

An Update on the Development of the Modular Mission Planning Toolkit (MMPT)

Robert W. Nitzel¹ rnitzel@technologysystemsinc.com
Matthew Haag¹ mhaag@technologysystemsinc.com
Chance Yohman¹ cyohman@technologysystemsinc.com
Chuck Benton¹ cbenton@technologysystemsinc.com
Rick J. Komerska² komerska@ausi.org
Steven G. Chappell² chappell@ausi.org

- 1) Technology Systems, Inc. (TSI), Fort Andross, 14 Maine Street, Box 41, Brunswick, ME, 04011 – USA, 207.798.4646
- 2) Autonomous Undersea Systems Institute (AUSI), 86 Old Concord Turnpike, Lee, NH, 03824 – USA, (603) 868-3221

Abstract

This paper presents the details of the ongoing work within the development of the Modular Mission Planning Toolkit (MMPT) application. Early versions of this application have been tested in simulation and in several operational experiments conducted by team members of the Multiple Cooperating AUV (MCAUV) team, composed of personnel from TSI, AUSI, Naval Undersea Warfare Center (NUWC)-Newport, University of New Hampshire (UNH), Rensselaer Polytechnic Institute (RPI), and Falmouth Scientific, Inc (FSI).

The MMPT project has focused on providing a feature rich operator interface tool for multiple AUV planning, monitoring and control. Development of the MMPT application has been closely coupled with the development of the Solar-powered Autonomous Undersea Vehicle (SAUV) developed by members of the MCAUV team. Through this close collaboration, we have developed an open framework capable of providing feature rich design implementations by potential 3rd party developers. Currently we have produced

a number of plug-ins which includes SAUV command generation, AUV status display and processing, environmental forecasting display, mission planning, AUV route prediction based on environmental forecast data, and Augmented Reality Visualization of the Common Operational Picture (ARVOPtm).

Within this paper, we will examine three plug-in frameworks developed to create the MMPT application. These frameworks include a GIS plug-in API, a communication plug-in API and a Graphical User Interface (GUI) plug-in API.

In June of 2007, the MMPT development team attended AUVFest 2007 in Panama City, FL and supported three fielded SAUVs running cooperative data collection missions. A prototype version of MMPT was used in this operation. This paper will present the results of this operation in regards to the mission planning, monitoring, and playback functionality of the MMPT architecture.

Introduction

This paper discusses the continued development of MMPT. The team consisting of TSI, AUSI, NUWC-Newport, UNH, RPI, and FSI has virtually and practically tested the system. MMPT provides planning, monitoring, and control for the SAUV. We use an open system that commands the SAUV, AUV status, routes, forecast data, and an augmented reality display.

We also examine the three frameworks that support MMPT. These frameworks are Communication Tools, GIS, and the GUI. Additionally we will discuss the API involved in creating a GUI plugin.

Finally, we wrap up with a summary of AUVFest 07. We detail MMPT's mission planning, monitoring, & playback of the SAUVs cooperative behavior.

Geographic Information Systems (GIS) Framework

At TSI, we use a high level architecture to plug-in multiple GISs. The architecture has the following functionality:

- ◆ Setup
- ◆ Preferences
- ◆ Visualization
 - Drawing the map
 - Panning
 - Zooming in & out
 - View Rotation
 - Multiple View Support
- ◆ Calculations
 - Screen Coordinate and Latitude & Longitude Conversion
 - Navigational Aid Search

This is an evolving architecture that meets the needs of our current GISs, C-Map¹ & CJMTK² (Commercial Joint Mapping Toolkit), a derivative of ESRI's ArcGIS.. Our architecture's vision is to plug-in additional GISs (e.g. Autodesk, Intergraph, Grass, GeoTools, uDig, and more). It will also provide enhanced functionality- 3-D view, view over time, and support of multiple data formats and projections.

The plug-in architecture now uses a base class known as TSIMap. It defines the above functionality in virtual C++ functions. We use virtual functions, so we can override functions with the appropriate code from the API of C-MAP, CJMTK, or the GIS we are plugging in. From these definitions, it is able to load the function declarations from a dynamically linked library (DLL). This gives us the capability to use the GIS on any system we choose to install MMPT on.

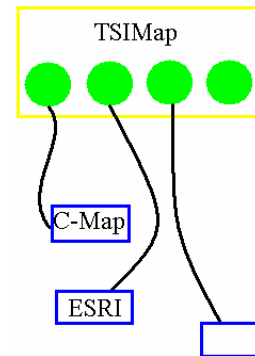


Figure 1- Plug-in Architecture Visualized

Why does TSIMap need the capability to use any GIS system? It is needed to deal with emerging GIS systems. There also exists the issue of working with GIS systems in classified environments.

¹ <http://www.c-map.no/>

² <http://www.cjmtk.com/>

Furthermore, interoperating with a GIS familiar to the user can result in satisfaction.

