DSAAV - A Distributed Software Architecture for Autonomous Vehicles

Mandar Chitre

Acoustic Research Laboratory, Tropical Marine Science Institute,
National University of Singapore, 12A Kent Ridge Road, Singapore 119223.

***Abstract*- Autonomous Underwater Vehicle (AUV) technology has matured over the past few decades but commercial AUVs today remain complex, proprietary and expensive. Modularity in AUVs at a software, electronics and mechanical level allows users to configure AUVs for specific missions by only including the required components. With multiple base AUVs, users may easily configure heterogeneous teams of AUVs for collaborative missions. Modular AUVs are also easier to maintain. We expect that open-architecture AUVs with open software/hardware interfaces, changeable modules and open source components will become widely available in the future. However AUV configuration management and module compatibility are issues that arise with modularity.**

An initiative at the Acoustic Research Laboratory (ARL) of the National University of Singapore (NUS) has yielded an open-architecture collaborative prototype AUV known as STARFISH. The software components in this AUV are based on the DSAAV architecture. DSAAV has been designed ground up with modular AUVs in mind. In a DSAAV compliant AUV, each module provides a uniform software interface that other AUV modules can access. This interface allows configuration of the module, logging of critical information, discovery of services, access to sensor & actuator services, health monitoring and automated software update functionality. The interface is rich in functionality, yet light weight and portable to ensure that even low power micro-controllers can easily implement it. DSAAV can be implemented on any underlying communication backbone such as Ethernet, UDP/IP, etc. The software components running under DSAAV are independent of the underlying communication backbone and function without change in various AUVs and simulation environments.

In this paper, we describe the basic philosophy and concepts behind DSAAV. We also outline the Application Programming Interface (API) for DSAAV compliant systems and describe its key functionality. It is our hope that DSAAV will be adopted and extended by other AUVs in the future.

I. INTRODUCTION

Over the past few decades Autonomous Underwater Vehicle (AUV) technology has matured from research prototypes to commercial systems. However, commercial AUVs today remain complex, proprietary and expensive. As AUVs mature further, we expect that open-architecture AUVs with clearly documented open software/hardware interfaces, changeable hardware modules and open source components will become widely available for research and commercial use. Some open interfaces are already emerging in the field [1][2]. An initiative at the Acoustic Research Laboratory (ARL) of the National University of Singapore (NUS) has yielded an open-architecture collaborative prototype AUV known as STARFISH [3]. The software components in this AUV are based on the DSAAV architecture described in this paper. Although DSAAV was developed with AUVs in mind, it is equally applicable to other autonomous vehicles such as unmanned surface vessels (USVs).

Modularity in AUVs at a software, electronics and mechanical level provides significant benefits to the user. Changeable modules allow users to configure AUVs for specific missions by only including the required components. By adding optional AUV sensor modules to a set of basic AUVs, users may easily configure heterogeneous teams of AUVs for collaborative missions. When hardware failures occur, modular AUVs are easier to maintain than monolithic AUVs. Although modularity provides these benefits, it also has some costs. AUV configuration management and module/component compatibility are issues that any modular AUV designer has to address. Additionally a robust communication backbone is needed to ensure that all modules/components can communicate effectively.

The Mission Orientated Operating Suite (MOOS) developed at MIT addresses theses problems to some extent by the use of a central database with a well-defined communication protocol for client software components to access it [1]. MOOS adopts a "star" architecture with all components connecting to a central database in order to deposit or retrieve information. This can potentially lead to a single data bottleneck and a single point of failure. DSAAV, in contrast to MOOS, adopts a peer-to-peer communication architecture. By being distributed, DSAAV spreads the load and traffic across all processors in the AUV and avoids high load and dependency on single processors and databases. However a centralized system is easier to administer and monitor; for this reason, DSAAV provides a central configuration database and logging service. As the configuration database is required primarily during start-up and the logging service is not essential to the operation of the AUV, the advantages of a distributed architecture are not lost by having these centrally managed services. MOOS uses a TCP/IP as its communications backbone. This limits the use of MOOS to

The work presented in this paper was supported through a project grant from Singapore's Defence Science & Technology Agency (DSTA).

processors and operating systems that can support a TCP/IP stack (e.g. Linux, Windows NT and Windows 2000). DSAAV can operate on a variety of communication backbones including raw Ethernet; this makes it extremely lightweight and suitable for implementation on micro-controllers without a TCP/IP stack or multi-threading support as well as single-board computers running Linux or Windows.

DSAAV has been designed ground up with modular AUVs in mind. In a DSAAV compliant AUV, each module provides a uniform software interface that other AUV modules can access. This interface allows configuration of the module, logging of critical information, discovery of services, access to sensor & actuator services, health monitoring and automated software update functionality. The interface is rich in functionality, yet lightweight and portable to ensure that even low power micro-controllers can easily implement it. The interface is easily extensible through Remote Procedure Call (RPC) constructs and therefore provides forward compatibility.

DSAAV can be implemented on any underlying communication backbone. In the STARFISH AUV, the communication backbone used is Ethernet. By using raw Ethernet packets and avoiding the overheads of TCP/IP, DSAAV implementations on micro-controllers are small and fast. In simulation environments, DSAAV can be implemented over POSIX message queues or UDP/IP. The software components running under DSAAV are independent of the underlying communication backbone and function without change in various AUVs and simulation environments. In fact many STARFISH components achieve platform independence using DSAAV and can be deployed on Linux, Windows, Mac OS X systems or micro-controllers with no change in source code. This provides immense flexibility for AUV deployment.

