRequest message.

*[7] **CoordinatedAutomationReq*** the **CoordinatedAutomationRequest** message contains the requested tasks and eligible UAVs to perform the tasks. Is also has a *MaximumResponseTime* entry that defines the amount of time to wait for inputs from all of the eligible vehicles before initiating the assignment with a sub-set of vehicles.

*[8] **AssignmentCoordination(1000)*** when a **CoordinatedAutomationRequest** message is received, the **AssignmentCoordinatorTask** calculates a state that the (local) UAV will use to complete the request. The state is put into a *AssignmentCoordination* message and sent out.

*[9] **AssignmentCoordination(1000)*** since the **ZyreInterface** subscribed to *AssignmentCoordination* messages, it forwards the message to all connected UAVs.

[10] AssignmentCoordination(2000) when the **ZyreInterface** receives a *AssignmentCoordination* from a connected UAV, it is sent out to the local services.

[11] AssignmentCoordination(2000) as the **AssignmentCoordinatorTask** receives *AssignmentCoordination* messages, it checks to make sure they are associated with the *CoordinatedAutomationRequest* and then stores them. If *AssignmentCoordination* messages have been received for all of the UAVs in the *CoordinatedAutomationRequest*'s *EntityList*, then the **AssignmentCoordinatorTask** generates and sends a *TaskAutomationRequest*.

[12] WaitForMaxTimeOut since there is no UAV3000 in this example, the **AssignmentCoordinatorTask** waits for *MaximumResponseTime* seconds, since receiving the *CoordinatedAutomationRequest*, and then sends the request.

[13] TaskAutomationReq this is the *TaskAutomationRequest* message used to request planning assignment.

[14] TaskAutomationResp once the plans/assignments have been calculated, they are returned in an TaskAutomationResponse message.

5.4.2 *Distributed Cooperation Example - Specifics*

This is an example of running UxAS services that communicate with each other to synchronize plannning and assignment of vehicles to tasks. The copies of UxAS exchange 'AssignmentCoordination' messages which contain the vehicle positions and headings that each vehicle will use for planning and assignment. Since the copies of UxAS are running the same assignment software, synchronizing the inputs will also synchronize the outputs. Each copy of UxAS runs the assignment algorithm for all known vehicles and implements the assignments corresponding to its own ID, this is using a "centralized" algorithm in a "decentralized" implementation.

Files

cfgDistributedCooperation_????.xml - UxAS configuration file for vehicle ????

runUxAS_DistributedCooperation.sh - bash script to run the example

The '*MessagesToSend*' directory contains files with xml encoded LMCP messages that are sent in to UxAS using the **MessagesToSend** service.

MessagesToSend/AirVehicleConfiguration_V????.xml - the air vehicle configurations for vehicles 1000, 2000, and 3000.

MessagesToSend/AirVehicleState_V????.xml - initial air vehicle states for vehicles 1000 and 2000.

MessagesToSend/AngledAreaSearchTask_51.xml - an IMPACT angled area search task.

MessagesToSend/AreaOfInterest_100.xml - the area that AngledAreaSearchTask_51 will search.

MessagesToSend/AreaSearchTask_50.xml - a CMASI area search task.

MessagesToSend/AssignmentCoordinatorTask.xml - the task that coordinates vehicle states before the assignment

MessagesToSend/CoordinatedAutomationRequest.xml - the automation request used in conjunction with the AssignmentCoordinatorTask

MessagesToSend/ImpactLineSearchTask_21.xml - an IMPACT line search task.

MessagesToSend/KeepOutZone_??.xml - polygons that represent areas that the vehicle are not to enter.

MessagesToSend/LineOfInterest_101.xml - points of the line for ImpactLineSearchTask_21.

MessagesToSend/LineSearchTask_??.xml - a CMASI line search task.

MessagesToSend/OperatingRegion_100.xml - the operating region, i.e. set of keep-in and keep-out tasks, to be used in the assignment.

Running the Example

1. open a ternimal window in the directory: *examples/03_Example_DistributedCooperation/*
2. enter the command: *./runUxAS_DistributedCooperation.sh*

What Happens?

- Two console windows will open, each will have UxAS running.

FOR EACH COPY OF UxAS:

- By the end of the first second, all air vehicle configurations and states, as well as all tasks and associated messages are sent in to UxAS using the 'SendMessages' service.
- Each 'LmcpObjectNetworkZeroMqZyreBridge' will make a connection with the other copy of UxAS.