C-Map is an electronic charting technology. It is produced by C-Map Norway. Their CM93/3 Software Development Kit allows users to access a local database of geographic information. For MMPT, this information is of the nautical type. This includes coastline and bathymetry data. It can be adapted to include terrain data also.

CJMTK is the Commercial Joint Mapping Toolkit. It is the defense-enhanced version of the ArcGIS suite. The CJMTK piece that corresponds to C-Map is the ArcGIS Engine. They are both embeddable, or have the ability to provide a map image in an application. Through programmatic manipulation, the image can be zoomed, panned, rotated, or duplicated. Additionally, unseen calculations are performed to convert between image coordinates and world coordinates and query buoy, beacon, landmark, and daymark data.

C-Map plugs in the following way. Setup is performed via the Component Object Model (COM). COM is a way of reusing code. Initialization is performed using its standard practices. Drawing the map makes use of TSI's WingLib. WingLib is an off-screen buffer that provides smooth panning & zooming of the map.

The remaining functionality is handled by querying C-Map's COM interfaces for projection and searching its database of nautical aids.

CJMTK attaches to the architecture by using ArcObjects. ArcObjects is the programmers' access to the geographic tools provided by CJMTK. Setup is performed using COM and Smart Pointers. Drawing the map involves the use of handles to device contexts and passing those to the application for use. Map movement involves interaction with the IMapDocument and IDisplayTransformation interfaces. Querying of navigational aids is facilitated by the ILayer interface.

Our vision for TSIMap is to perfect the integration of CJMTK. This will keep us compatible with the military's integration of the software. At the same time, we want to explore alternatives such as Grass, GeoTools, and uDig. This will keep us updated on the bleeding edge of the technology. As well, we want to push the technology further. We want to provide traffic light analysis. This will allow the operator to easily see based on current data and depth data where it is safe (green), caution must be exercised (yellow), and it is unsafe (red).

Communications Framework

Because a requirement of MMPT is the communication with various devices via serial ports there has been an extensive development of how such data is handled by the Application. When a user requires the use of a COM port they create what is called a "Com Chain" which creates a linked series of DLLs to handle the incoming serial information. Each DLL is optional; however, if no DLLs are loaded in the COM chain the raw serial data will simply be dumped into the Communications Terminal Console window of the main application. This can be useful for debugging serial

connections, or for connecting to the SAUV via the RF link, however it is rarely useful for programmatic data transfer. There are three DLL types, a Verification Wrapper (VerWrp), a Protocol (Prctl), and a Command Engine (CmdErg). When running SAUV operations with MMPT the corresponding selections are Frames, COFSNET, and CCL2, however in reality they can be anything.

A Verification Wrapper is the simplest of these DLL types, both in terms of normal function and actual API. It verifies the data within the received communications packet is complete. It will hold incomplete data until it receives the next data transmission to attempt to stitch together disconnected data. It acts in the opposite sense on an outgoing packet. It will wrap the outgoing packet in a verification layer so it can be verified within the receiving system.

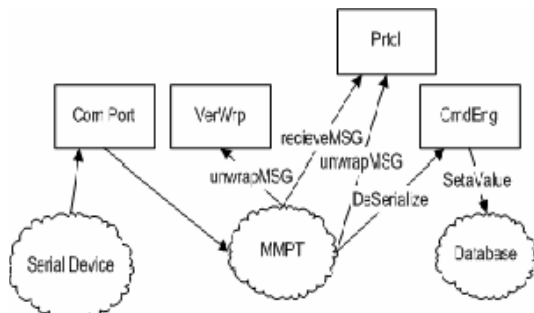


Figure 2- Com Chain Visualized

The DLL API calls that MMPT will call to invoke this behavior are (note, function calls, messages, and classes are bolded for clarity) **unwrapMSG** and **wrapMSG** respectively. Each return a string which is unwrapped or wrapped by the verifier. For SAUV operations, Frames2, an AUSI developed checksum protocol is utilized here. Another example of a verification wrapper

concept is the WHOI Micro Modem (with either NUWC/AUSI Common Control Language (CCL) or WHOI CLL), in the Verification Wrapper, where the NMEA data in the Micro-Modem could be stripped out and the binary hex data converted to actual data in memory and returned for later pieces of the Communications Chain. Clearly the method of verification is left to the third party developer; however, possible options include CRC16, CRC32, or MD5 depending on desired security of verification and many networking issues.

A Protocol DLL is what it sounds like, a DLL to act as a networking protocol as needed over serial communications. This is where Protocols Such as COFSNET and AUSNet³ exist. The access to a Protocol DLL is more complex than a Verification Wrapper. This is for good reason.