In this paper, we describe the basic concepts underlying DSAAV. We then outline the Application Programming Interface (API) for DSAAV compliant systems and describe its key functionality. Through examples, we illustrate the use of DSAAV in AUVs and provide guidance for implementation. It is our hope that DSAAV will be adopted and extended by other AUVs in the future.

II. ARCHITECTURE

A. *Architectural Overview*

DSAAV is a four-layer architecture as depicted in Figure 1. The bottom-most layer (*IComms*) provides an implementation of a unreliable messaging service over the communications backbone available. The next higher layer is the "RPC" layer which implements a remote procedure call semantic using the IComms messaging service. The third layer consists of framework and sensor/actuator services implemented using the RPC framework. This includes core services for vehicle configuration, logging and health monitoring. It also includes hardware drivers for all the sensors and actuators (collectively known as *sentuators*) as well as an external communications (*EComms*) interface for communication to other vehicles and/or the control center. The EComms interface may potentially use CCL for the external communication [2]. The top layer houses the command & control components which utilize the services provided by lower layers to achieve the mission of the vehicle.

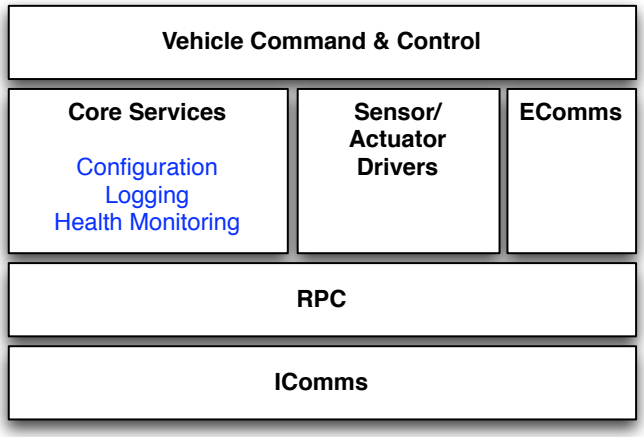

Figure 1. DSAAV's 4-layer architecture

B. *IComms Layer*

The IComms layer abstracts the underlying communications infrastructure from the upper layers. It provides an API to send and receive messages containing a set of key-value pairs. The keys are predefined numeric constants, while the values may be integers, single-precision floating point numbers, double-precision floating point numbers of null-terminated strings.

The Unified Modeling Language (UML) [4] representation of the API is shown in Figure 2. The API consists of a ParamSet class which holds key-value pairs. A Message extends this class to add in special keys for message id, message type and reference id. The reference id is used for messages which refer to other messages (such as replies and acknowledgements). Messages can be sent and received using the IComms interface class. If a destination is known, its address is used to send the message. Alternatively, a channel broadcast may be used to send the message to all clients subscribing to the channel. The destination address is represented by the MsgSvcAddr class and consists of a hardware address and a logical port number. The port number allows multiple services to run on the same hardware node.

Various implementations of the IComms interface may be available for use. For example, in the STARFISH AUV, we have Ethernet based IComms for Linux, Mac OS X and STR912 micro-controller. We also have an IComms implementation over UDP/IP for WiFi access to the AUV and an IComms implementation using POSIX message queues for a simulation environment on Linux / Mac OS X.

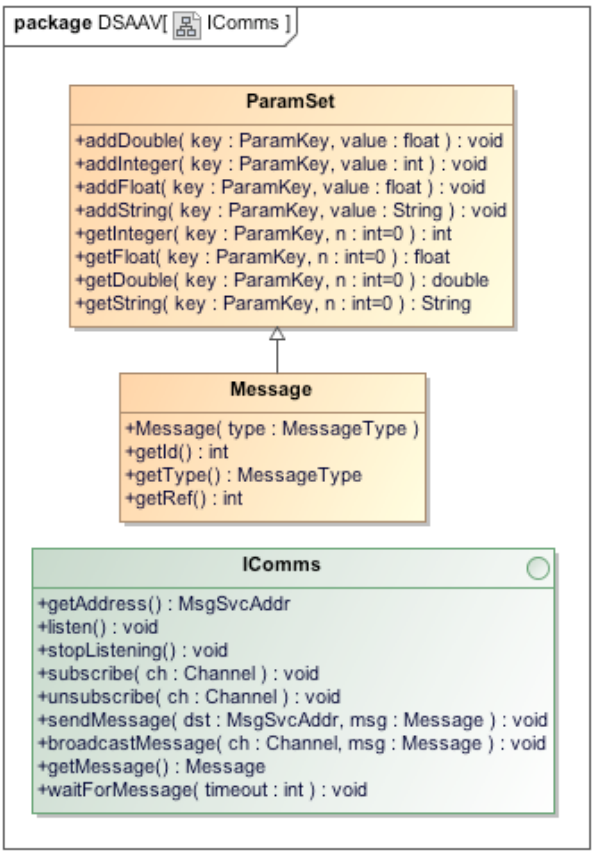

Figure 2. The IComms API

C. *RPC Layer*

The RPC layer implements a procedure call semantic using the unreliable messaging service provided by the IComms interface. The procedure to be called is identified by a destination address and a RPC operation id. A set of parameters may be passed to a procedure call, and another set of parameters may be optionally returned by the call. In cases where the procedure call does not return any parameters, the caller may choose a blocking reliable call or a non-blocking unreliable call. The reliable mode is used for most calls and is implemented through the use of acknowledgements and retries. Unreliable calls may be useful in scenarios where latency is important but reliability is secondary - for example, logging of routine data in the AUV or sensor data notifications for time-sensitive data streams.