- At two seconds an air vehicle state, corresponding to the entity ID, is sent to UxAS. This state must be sent to UxAS after the assignment coordinator task, so the task has an air vehicle state from the local vehicle.
- At five seconds the coordinated automation request is sent in which starts the assignment process.
- After receiving the coordinated automation request, the assignment coordinator task send out an 'AssignmentCoordination' message containg the state that the local vehicle will use for planning.
- The 'AssignmentCoordination' is picked up by the 'LmcpObjectNetworkZeroMqZyreBridge' and sent to the other running copy of UxAS
- The assignment coordinator recieves the 'AssignmentCoordination' message from the UxAS and checks to see if its time to send in a TaskAutomationRequest. The 'CoordinatedAutomationRequest' specifies three vehicle IDs and since the the third vehicle is not present, the assignment coordinator must wait untile the specified has passed, 10 seconds from recipt of the request, to sent out the 'TaskAutomationRequest'.
- Once the timer has expired, a 'TaskAutomationRequest' specifing two vheicles is sent out. This causes the UxAS services to calculate assignments for both vehicle.
- The resulting assignment can be plotted using the python scripts located in the sub-directory of:

  ```
  03_Example_DistributedCooperation/UAV_1000/datawork/AutomationDiagramDataService/
  ```

Things to Try

- Edit the automation request,

  03_Example_DistributedCooperation/MessagesToSend/CoordinatedAutomationRequest.xml

  and comment out, or remove the entry '*<int64>3000</int64>*' in the '*EntityList*'. Rerun the example. Since it will have '*AssignmentCoordination*' for all of the requested vehicle, the assignment coordinator will not have to wait until the end of the '*MaximumResponseTime*' to start the assignment process.

Testing

6.1 Running automated tests

6.2 Adding New Functional Test

New Functional Test Step-By-Step

1. download and install GoogleTest from:

   ```
   https://github.com/google/googletest
   ```

2. create a directory for the test:

   ```
   mkdir uxas/code/tests/Test_Services/00_ExampleTests/01_HelloWorld/
   ```

3. add a UxAS configuration file and inputs files to the new test directory. IMPORTANT: Since functional test run from the folder:

   ```
   uxas/code/builds/netbeans/nb_8.2/FunctionalTests
   ```

   any path entered in the configuration must be relative to that folder, e.g. :

   ```
   ../../../../tests/Test_Services/00_ExampleTests/02_Test_Example_WaterwaySearch/MessagesToSend/
   ```

4. make a copy of the functional test template source code:

   ```
   cp uxas/code/tests/Test_Services/GTestFunctionalTestTemplate.cpp
   uxas/code/tests/Test_Services/00_ExampleTests/01_HelloWorld/01_HelloWorld_test01.cpp
   ```

5. edit the new functional test source code to modify, at minimum, the following:

   TestName - change the test name.

   testPath - add the relative path to the new directory from:

   ```
   "uxas/code/builds/netbeans/nb_8.2/FunctionalTests/"
   ```

   uxasConfigurationFile - add the relative path, including file name to UxAS configuration file

   EXPECT_EQ - add/edit the google test statements necessary to test the results

6. add the new functional test to netbeans:

   (a) open the "*Test Files*" folder under the "*FunctionalTests*" project

   (b) (optional) add/open a logical folder: "right-click" on the "*Test Files*" folder and choose "*New Logical Folder ...*"

   (c) add a new "Test Folder": "right-click" on a logical (or "*Test Files*") folder and choose "*New Test Folder ...*"

   (d) add the test source file to the new Test folder: "right-click" on the new "*Test*" folder and choose "*Add Existing Item ...*"

Contributing

7.1 Code of conduct for contributors

All contributors are encouraged to follow the code style recommendations, and are required to follow the outlined branching system. These guidelines are outlined in the following sections.