While it is often thought that MMPT is the end point for all communications, and it often is, this is not always the case. Typically MMPT communications come via the use of a gateway buoy located close to the operational AUVs. Under most Ad-Hoc networking schemes, including COFSNET and AUSNet, this implies that MMPT is a full feature node in the wireless network, and thus may be depended upon to act as an intermediary hop to two AUVs that could not communicate otherwise. This means that a message received up to this point in the Com Chain could be destined for another node and not for us.

³Sai Mupparapu, Radim Bartoš and Matthew Haag, (2005) "Performance Evaluation of Ad Hoc Protocols for Underwater Networks", in Proceedings of the Fourteenth International Symposium on Unmanned Untethered Submersible Technology, Durham, NH, August 2005.

While the ability to do this is optionally removable within MMPT, it is enabled by default.

This means that there needs to be two sets of accessors to this tool. The DLL API for this function consists of **unwrapMSG**, **wrapMSG**, **receiveMSG**, and **sendMSG**. Data is promoted up the chain from the verification wrapper, to the protocol that MMPT will pass data to via **receiveMSG**. The Protocol does as needed with the data. The Protocol will then store the payload of the packet, if it is the proper type of data, as well as flag it for rebroadcast or response as needed.. MMPT will then after **receiveMSG** call **unwrapMSG** to retrieve the message if it is in fact for this node of its network and send the payload up the Com Chain.

MMPT has a timer which regularly checks if any of the loaded protocols need to transmit data. It does this by regularly calling **sendMSG**, which if a protocol needs to transmit it will fill the provided buffer with the data needed to be sent. MMPT will then go down the chain with this information, to the Verification Wrapper.

The final component of a fully formed Com Chain, and arguably the most important, is the Command Engine. The command engine is the most important component because it is the end of the chain, and therefore where the data is by convention stored in the database for the application to have access to. There is no reason that a lower piece of the Comm Chain couldn't store data in the database, or otherwise alert other tools of the application, however MMPT expects that data could be passed up the chain to the command engine.

The Command Engine is also the only component of the Comm Chain that MMPT will interface with to generate commands to be sent down the chain. Traditionally, in SAUV Operations, the Command Engine is a AUSI/NUWC Common Control Language based DLL, alternatively for other MMPT applications, there exists a NMEA command engine to parse and store NMEA data from attached serial devices, such as GPS and heading sensors. In both cases, the incoming serial data, after traveling through the proper chains arrives in the command engine and is stored within the database. Other Tools, discussed later, observe the changes in the database, and alter their internal data state, and data display accordingly.

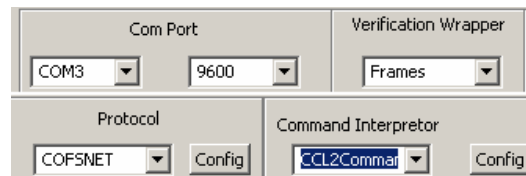


Figure 3- Com Chain Setup

There are 4 key API functions that MMPT Calls within a Command Engine, **DeSerialize**, **VerifyMessage**, **CompileMessage**, and **ShowCommGen**. **DeSerialize** is the end point of the Com Chain, after all previous pieces it is where the final steps are taken to produce useable data out of incoming information.

CompileMessage and **VerifyMessage** are a component of the Communications Terminal Console. If a Command Engine possesses the ability to encode strings, such as CCL2 does, then raw text commands can be entered manually. **VerifyMessage** verifies that the message entered into the console is properly formed for encoding, and

CompileMessage generates the proper data to be sent down the Com Chain to the eventual end receiver. This is so quick input options can be made, rather than journeying through the GUI based interface to designing messages that are created by MMPT invoking **ShowCommGen**. This interface is meant to provide an interface for the simple creation of Commands for remote assets, such as AUVs, but is part of the Command Engine. Because the Command Engine contains significant GUI code it is required to maintain a **ToolCmdMsg** method which ensures the proper distribution of Windows Messages to its internal buttons and windows.

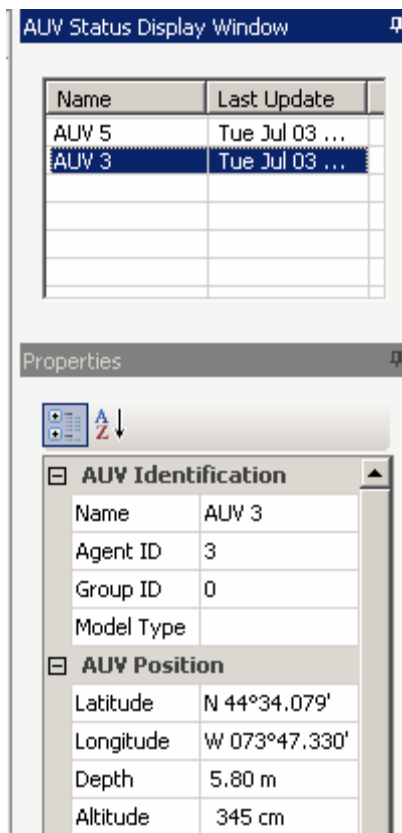


Figure 4- Example Application Tool : AUV Status Display

Application Tools

The major components of MMPT, what enable nearly all planning and monitoring functionality, are the Application Tools. Application Tools while similar to Com Tools differ in both scope and interaction. While Com Tools only interact with data that is being received or transmitted, Application Tools react to internally generated data, user generated data and received data.