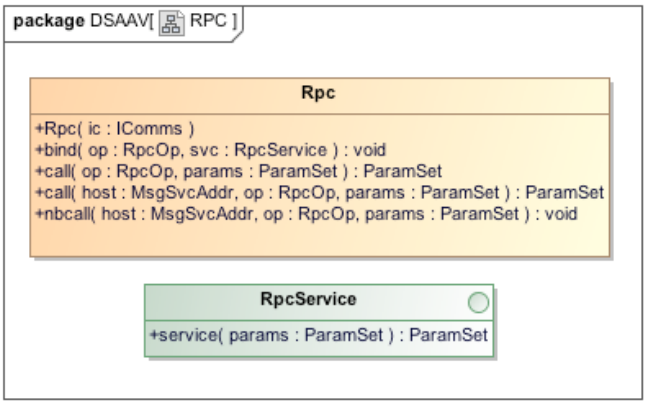

Figure 3. The RPC API

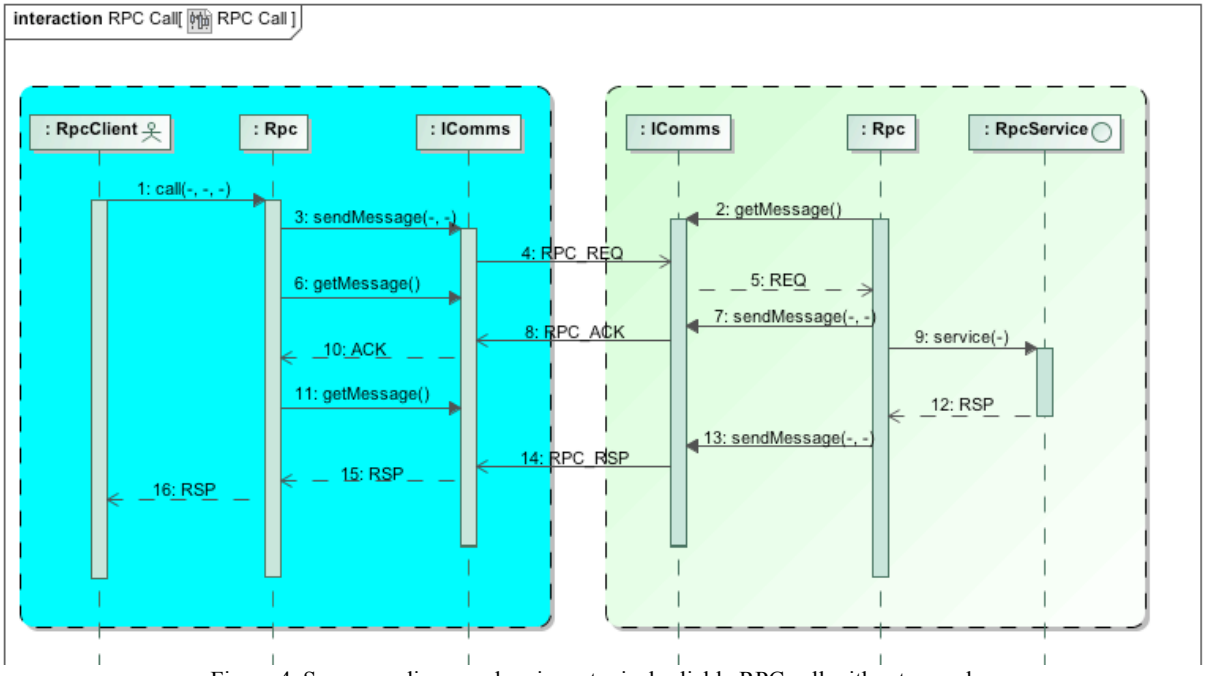

Figure 4. Sequence diagram showing a typical reliable RPC call with return value

Figure 3 shows the API of the RPC layer. The RpcService interface is implemented by software components providing procedures to be called over RPC. The call method without a destination address uses a RPC broadcast channel to discover the service to be called. This is typically used to discover the configuration server, which in turn provides information about all other services in the AUV as explained in the next section. Figure 4 shows a sequence diagram for a typical reliable RPC call with return value.

D. Core Services

A set of core services are implemented using the RPC layer described in the previous section. The core services provide a common framework for managing all software components in the vehicle. There are three core service - the configuration service, logging service and health monitor. These are described in the sub-sections below.

1) Configuration Service

The configuration service enables each component to be initialized with appropriate settings. It also enables the component to store persistent data (such as adaptive control parameters, etc.) centrally, even if the component runs on a micro-controller with no persistent storage. Finally, it determines the service binding between components, thus providing the "plumbing" between components for data to flow.

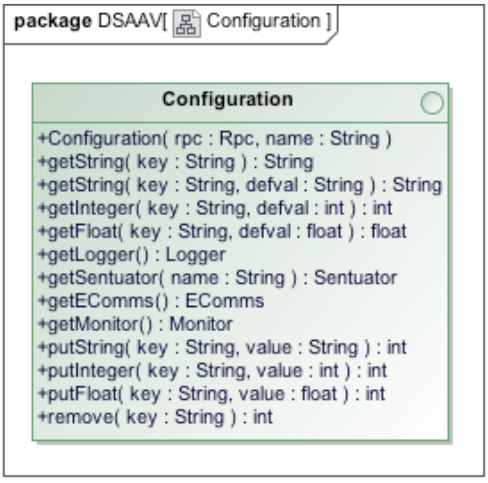

Figure 5. The configuration API

The configuration API is shown in Figure 5. The central configuration database consists of two parts - a user administered *configuration data* and a software controlled *application data*. The getString, getInteger and getFloat methods provide access to the information in the both parts; if the same key is present in both parts, the application data is returned. The putString, putInteger and putFloat methods put value in the application data, masking any value with the same key in the configuration data. The remove method removes the application data section and unmasks any configuration data with the specified key. The getLogger, getEComms and getMonitor methods provide access to the logger, external communications interface and the health monitor respectively. The getSentuator method provides access to sensor/actuator services in the vehicle. The addresses of all services are obtained from the configuration database.