7.2 Code style recommendations

7.2.1 Background

UxAS grew out of numerous research project and can still seem pieced together. A similar style across the code base can make it easier to navigate through the software. A list of our preferred style and best practices are captured in two documents:

- Practices
- Style

The top guidelines are as follows:

- Use smart pointers, never bare
- Use the full namespace for clarity
- Minimize logic in constructors
- Use C++11 types (e.g. int32_t rather than int)
- Prefix member variables with 'm_'
- Booleans start with a verb, i.e. isOpen
- Use abbreviations sparingly
- Any complex logic should be in the .cpp file, not the .h
- Use 4 spaces instead of tab
- Append units to variable names (e.g. distance_m)

- Braces on own lines
- Keep parameters local, not in global files
- Document with Doxygen comments

7.3 *Branch description and git flow expectations*

The OpenUxAS branching model addresses the following concerns:

- We have a stable branch that always builds and passes tests
- Multiple collaborative teams can proceed with their development independently
- Discrete features can be contributed to the main line of OpenUxAS development, and these can be integrated into other teams' ongoing work
- Until OpenUxAS is public, all teams can use the `afrl-rq` organization's Travis-CI account for continuous integration

To address these concerns, OpenUxAS uses a variant on the Git Flow, GitLab Flow, and GitHub flow models.

Because OpenUxAS does not yet have a fixed cycle of releases, we do not need the additional complexity of `hotfix/` and `release/` branches present in Git Flow. However, since a number of collaborating teams work on OpenUxAS simultaneously, it makes sense to have long-lived branches for each team, rather than only having feature branches and a stable branch.

This README does not go into detail about the various Flow models, but instead provides instructions for common scenarios. We encourage you to read about the Flow models to get more of a sense for the "why"; here we are focusing on the "how".

7.3.1 *Quick Overview*

The repository will typically have a branching structure like the following:

- `master`
- very stable, only updated by pull request from `develop`
- `develop`
- stable, only updated by pull request from feature branches
- `teamA`

- team branch for Team A
- stable at the discretion of Team A
- updated by merging in feature branches and `develop`
- `teamA-feature1`
- feature branch for Team A
- when finished, merged into `develop` via pull request
- `teamB`
- `teamB-feature1`
- etc.

7.3.2 *Team Branches*

The team branch is the branch off of which your team will work. It serves the role of the `develop` branch of Git Flow or the `master` branch of GitLab and GitHub Flow. This branch is never intended to be directly merged back into `develop`, but feature branches based off of it will be.

If you have experience with these models, this concept probably seems odd. Eventually, we would like to replace these team branches with entire repo forks for each team, but until OpenUxAS is public, this would prevent forks from using the `afrl-rq` Travis-CI account.

7.3.3 *Create New Branch*

Start by creating a new branch that will serve as the active development branch for your team. This step should only be necessary once for your team; this branch is meant to be long-lived as opposed to a feature branch that is quickly merged in and deleted.

```
$ git checkout develop
$ git checkout -b teamA
```

7.3.4 *Updating*

You will want to regularly incorporate the latest changes from the `develop` branch in your team branch. This reduces the pain when merging your team's changes back into `develop`.

Start by making sure your local `develop` branch is up-to-date:

```
$ git checkout develop
$ git pull
```

Then merge the updated `develop` with your team branch:

```
$ git checkout teamA
$ git merge develop
```

7.3.5 *Feature Branches*

Feature branches are shorter-lived branches meant to encompass a particular effort or feature addition. These branches will be the means for you to incorporate your team's changes into the main `develop` branch via pull requests.

Feature branches will always be based off of your team branch, so if your team branch has commits you would like to see in `develop`, you can simply create a new feature branch and begin the pull request process right away.

7.3.6 *Naming*

To help the OpenUxAS maintainers know which branches belong to which teams, feature branches should be named using your team name as a prefix, for example `teamA-feature1`.

7.3.7 *Creating*

Create a feature branch by checking it out off of your team branch. Note that it will save you some effort at the later merge to update your team branch from `develop` first.

```
$ git checkout teamA
$ get checkout -b teamA/feature1
```

7.3.8 *Merging to Team Branch*

For a long-running feature branch, you may want to occasionally merge it back into your team branch so it can be shared within your team before it's ready to be merged into `develop`.

```
$ git checkout teamA
$ git merge teamA/feature1
```

7.3.9 *Merging to develop*

You cannot directly merge a feature branch into `develop`, because it is protected. Instead, open a pull request from the feature branch into `develop`, and your changes will be merged after review.

It is a good idea to update your team branch and delete your feature branch once it is merged into `develop`.

```
$ git checkout develop
$ git pull
$ git checkout teamA
$ git merge develop
$ git push origin --delete teamA/feature1
$ git branch -d teamA/feature1
```