There are four key components required for a tool to fully interoperate with the greater MMPT application. The Application Tool API, which provides the means to create a tool specific sidebar and toolbar. The Database Interface, which provides access to an observer enabled in memory database cache. The Messaging System, which is an internal MMPT message passing API. Finally, there is the 2-D Drawing System, a system that allows objects to be drawn on the MMPT map view.

The first piece to discuss is the Application Tools API. This is where basic tool operations, including the creation of toolbars and control bars happen. For easy manipulation of the GUI by end-users, and to maintain a constant look and feel, MMPT GUI components are built using Prof-UIS⁴, a third-party MFC extension library. Users of Visual Studio .Net and higher will recognize the MMPT interface as the same interface that is used within Studio, and to a lesser extent, Office as well. This provides MMPT the ability to have easily dockable components, auto hidden components, and a constant look and feel between executions.

⁴ <http://www.prof-uis.com/>

In **InitTool**, a tool desiring a controlbar must create a **TSIControlBar** instance and create a dialog bar class inherited from **CExtNCW** **<CExtResizableDialog>** and then create the instance of the dialog class with a pointer to the control bar. Finally the Control bar must be docked to the provided dock. The creation of a toolbar doesn't vary from the standard method of adding a toolbar, except that the toolbar class must be inherited from **CExtToolControlBar**. There are also certain esoteric registry calls which can be found in the extensive MMPT documentation. These calls ensure that Prof-UIS properly stores all settings of the control bar and tool bar for each time MMPT is loaded.

There are three other API functions that merit discussion here: **ToolCmdMsg**, **UpdateToolLocation**, and **MMPTDestroy**. **ToolCmdMsg** is identical to the function discussed within the Command Engine section, and is even more imperative to Application Tools, as without it none of the toolbar or control bar buttons will operate. What needs to occur in this function is to call the **OnCmdMsg** method for all Tool global objects, i.e. toolbars or controlbars, with the same arguments as the function receives. **UpdateToolLocation** informs a tool of a change to the center location of the map view, however it is most useful because it is first called after the 2-D Drawing System has completed initializing. So, it is a good place to initialize drawn objects. This will be discussed in-depth in the 2-D Drawing System section. **MMPTDestroy**, acts as a destructor for the tool, notifying the tool that it will soon be unloaded from MMPT. Within this

function it is recommended at all on close actions be handled for the DLL.

The MMPT Message System is one of the two key interconnected features of MMPT, the Database Interface being the other. It was designed to provide a C++ friendly alternative to the existing Windows Messaging system. Rather than using obscure structures, it utilizes inheritance in several ways to enable any class to send nearly any form of object to any other class anywhere within MMPT. The Messaging Systems uses two basic class types, **iBase_Plugin** and **iBase_Message**. **iBase_Message** is the base message type from which all messages are inherited. It possesses a string identifier defining its type. This type accomplishes two things. First, any class which inherits **iBase_Plugin** is capable of subscribing to a list of message types, so the type provides information to the messaging system to inform it of to whom the message needs to go. Second, type implies to receiving class what to cast the message to. It is important to mention that messages are not standard pointers, but rather are of type **TSI::sharedpointer**, a polymorphic pointer container. To receive messages a class must inherit from **iBase_Plugin** and implement **Post_Message**. Inside **Post_Message**, it must properly handle all subscribed types.

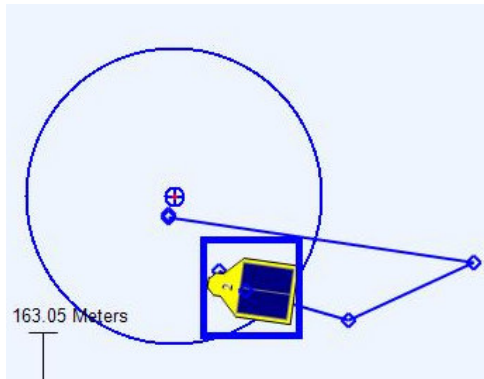


Figure 5- Example Drawing and Icon

One of the most important applications of the Messaging System is the ability to send messages to the 2-D Drawing System. It is through these messages that tools are able to draw onto the 2-D map view. The 2-D Drawing System takes five types of messages, but for clarity we will only discuss four:

M_ADDOBJECT,
M_REMOVEOBJECT,
M_REFRESHBUFFER, and
M_CENTERVIEW.