The STARFISH implementation of the configuration service reads the configuration data from a text file. It maintains a separate text file database for the application data. The configuration data file has sections for each software component. These sections contain key-value pairs representing each setting. The data file also has sections for each service available. These sections have a "Server" key providing the address of the service. The values in these sections can be overridden by keys in the component's section, allowing a flexible way to control data flow. This best illustrated through an example configuration file extract:

```
[Logging]
Server = 1:0           # running on node 1 port 0
LogLevel = LOG_INFO    # default log level

[Elevators]
Server = 2:0           # driver on node 2 port 0

[Depth]
Server = 3:0           # depth from node 3 port 0

[DepthSensor]
Server = 4:0           # driver on node 4 port 0

[DepthFilter]
Server = 3:0           # filter on node 3 port 0
Depth.Server = 4:0     # get depth data from sensor

[MyDepthController]
ControlGain = 2.3      # gain control parameter
Logging.LogLevel = LOG_DEBUG  # override log level
```

The scenario described by the above configuration file extract is shown in Figure 6. The scenario includes five software components running on different hardware nodes in the AUV and communicating to control the depth of the AUV.

The configuration file starts off defining the address of the logging server and the default log level as INFO for all components. However, the MyDepthController component overrides the log level to DEBUG for troubleshooting. The address of the elevator actuator service is defined next; any component requesting the "Elevators" sentuator will access the service from this address. This is followed by the address of the depth information service. In this example, we have a DepthFilter which filters the depth data from the sensor to provide a better estimate of real depth. Hence the depth information service points to the DepthFilter service rather than the DepthSensor service; when a component asks for the "Depth" sentuator, it'll receive data from the DepthFilter. However, the depth information for the DepthFilter comes from the DepthSensor as the "Depth.Server" key is overridden in the DepthFilter section. Finally, the MyDepthController

section defines a ControlGain parameter which can be read by the component to configure itself. This simple example illustrates how the configuration file is used to provide "data plumbing" between components.

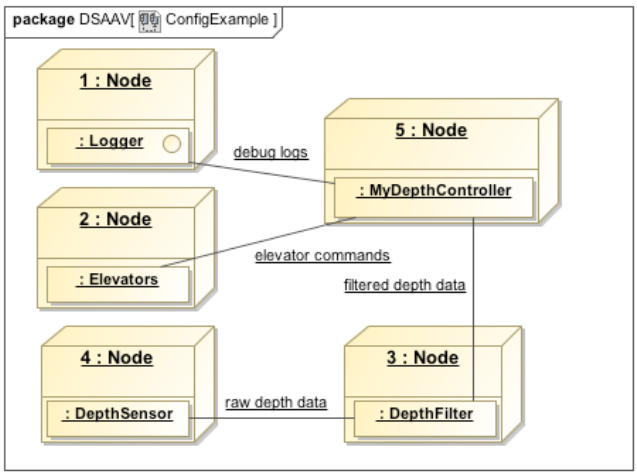

Figure 6. Scenario described by the sample configuration file extract

1) Logging Service

The logging service provides a central store for logs from all software components in the AUV. The logging API is shown is Figure 7.

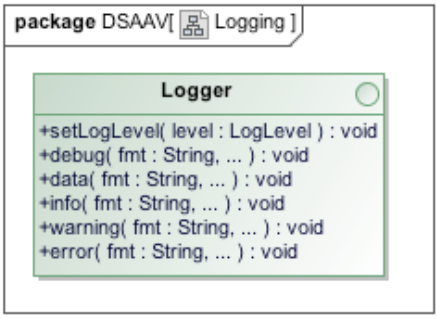

Figure 7. The logging API

The log level of each component can be set through the configuration service or via software. This allows different level of logging for each component, allowing the user to control the details logged from the component. Five different log levels are defined:

- ERROR - Used for logging critical/unrecoverable errors.
- WARNING - Used for logging potential problems and warnings; the software component is expected to continue operation.
- INFO - Used for logging informational messages such as version information, state changes and other major events.
- DATA - Used for logging sensor data; these messages are logged using unreliable calls to the logging server to avoid a performance bottleneck when large volumes of data is generated.
- DEBUG - Used for logging debug information; typically enabled temporarily on components for troubleshooting.

A central logging server enables software components running on micro-controllers without persistent storage to log information. Additionally the logs from all components are stored in a single place chronologically, allowing easy analysis. The logging server may also provide additional functionality such as log rotation, real-time monitoring and archival.

2) Health Monitor

Fault-tolerance is important in autonomous vehicles. To help the command & control system to achieve this, the DSAAV architecture recommends a health monitor component that keeps track of the health of all other components in the vehicle. In addition it provides a overall system health status. The health monitor API is shown in Figure 8.

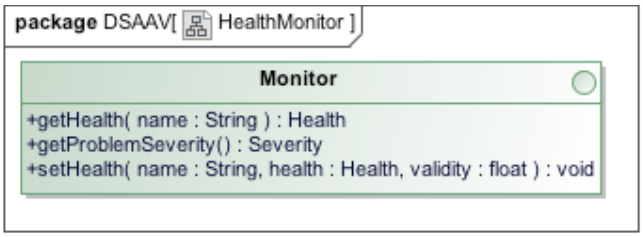

Figure 8. The health monitor API

Each component in DSAAV provides a health status update to the health monitor. This can be "active" i.e. provided on a regular basis by the component or "passive" i.e. polled by the health monitor. Based on the health updates, each component status can be one of the following:

- HEALTHY - Component working as expected.
- UNAVAILABLE - Component is responding but no data is available. An example of this state is a GPS driver with no GPS fix or a DVL driver with no bottom-lock.
- MALFUNCTION - Component is responding but has detected a malfunction in software or hardware.
- OFFLINE - Component is not responding (passive mode) or has not sent a health update recently (active mode).

