An Object Oriented Approach to AUV Software Development

Thomas C. O'Reilly
Software engineer
Monterey Bay Aquarium Research Institute
7700 Sandholdt Road, Moss Landing, CA 95039-9644
USA

Introduction

The Monterey Bay Aquarium Research Institute (MBARI) is currently developing a new generation of AUV. Called *Dorado,* this vehicle has many unique features, including "plug-and-play" sensor configuration, an articulated tail-cone with independent microprocessor, and capability to carry a fuel cell for long-duration multi-day missions. The vehicle design emphasizes modularity, from both a hardware and software standpoint.

In this paper, we describe the design of the *Dorado* main vehicle computer software. We do not describe specific components such as motor control or navigation algorithms, nor do we discuss the independent tailcone microprocessor software. Rather we focus on the architectural framework that glues the system together. The

Figure 1: The *Dorado* AUV. Note the ring-ducted tailcone.

system architectural design has direct bearing on the system's ease of implementation, maintainability, autonomy, and extensibility, and hence impacts the engineering, performance, and economic costs of the vehicle as well.

Software requirements and goals

Requirements and goals for the *Dorado* vehicle software include the following:

1. Autonomy. Mission duration for the vehicle can range from minutes to many days. During this time the software must function reliably without human intervention.

2. Real-time performance. Certain vehicle functions must execute at a specific rate, with little or no tolerance for missed execution. This is termed a *hard real-time requirement*; the navigation and dynamic control functions are in this category.

3. Maintainability and extensibility. The design must be understandable to those who will implement, debug, and extend the system framework and applications. It must be possible to change and improve the implementation of system components, with minimal or no "ripple effects" throughout the rest of the system. It should be straightforward to implement new subsystems, such as device drivers. The system should have well-defined interfaces to external, off-board components, such as graphical user interfaces and MBARI ocean observing system components.

4. Scalability. The system must accommodate interaction with multiple sensors and devices on the vehicle. Ideally the system design should incorporate a "distributed computing" architecture, should additional processors be added to the vehicle.

5. Configurability. The suite of payload sensors and devices will vary radically from one mission to the next, as may the vehicle length, shape, and even propulsor hardware. These variations will naturally affect dynamic control as well as data logging.

6. Portability. At some future time, we might desire to port the system to a different hardware or operating system (OS) platform. Software layers which directly depend on OS or hardware should be limited and "hidden" from the rest of the system.

7. Reusability in other applications. Where feasible, components should be designed such that they could be used in other projects, either as infrastructure or utilities.

Hardware and operating system

Dorado's main vehicle computer is a 486 processor packaged in a PC/104 form factor, running at a clock speed of 133 MHz. The system currently includes a nine Gigabyte hard disk and 32 Megabytes of RAM. The main vehicle computer runs the real-time QNX operating system from QSSL Ltd. QNX is a Unix-like operating system, is highly configurable and scalable, and is nominally compliant with the IEEE POSIX standard. POSIX is a UNIX-like operating system interface specification, which has been at least nominally implemented by many operating systems, including QNX, VxWorks, and Linux [8].

Framework vs application

Software components of a well-designed system can be categorized as either *framework* or *application* code. *Framework* code forms the infrastructure of the

system, and typically includes software for functions such as interprocess communication and time-keeping. *Applications*, such as a vehicle dynamic control task, are built using framework components. The framework code is never modified or even seen by applications programmers; the framework only exposes well-defined *public interfaces* to the application code. This segregation into framework and applications helps to ensure portable software, if all OS-specific function calls (i.e., implementation details) are confined to the framework. Then if the system is to be ported to a new OS, only the framework implementation needs to be modified; the applications are unaffected.

In our system, we have been careful to confine all QNX-specific system calls to the framework. Our rule (admittedly not enforceable by a compiler) is that any OS functions called by application code must be defined in the POSIX specification. Thus, our system will hopefully be portable to any POSIX-compliant operating system.

Object oriented design

When developing a new software architecture, designers are faced with two major choices; traditional *top-down procedural design*, or the newer *object oriented* (OO) approach. OO design has been widely accepted over the past decade as being superior to the procedural approach in modeling and designing complex software systems [4,6,7]. This acceptance is reflected in the current and growing popularity of OO languages such as C++ and Java. The many benefits of OO design have been extensively documented [3]. We have found through experience that the software requirements listed above are best addressed by the OO approach.

However, OO design and implementation has been most commonly applied to non-real-time applications, such as graphical user interfaces and databases. The very different domain of real-time and embedded applications (such as vehicle control systems) is still dominated by the procedural approach; most of these systems are implemented in 'C' and assembly language. . In addition to having hard real-time requirements, these applications typically reside on platforms which are limited in system resources such as memory and disk space. There is a common perception that OO design and implementation incurs excessive code-size and performance overhead. However, this picture is beginning to change. Recently, there has been much interest in applying OO design to real-time and embedded applications [4]. As pointed out by Herity [5], the actual overhead of C++ relative to C is usually minimal or nonexistent, so long as well-defined commonsense design and coding rules are followed. There are now descriptions in the literature of actual OO real-time, safety-critical applications, ranging from household appliances to avionics systems [9].