M_ADDOBJECT displays an object on the map. **M_REMOVEOBJECT** removes an object from display on the map. Depending on the setting of the redraw flag, the map will be updated to show or remove the objects.

M_REFRESHBUFFER redraws the onscreen buffer. **M_CENTERVIEW** centers the view at the provided coordinate.

The final piece that must be discussed with the Messaging System is **CObjectBase**. **M_ADDOBJECT** and **M_REMOVEOBJECT** each take a **TSI::sharedpointer** to an instance of a **CObjectBase** class. For a tool to create an object to be drawn on the map view it must create its own class inherited from **CObjectBase** and is presented with two options to draw on the screen. Either the

inherited object can set **CObjectBase::m_FileName** to any GDI+ compatible image type and call **CObjectBase::setLoc** to have an image drawn or it can overload **CObjectBase::Draw** to draw directly to the screen buffer every time it is redrawn or both. The choice depends on the goals of the tool for the object. For example, to display AUV positions on the screen the first method is used because the location of the AUV is distinct. To display current and wind data, the second is used because the current and wind data is available for all spots.

Database Interface

The final piece of MMPT interconnectivity is the Database Interface. It is a wrapper for various features existing around a PostgreSQL back end. Functionally, the database is accessed similarly to a perl hash object, where a **CString** is used as a key through **GetaValue** and **SetaValue** functions to grab a value for a specific entry. These keys are formed using the formula "tablename.rownumber.columnname."

To ensure that changes to the in-memory cache are properly reflected back to the underlying database there is a small set of control functions: **MakeRow**, **EditRow**, **EndRowWrite**, and **EndRowEdit**. The first two correspond to a traditional **Insert** and **Update** and flag a row as in the process of being changed. The second two correspond in order with the first two to flag the writing to the new or updated row to be complete, and hence ready to be written back to the underlying database.

Useful interaction with the cache is accomplished through three classes: the **TableInfo** Class, the **Variant** Class, and the **Observer** Class. The **TableInfo** Class is a container filled with all of the setting and information for a given database table. This class is populated in two ways. First, if a table already exists in the database at the launch of MMPT, a sister table exists in the database containing all the relevant information to populate the structure. Second, when a user creates a table all of this information is provided within the XML document that the user supplies the interface with beforehand.

According to the the MMPT database XML schema, each column or **Element** must contain a **name**, **type**, **description**, and **order**, with the optional tags of **full_name**, **precision**, and **unit**. The tag **name** represents the name of the column within the database; **type** can be one of five supported database types: **int**, **float**, **varchar**, **bigint**, and **bool**; **description** is simply a description of the column; and **order** represents the value's display setting. When using a prewritten display code a column with an **order** of 0 will not be displayed. The tag **full_name** represents the desired display name of the column, while standard column names must conform to SQL standards the name displayed for the column need not, **precision** represents the desired display precision of the value of the column, and **unit** is simply the units of the column if they exist.

All of this information is accessible though the **TableInfo** Class which has array **mInfo** of the structure **tInfo** which contains all of this information in clearly named members. The **TableInfo** Class for each table in the database can be

accessed through the use of the table's name as a key through **GetaValue**. The **Variant** Class on the other hand is the key to the cache's unique ability to take and put any data via **GetaValue** and **SetaValue**. **Variant** is a heterogeneous container, which through judicious use of the **type** tag can be used to contain any of the database supported types though each **type** option has a specific C++ type associated with it: **int** maps to a **long**; **float** maps to a **float**; **varchar** maps to a **CString**; **bigint** maps to a **__int64**; and **bool** maps to a **bool**.

Use of types other than these will lead to information not being properly transitioned to the underlying database and may cause undefined behavior of the MMPT application. To transition from a **Variant** type to a specific underlying type one must utilize a **variant_cast** similar to a normal C++ casting call. The **Observer** Class presents the most interesting and useful pieces of this database methodology. An **Observer** is a base class which observes **Subject** classes and when the observed class is altered in anyway the observing class is notified via an inherited function **Notify**. Because **Variant** is inherited from **Subject**, any class within MMPT can attach itself to observe any piece of the database and hence receive updates when it changes. It is particularly useful to observe the entry in the cache for a table's name. This will notify the observer that the table has been edited or expanded recently enabling the observing class to update its display. This functionality is utilized by the **AUVStatusDisplay** to update SAUV locations on the screen without being directly informed by the **CCL2CmdEng** that new information has been added to

the data, it simply is informed by the database cache itself.

AUVFest 2007 Operations

This year AUVFest was in Panama City Beach FL. The MMPT team as provided with good facilities by the Naval Support Activities (NSA) Base. The utilized site was Beach Site 4 (BS4). In-water testing was conducted just off the beach within a 1.5 by 2.5 km rectangle. The area of operations was in line of site to BS4 and enabled the use of the gateway buoys as communications nodes for the whole operation. There was limited connectivity directly to the vehicles, due to their low profile in the water and the use of FreeWave modems within BS4.