Based on the health of each component, the health monitor computes a overall health status or problem severity level. This information may be used by higher level command & control algorithms for decision making. The severity level can be one of the following:

- NONE - Normal operation.
- WARN - There is a problem with the vehicle, but it is not a show-stopper. The mission can be continued.
- ABORT - There is a problem which requires the mission to be aborted. The vehicle should stop the mission and return to the recovery point.
- EMERGENCY - There is a problem which requires emergency action. The mission should be aborted and the

emergency action should be taken. For an AUV, the emergency action could be to rapidly surface by dropping ballast weights and sending a radio SOS message to the control center.

In STARFISH, the health monitor uses an eXtensible Markup Language (XML) file to control the behavior of the monitor. The file specifies the components to be monitored, whether they are to be active/passive, the timeouts and the severity levels as a function of the component health. This diagnostic information is made available to the command & control system as well as the diagnostic Graphical User Interface (GUI) at the control center. A screenshot of the diagnostic GUI is shown in Figure 9.

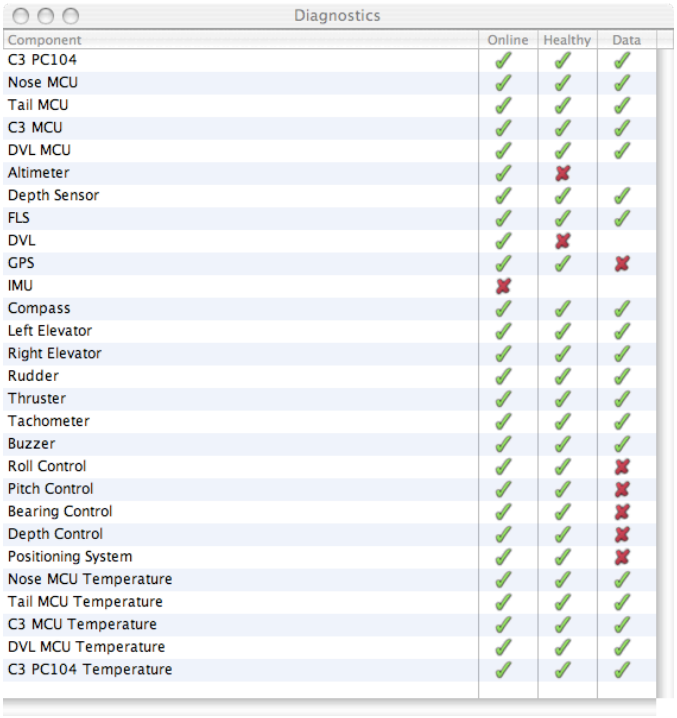

Figure 9. Screenshot of the diagnostic GUI

E. *Sentuator Services*

1) *Sentuator Drivers*

The sensors/actuators in the vehicle are accessed via the sentuator drivers. In addition, several algorithms in the data processing chains may also be implemented as sentuators. For example, the depth sensor data from a depth sensor may be noisy and require filtering. The depth filter component may provide a sentuator interface so that all components requiring depth information can access the depth filter's sentuator interface rather the depth sensor's.

The sentuators are accessed via the *Sentuator* interface obtained via the configuration service. Software components providing sentuator services implement the *SentuatorService* interface. These services are registered with a *SentuatorServer*, which in turn processes the RPC requests from client components. Sentuator services are identified using predefined constants for measurement type or actuator type. Measurements obtained from sentuators may contain multiple quantities identified using predefined measurement quantities constants. The API is shown in Figure 10.

2) *Sentuator Notification*

The sentuator service API provides for a notification option. This enables components to be registered (via the configuration database) as listeners for specific measurement notifications. When data for these measurements becomes available, the sentuator driver calls the notify() method to deliver unsolicited unreliable notifications to all listeners. This construct is especially useful for components such as navigation computers which require a continuous stream of data from navigation sensors. If not for the notification construct, these components would have to resort to inefficient polling.