Single- vs Multi-tasking

The simplest applications need only a single thread of control. However, like most robotic vehicles, *Dorado* includes numerous sensors and devices, running at different frequencies and having different response requirements. For example, navigation sensors and vehicle dynamic control functions have a hard real-time requirement, and are critical to vehicle safety. A bioluminescence sensor or mapping sonar would usually not have such stringent requirements. Moreover, the mix of critical and non-critical devices is likely to vary considerably from one payload to the next. In this diverse environment, a single-threaded system is unable to guarantee hard real-time

response. If any single function in the task blocks for longer than expected, the timing of the control loop will be affected, introducing "jitter" and degradation of control. If a function blocks indefinitely (e.g., while reading a serial port), the control system will come to a halt, with potentially disastrous consequences.

A solution is to assign functions to different tasks, using the multi-tasking capabilities of the operating system. For example, the dynamic control loop runs in a single task (or set of synchronized tasks) at high priority, while non-critical functions (such as a bioluminescence driver) run in lower priority tasks. The control system task continues to run even if other tasks block or even terminate. (In a single processor system such as ours, multiple tasks are actually run in sequence, not in parallel; the OS multitasking features provide the illusion of multiple "virtual" processors.)

A multi-tasking architecture does pose technical challenges. Priorities and scheduling algorithms of the tasks must be adjusted "just so", to achieve desired timing and execution order. Fortunately, operating systems such as QNX provide tools to aid in this process. Another complexity is the debugging of interactions between multiple tasks; this can be difficult compared to a single-threaded system. We have addressed this problem by developing a framework of C++ classes to handle interactions between tasks. This framework, upon which all of our applications are built, has been well-tested and debugged. We describe this framework, termed *distributed objects*, in more detail below.

Dorado is in many ways an evolutionary progression from the *Odyssey* vehicles developed at MIT and described by Bellingham et al [1,2] The *Odyssey* vehicle software was designed using traditional top-down procedural methods, is implemented in 'C', and has a single thread of control. The *Dorado* design is both object oriented and multi-tasking, and so is a radical departure from the *Odyssey* architecture. The decision to "start from scratch" rather than build on an existing design is sometimes a difficult one. In our case, the decision to redesign the system was taken only after careful consideration of the pros and cons of each option, in light of the software requirements. In addition, our previous experience with object oriented methods and multi-tasking, in the context of MBARI's ROV *Tiburon*, convinced us that the benefits of the newer approach were very real indeed. (Although the *Dorado* architecture is entirely new, we have adapted several of *Odyssey's* algorithms for specific components, as discussed below.)

Distributed Objects

The *distributed object* approach combines object orientation with multi-tasking. A distributed object provides an interface between two tasks. The task which calls a method on a distributed object is referred to as the *client*, while the task which actually executes the method and returns output to the client is the *server*. The distributed object created by the client in its address space is referred to as a *server proxy*. There must be exactly one server instance corresponding to each proxy instance, but many proxies can connect to a single server.

The implementation part of the object resides and executes on the server machine. The server proxy hides all of the deta
Clients utilize a service through the service interface, whereas the actual service is performed on the server machine.

extends the notion of software objects across a network. Distributed objects are based on the client-server model. A *server object* is partitioned into interface and implementation. The server's interface (also called a *server proxy*) can be instantiated by clients on the network. A client can then call methods on the server proxy object.

2. On the server side, an application developer provides a subclass of the appropriate "skeleton" class, and implements the distributed object's interface methods. The proxy, server, and skeleton base classes convert the method invocation and return values into messages, which are passed between the client and server tasks. Since the interprocess communication mechanism, which may rely on OS-specific system calls, is hidden from application code, distributed object applications are inherently quite portable between operating systems. Porting to a new OS only requires that the server and proxy base classes be modified; the application code is insulated from any changes.

```
NavigationProxy *navigation;
double heading;
time_t sampleTime;

try {
  // Instantiate the Navigation server proxy object
  navigation = new NavigationProxy("navigation");

  // Get current vehicle heading
  heading = navigation->heading(&sampleTime);

  // Done with the proxy; delete it
  delete navigation;
 }
catch (Exception e) {
  // Process exception here
}
```

Figure 2: Example of C++ client application code which retrieves vehicle heading from the Navigation server through a proxy object.

Another advantage of distributed objects is that they are *location transparent*; the client application does not "know" where the server task physically lives. Rather, the client provides an abstract *key* (e.g., a character string) to the proxy's constructor; the details of how the proxy translates this to a server address are encapsulated, hidden from the client. Thus the configuration of the entire system is very flexible; a server may live on the same processor as the client, or it may reside anywhere on a network. In a multiprocessor system, the processing load can be balanced by distributing servers appropriately. Thus in addition to being portable, distributed object systems are also highly configurable.