Figure 6- SAUV Operations Center at AUVFest

Despite various developmental challenges, SAUV vehicles operations were very successful. While the environmental plug-in tool was not deemed releasable to the SAUV operations computer, it did run successfully on a secondary laptop, and was able to provide a screenshot of the operation with currents shown for the user.

The operational goal for the three SAUVs running a cooperative survey mission involved three roles: *survey*, *recharge*, and *networker*.⁵ The vehicles were to decide among themselves at runtime which was to run the survey, which was to simply recharge, and which was to become a communication gateway node for the other two. This new gateway role was defined to be positioning a vehicle to a place advantageous for becoming a networking gateway between the RF and acoustic communication realms. The geometry of the experiment is shown in Figure 7. Major in water operations were planned for Monday and Tuesday June 11th and 12th.

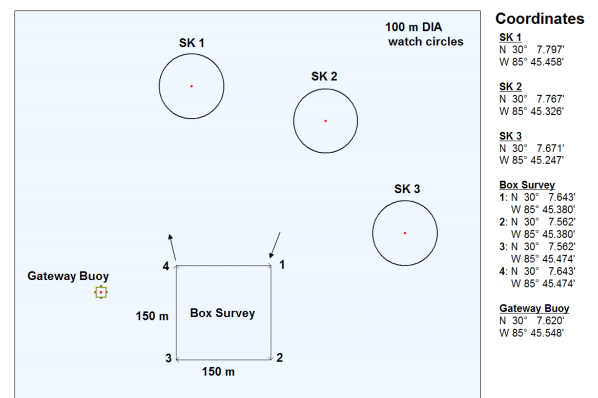


Figure 7- Planned SAUV 3-Vehicle Cooperative Survey Configuration

Unfortunately, Monday June 11, was marred by communication issues on the vehicles. When all 3 vehicles were operating and reporting status, one would stop updating. It appeared that the vehicle logic was sending packets but they were not being heard in the water. The focus was trying to get all three vehicles communicating at the same time, so attempts to rectify it

⁵ Chappell et al Recent Field Experience With Multiple Cooperating Solar Powered AUVs

consumed all the time in the water for the day. Around 4 o'clock thunderstorms began to roll-in and operations were cut.

On Tuesday, June 12th, the decision to run with only 2 vehicles as necessary was determined given the more urgent need to show cooperation. This provided good operational parameters for the day. From 7:34 am to 8:50 am, various attempts were made to diagnose the previously mentioned communication issues. A second gateway buoy was connected to inspect raw information that was being communicated within the water. It was verified that SAUV 4 was attempting to communicate, but no information was sent through the water during its transmission time. So operations went on with SAUV vehicles 2 and 3.

Vehicle 3 decided to run the box survey first, determined by cost analysis done within the processes running on the vehicles. The cost analyses for these tests are based on the available energy that each vehicle has, since vehicle 3 had the most energy it decided to run the mission first. This was executed by the user, sending a "CoSurvey" (Cooperative Survey) Start mission command. This command was broadcast to all vehicles simultaneously. At this time SAUV 3 stopped communicating status updates. The vehicle was commanded to stop, while the comms issue was investigated. It was determined that the modem was in command mode, this would interfere with the transmission of any data from the vehicle, all messages were sent to the modem command line, instead of the output buffer. The modem was reset on vehicle 3, and the vehicle sent the

"CoSurvey" command again, it then proceeded to run the box survey. SAUV-3 started running the mission at 10:22, verified by status update to MMPT.

The following is a transcript from the original notes taken during operations: (Edited for clarity, and brevity)

10:37 -- SAUV-3 was on last leg of Survey. The chase boat noticed the vehicle acting squirrely in the water, it was doing donuts, or loops around in circles. After 3 minutes we notice the vehicle still within the same location area at the status report time. Looks like the vehicle is doing donuts on the MMPT screen, the updates are slow, 3 minute updates for each vehicle. So now we have confirmation that the vehicle is "disabled" and are starting to formulate a contingency plan.

10:55 – We determine the best course of action was to put vehicle 3 in a watch circle pre-planning the watch circle down current, as the vehicle is disabled and floating with the current at this point. We plan on MMPT a new watch circle at Lat: 30.125100 Lng: -85.756017. This data was put into a yUpdate file for the "station_keep" command.

10:59 -- Sent the file, acoustically to the vehicle.

11:08 – Confirmed that the vehicle successfully updated it's internal command file. The vehicle is trying to run station keep, at the new location, but is having thruster/actuator issues and cannot keep station. At this time SAUV-2 has noticed that SAUV-3 is no longer running "CoSurvey" so it starts running the Cooperative Survey.