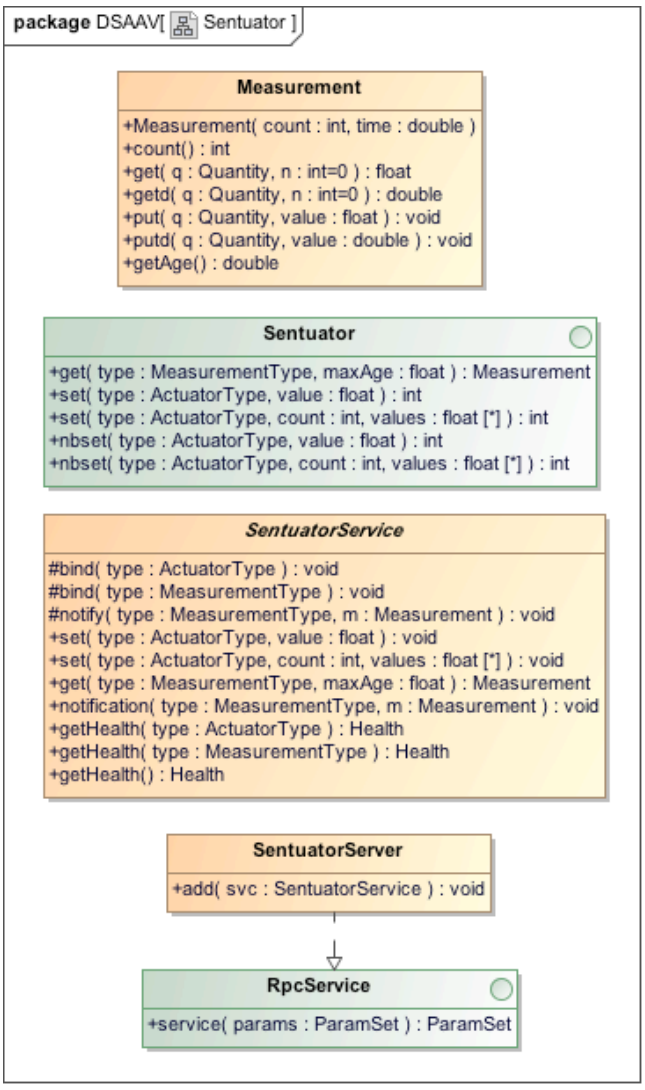

Figure 10. The sentuator API

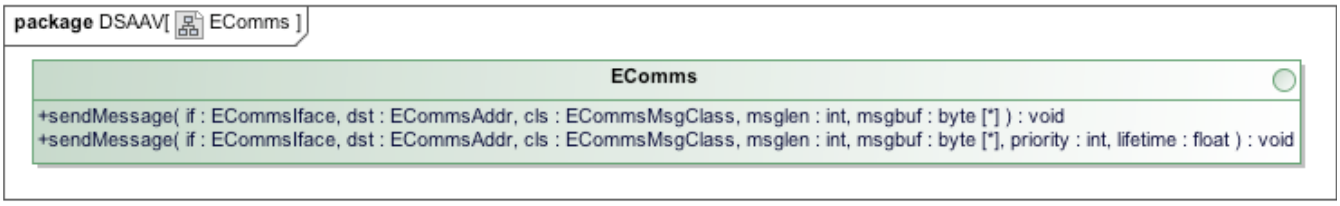

Figure 11. The EComms API

3) *Sentuator Service Discovery*

The sentuator server handles get, set and notification RPC requests. In addition, it also supports health check RPC requests from the health monitor. These response to a health check request contains a list of sentuator services offered and their health information. This feature can be used to discover services available on the server. In STARFISH, we use this feature to discover services automatically as display them on the administration GUI. This could be extended in the future to develop a graphical configuration tool to connect together the software components in the vehicle and automatically generate the configuration file for a mission.

F. *EComms Interface*

The external communications interface provides an API to communicate with the external world - other autonomous vehicles and manned control centers. The EComms API defines the constructs for such communication, but it does not define the external communication schemes or data formats. Emerging standards such as CCL complement DSAAV by providing a specification for the external messages [2].

The EComms API is shown in Figure 11. It consists of a single method to send an external message. The parameters specify the interface over which the message is to be sent (e.g. acoustic communications, WiFi, etc.), the destination address, the message class (e.g. CCL) and the message content. Optional parameters include the priority and lifetime of the message; these parameters help the network stack determine how the messages are queued and processed. Incoming messages are delivered using a RPC_ECOMMS_RECEIVE message on the RpcService interface. Listeners are registered via the configuration database in a similar manner as the sentuator notification listeners.

G. *Components & Containers*

To promote portability of the software components developed using the DSAAV architecture, we provide a component-container construct. Every DSAAV software component extends a *Component* class (see Figure 12). All software components run in a container - each container contains one or more components. Containers are deployed on various hardware nodes in the vehicle. Micro-controllers with no multi-tasking or multi-threading support run a single container. Single-board computers with multi-tasking operating systems can run multiple containers as separate threads or processes.

The component-container construct allows single-threaded systems to run multiple components. It also enables seamless portability of components across hardware nodes and containers provided the components have no hardware dependency (such as access to serial ports or general-purpose input-output pins). The container supports multiple components by implementing a simple form of cooperative (non-preemptive) multi-tasking. Each component in a container receives RPC requests for the component. In addition, it also receives timer ticks at regular intervals for data processing and housekeeping tasks.

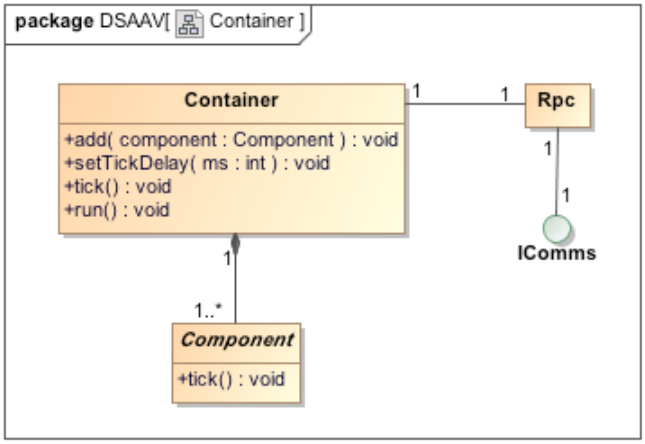

Figure 12. The component & container API

III. IMPLEMENTATION

So far in this paper we have presented the DSAAV architecture with some comments on how its implemented in the STARFISH AUV. In this section we further expand on the implementation by giving a system-level overview of the STARFISH AUV. We hope to illustrate practical use of the architecture through this example.

A. *Brief Overview of STARFISH*

The STARFISH research program was started in 2006 and aims to develop a research platform for exploring cooperation within small teams of AUVs. The base AUV developed as part of this program is modular at the mechanical, electronics and software level. The modularity and open interfaces in the AUV ensure that different research working groups can easily develop and test their ideas simply by replacing software components, and in some cases adding/changing hardware

sections to the AUV. The DSAAV architecture adopted in the project helps achieve this goal.

The base STARFISH AUV is about 1.5 m long and 0.2 m in diameter. It consists of 3 sections - the nose, the tail and a command, control & communications (C3) section. The nose section has a flooded nose cone with a depth sensor, an altimeter, a forward looking obstacle avoidance sonar and an emergency ballast drop mechanism. The tail section houses 4 servo motors which control 4 independent control fins. It also has a DC thruster for forward propulsion. The C3 section incorporates a compass, a low-cost inertial measurement unit (IMU) and a GPS receiver. In addition it contains the communications interfaces including an acoustic modem and a WiFi bridge.

Additional payload sections can be added to the AUV. For increased navigational accuracy, we have a Doppler velocity log (DVL) section that is typically attached between the nose and the C3 section. A side-scan sonar section is currently under development. Other sections containing environmental & chemical sensors, bow & heave thrusters for hovering, etc. may be added in the future. A photograph of the STARFISH AUV during one of the early field trials is shown in Figure 13.

Figure 13. STARFISH AUV

A common bus connects all the sections in the STARFISH AUV. This bus primarily supplies power and provides an Ethernet communications backbone to all sections. Each section has one or more micro-controller units (MCU) or single-board computers in order to interface with the sensors/ actuators in the section. The nose, tail and DVL sections use a STR912 MCU for this purpose. The C3 section contains one STR912 MCU and one PC104+ format single-board computer running Linux.

B. DSAAV implementation in STARFISH

The DSAAV implementation in STARFISH is based on an IComms implementation using raw Ethernet packets. The code is written in C++ and compiled for Linux as well as the STR912 micro-controller using appropriate GNU tool-chains. By restricting the code to use a subset of C++ without template libraries and C++ standard library, the code is easily portable to most platforms with a C or embedded C++ compiler. A Java interface to the RPC layer allows the control center GUI (developed in Java) to make RPC calls. A UDP bridge software component runs on the Linux node as a gateway between RPC over UDP/IP (through the WiFi connection) and RPC over Ethernet, allowing the control center to access RPC services on the AUV for real-time monitoring and diagnostics.

A *core server* consisting of the configuration service, logging service, health monitor and the UDP bridge runs on the Linux node in the C3 section. The node also runs the EComms server. This node is connected via the WiFi bridge to the a control center for administration. The mission file and configuration file is uploaded to this Linux node at the start of each mission. Log files may be downloaded at the end of the mission.

The sensors and actuators in each section are connected to the micro-controller in that section via an appropriate interface (RS232, I2C, SPI, etc.) The micro-controller runs a single container with all the appropriate sentuator drivers, thus providing a common software interface for access by other components in the AUV. In this way, each section is self-contained in terms of hardware and the software drivers associated with it.

Although each section contains the software drivers needed for the sensors/actuators in the section, it is more convenient to manage the settings of all sections centrally. This is done via the common configuration file on the Linux node. All sections also log their data and other messages at the central logging server, making troubleshooting and data analysis easy.

The Linux node in the C3 section runs a command & control (C2) system, a navigation & positioning system, a depth/bearing/roll control system and a AUV safety monitor. All these systems use the sensor/actuator services provided to control high level operation of the AUV.

The high-level software architecture of the STARFISH AUV is shown in Figure 14.

C. Simulation Environment

STARFISH development was greatly aided by a simulation environment which allowed the researchers to test their algorithms and implementations prior to field trials. The simulator implements a physics-based model of the environment, providing RPC services equivalent to all sentuators in the AUV. As the simulated sentuator services are