11:12 – Sent Stop mission command to SAUV-2. This will allow the chase boat to put SAUV-2 in tow. SAUV-2 will continue to provide periodic status updates. The Chase boat attaches to SAUV-2 and proceeds to tow the vehicle towards the gateway buoy. The boat is between the gateway buoy and the vehicle, so we are not receiving updates during this transition, due to noise in the water.

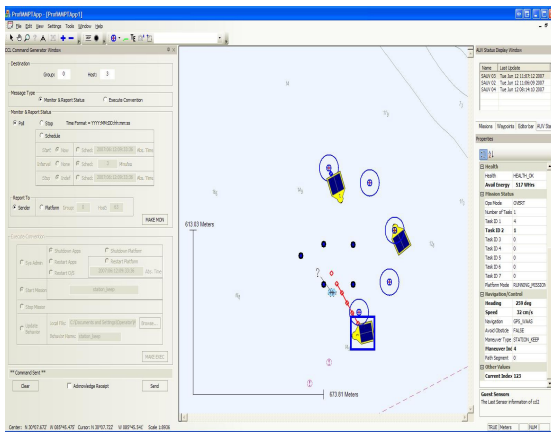


Figure 8- Screenshot showing SAUV-3 running “station_keep” and SAUV-2 starting “CoSurvey”

11:22 – Trying to get SAUV-4 up and running. Polling for status from SAUV-4 reveals its energy is at 1008.

11:30 – Last time we have heard from SAUV-2.

11:32 – Request periodic status. Energy levels: SAUV-2 1153; SAUV-3 551; SAUV-4 1007.

Chase boat has inadvertently crossed the survey box while towing SAUV-3, which may have stepped over SAUV-2 communications in the water. For about 20 minutes we were not able to get status from SAUV-2, so it ran survey box again. Need to determine if it heard the

last status message from SAUV-4, it was still higher in available energy than SAUV-4. During the second time around the survey, the available energy went negative and the status value grew to 65,000 (Overflow for the size of bytes we are sending for available energy). Now if we try to get SAUV-4 in the game they will not swap because it appears SAUV-2 has the highest energy.

11:57 – At the last leg of the survey we received a status message from SAUV-2 at which point we sent a run “station_keep” command.

12:03 – Confirmed SAUV-2 is running “station_keep”. We lost SAUV-4 for one or two status packets, but it is back.

12:24 – SAUV-3 moored back to its original SK-2 watch circle. SAUV-3 energy value is 2122. It appears to be fully recharged. SAUV-4 energy is 990. SAUV-4 does not seem to be recharging.

13:00 – Operations stopped due to impending storm. Vehicles left moored.

By backing up from the operations computer and restoring the database to another laptop, it was possible to recreate the mission events for the day. Also, using the provided environmental forecast for the day, comparisons were made of the events to the forecasted data to determine how MMPT would have performed in planning a mission using the environmental data. Here is a screenshot showing the results at the time we lost maneuverability with SAUV-3 on its last leg of the survey.

Some analysis of the predicted environmental data versus the observed state of SAUV-3 drifting was undertaken

immediately following AUVFEST07. Based upon the previously discussed situation with vehicle 3, and for the purpose of this analysis, we are to assume the SAUV was not operating its propeller, and was drifting over the surface.

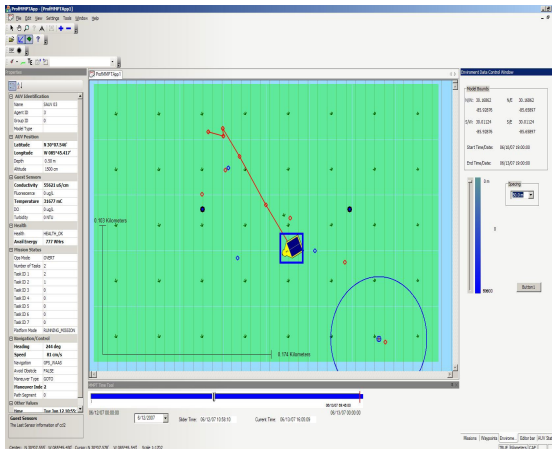


Figure 9- Screenshot showing the environmental data compared with the drifting SAUV.

The MMPT team was provided with two fine grain data sources for AUVFest 2007. These were used to compare predicted versus actual ocean currents. Predicted currents came from two sources: NCOM-COAMPS (NRL-Stennis), and a single NAVO daily prediction based upon the meteorological briefings provided for AUVFest.

The predictions generated came from the utilization of linear interpolations from the four closest provided data points in each of the models. The data in the models was similar in some ways, but quite different in others. Furthermore, the “Sample nc file from Hydromap to Swan Grid” known as “hydro_pan_061107” contained a variable sigma, which current data was

dependent on, but we were unable to find documentation for. We asked the staff providing the briefing data to us, however they did not know the proper use of this variable either. For purposes of this comparison we used a sigma of 2 for data generation, which best matched predicted behavior to the currents provided in a printout.