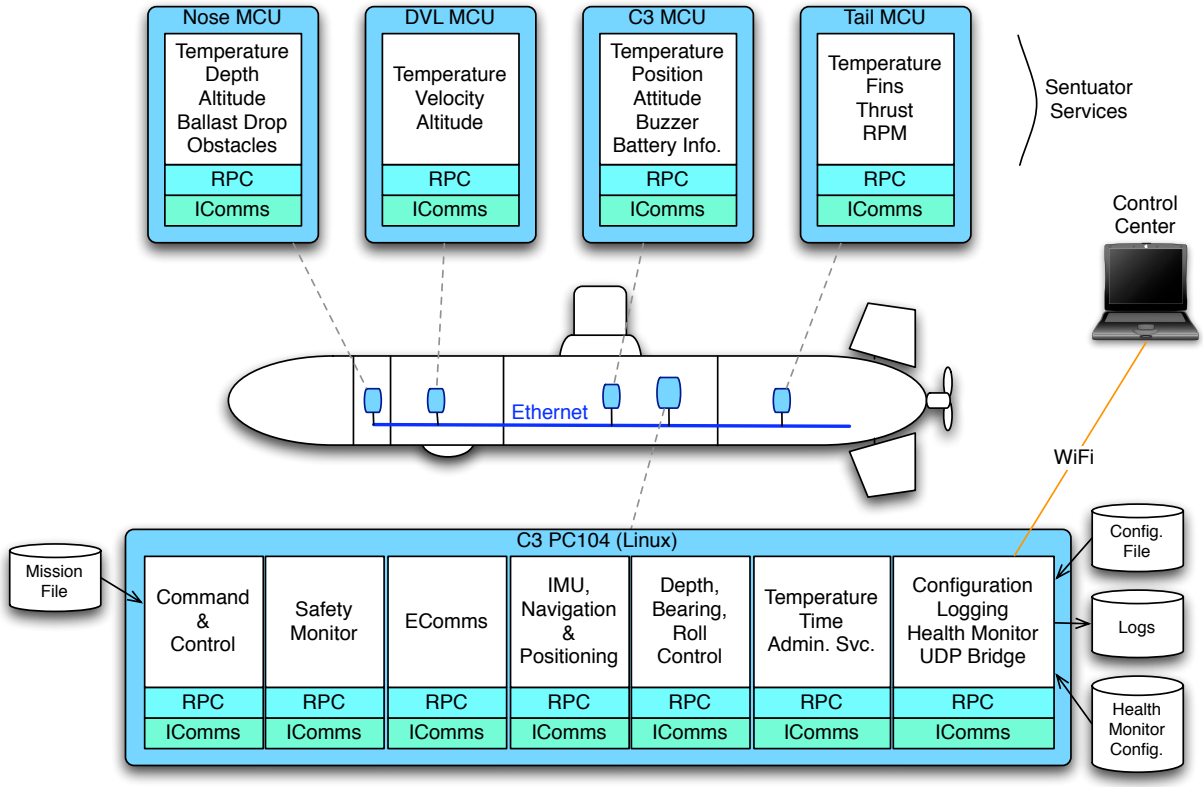

Figure 14. STARFISH software component deployment in a typical AUV configuration

available to all AUV software components, the components do not require any changes between the simulation environment and the actual AUV. A POSIX message queue based IComms implementation is used in the simulator so that all components can be tested on a single computer with no network connection.

D. Benefits from the DSAAV Architecture

The DSAAV architecture has benefited STARFISH immensely. The use of RPC has allowed distributed deployment of software components in the AUV. Moreover, re-deployment of components has been extremely easy. For example, we initially used two PC104+ single board computers in the AUV. The IMU and compass were connected to one of these PC104s. However, due to space, power and heat dissipation considerations, we eventually removed one of the PC104s. The IMU and its driver were moved to the other PC104 with no changes. The compass and its driver were moved to the C3 MCU with very minor changes.

Data plumbing through the configuration file is extremely easy. During actual AUV operation, we expect the command & control system to use the position estimate by the navigation & positioning system. However, during initial testing, we wanted to decouple the testing of the two systems. As the initial tests were conducted on/near the surface, we had position estimates from GPS available. A small change in the configuration file allowed us to direct the command & control system to use the position service on the C3 MCU (where the GPS is connected) rather than the position service from the navigation & positioning system. Once both systems were independently validated, the configuration file was reverted back to resume normal operation. Another example of the use of data plumbing is the optional DVL section. If the DVL is present, the velocity estimates from the DVL are used through out the AUV. If it is absent, the configuration file directs all components who require velocity estimates to the navigation & positioning system which estimates velocity based on thrust. As the altitude data from the DVL is more accurate than the altimeter, all components needing altitude information are directed to use the DVL altitude if the DVL section is attached. All of these changes in data sources can be implemented without any change to the actual software components.

A final example illustrates the flexibility offered by the RPC construct. The MCUs are usually programmed through a JTAG port. Re-programming the MCUs therefore requires physical access to the MCU board inside each AUV section. To avoid having to open up the AUV for each software change, we wanted a software download facility that would allow us to reprogram the MCU over Ethernet. This was easily implemented using the RPC layer as follows. An update server offering a RPC service for software download was

implemented at the control center. Each MCU was programmed with a boot-strap code which made RPC calls to check for new updates and optionally downloaded the updates to the MCU using RPC. Although the DSAAV architecture did not originally plan for such usage, the flexibility and extensibility provided by RPC made this easily possible.

IV. Conclusions

In this paper, we have presented a distributed software architecture for use in autonomous vehicles. The architecture was implemented and tested on a research AUV and found to provide many practical benefits during development and operation. Being distributed, the architecture facilitates and encourages modularity at a hardware and software level. It also avoids single points of failure and load bottlenecks. The distributed architecture can easily be extended for use across a team of AUVs.

In summary, DSAAV provides the following key benefits and is therefore well suited for use in autonomous vehicles:

- Distributed architecture with support for hardware and software modularity.
- Robustness and load distribution through peer-to-peer communications.
- Independence of communications back-bone available.
- Lightweight protocol for high speed implementation on low-power micro-controllers.
- The RPC construct along with the component-container architecture makes the distribution of components across multiple nodes almost transparent to software developers.
- Basic services such as configuration, logging and health monitoring are defined and integrated as part of the basic architecture.
- Automatic discovery of sensor/actuator services through the health monitoring API.
- Easy "plumbing" of data flow between software components via configuration files.
- Flexibility and extensibility provided through access to the underlying messaging API and RPC API.

The author hopes that this architecture is adopted and extended by others working on autonomous vehicles. The author is happy to make detailed documentation and relevant implementation available to interested researchers.

Acknowledgement

The author would like to thank Mr. Shiraz Shahabudeen for his comments and suggestions on the material presented in this paper.

References

[1] Newman P.M., "MOOS - Mission Orientated Operating Suite," available at: http://www.robots.ox.ac.uk/~pnewman/TheMOOS/index.html. Accessed 16 July 2008.

[2] Stokey R.P., L.E. Freitag and M.D. Grund, "A Compact Control Language for AUV acoustic communication," OCEANS 2005 - Europe, Brest, France, 2005.

[3] Deshpande P.D., M.N. Sangekar, B. Kalyan, M.A. Chitre, S. Shahabudeen, V. Pallayil and T.B. Koay, "Design and Development of AUVs for cooperative missions," Defence Technology Asia 2007, Singapore, 2007.

[4] OMG, "Unified Modeling Language," available at: http://www.uml.org/. Accessed 16 July 2008.