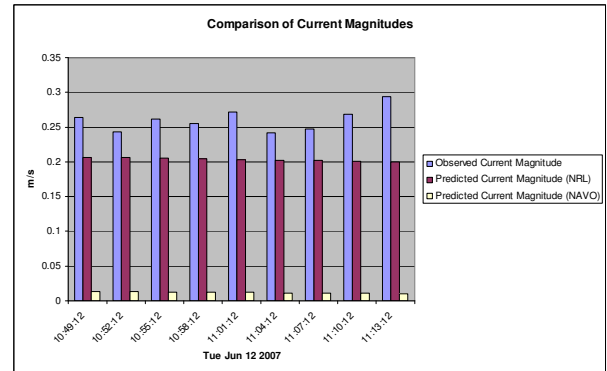


Figure 10- Chart Comparing Current Magnitudes

As the above chart shows, the NAVO originated data was significantly deviant from the observed data. This is possibly due to the previously mentioned unknown “sigma” value, however all sigma values yielded current magnitudes on a similar order of magnitude. It seemed likely that there was some additive or multiplicative modifier that needed to be applied to this data. Further support of this concept comes from the fact the the NRL-originated data contained within the self documented NetCDF file an appropriate scaling factor. This assumption is validated by the fact that while the NAVO current magnitude was off by an Average of 95.5% compared to the NRL data at 21.6% the Direction of the current as predicted by both, when compared to the Observed direction, differed on average 10.7 degrees for

NAVO and 10.8 degrees for the NRL data.

It must also be noted that the modeling utilized is inherently simplistic. To reach the conclusion regarding predicted destination the current estimation is done at the time of the vehicle being at the preceding point, and this speed is assumed to remain constant until the next known AUV position. Likewise, observed current information is derived from a similar assumption finding the distance between the two points and dividing by the update interval.

Conclusions

This paper discussed the continued development of MMPT. The team consisting of TSI, AUSI, NUWC-Newport, UNH, RPI, and FSI have virtually and practically tested the system. MMPT provides planning, monitoring, and control for the SAUV. We use an open system that commands the SAUV, AUV status, routes, forecast data, and an augmented reality display.

We examine the three frameworks in this paper that support MMPT. These frameworks are Communication Tools, GIS, and the GUI. Additionally we discussed the API involved in creating a GUI plugin.

Finally, we wrapped up with a summary of AUVFest 07. We detailed MMPT's mission planning, monitoring, & playback of the SAUVs cooperative behavior.

Future Work

The future work for MMPT lies chiefly in tool development with one

core architecture alteration needed. First, the environmental tool needs to be expanded to incorporate a more complex model of force interaction and communication between the MMPT team and the originators of the environmental prediction data improved. These two pieces will enable the environmental data to be utilized to a further extent, enabling more accurate predictions of vehicle-environment interactions. Second, the ties between the mission planning tools and the command engine will be strengthened, removing the present external step regarding the transmission of mission plans to in-water assets.

A third issue that needs to be addressed is the CJMTK GIS Plugin. This includes CJMTK's handling of Digital Nautical charts and an issue where CJMTK takes control of the current working directory. Finally while steps have been taken to enable multiple MMPT platforms to operate within the same database, this software is still incomplete, and the need to complete this represents the core architectural work remaining within MMPT.

Furthermore, there are two ongoing projects within TSI that utilize the MMPT framework to extend beyond the underwater realm. These projects are the Augmented Reality Visualization for the Common Operating Picture (ARVCOP) program and the Planning and Execution Augmented Reality System (PEARS). Both of these technologies aim to fuse unmanned asset and sensor information with Augmented Reality to provide an improved view of the operating picture for the warfighter. There is currently work undergoing at TSI to implement a Compact Control Language command engine to meet

customer desire for the display of Hydroid Remus AUVs as a component of these projects. Additionally these projects contain their own tools which are being developed within the MMPT Architecture.

References

1. <http://www.c-map.no/>
2. <http://www.cjmtk.com/>
3. Sai Mupparapu, Radim Bartoš and Matthew Haag, (2005) "Performance Evaluation of Ad Hoc Protocols for Underwater Networks", in Proceedings of the Fourteenth International Symposium on Unmanned Untethered Submersible Technology, Durham, NH, August 2005.
4. <http://www.prof-uis.com/>
5. Chappell, Steve, Rick J. Komerska, D. Richard Blidberg, Christiane N. Duarte, Gerald R. Martel, Denise M. Crimmins, Michel A. Beliard, Robert Nitzel, James C. Jalbert, and Radim Bartoš, (2007), Recent Field Experience With Multiple Cooperating Solar Powered AUVs in Proceedings of the Fifteenth International Symposium on Unmanned Untethered Submersible Technology, Durham, NH, August 2007