The distributed object model is a very powerful one, and has been implemented by industry variously as CORBA, Java RMI, and Microsoft's DCOM. None of these "off the shelf" systems are yet reliably supported by the vehicle's QNX operating system. Therefore we have developed our own framework of base classes and tools which implement a limited distributed object system for QNX. In our current implementation, clients and servers must reside on a single processor (our vehicle currently has just one main cpu anyway). However, the framework could be readily extended to multiple processors, without affecting applications in any way.

Our implementation also includes an event notification mechanism. Clients can subscribe to events defined by a given server, by calling the proxy's *subscribe()* method. The client then blocks by calling the proxy's *waitForEvent()* method, until the server determines that the specified event has occurred and sends notification to all subscribers. Thus a task's execution can be event-driven, and thereby synchronized with other tasks. (A client can also block simultaneously on multiple proxies, using our *EventService* class.)

Under the hood

The connection between the client and server tasks is established by the proxy when it is instantiated by the client (the proxy determines the server's process id via a QNX "lookup-pid-by-name" function). When the client invokes a method in the distributed object's interface, much happens "under the hood". Figure 3 is a sequence diagram which depicts interactions between client and server tasks during invocation of a method. Before the method is called, the server task is in a blocked state, waiting for

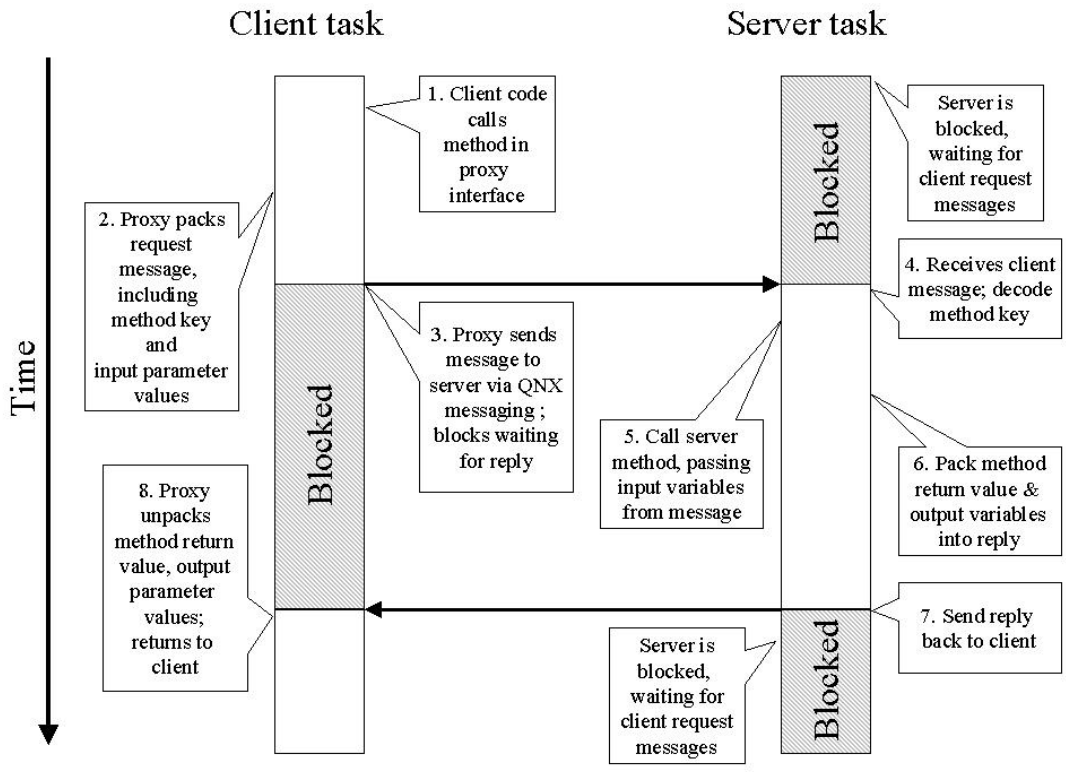

Figure 3: Sequence diagram depicting interaction between client and server tasks during a method call on a distributed object. Time flows from top to bottom of the diagram.

client requests. When the client calls the proxy method, the proxy copies the method's input parameters to a message buffer. The proxy also places a *method key* into the message; the method key identifies to the server which method is being invoked. The proxy then sends the message to the server via the QNX messaging system, and the client task blocks waiting for a reply. The server receives the message, reads the method key, and invokes the appropriate server method, passing the input parameter

values contained in the message. After return from the method, the server copies the return value and output parameter values into a reply buffer, and sends the reply back to the waiting proxy in the blocked client process. The proxy reads the reply, extracts the method return value and output parameter values, and places them on the calling stack in order to make them accessible to the calling application.

These implementation details are of course hidden from the application itself, but there is nonetheless quite a bit going on here. Exactly *what* goes on depends to a large extent on the data types of the method being called. It might be possible to devise software that would determine the types at run time, but it is far more straightforward to determine the types at compile time, and to write code corresponding to each type of distributed object. On the other hand, while it might be straightforward to write all those classes, it would also be extremely tedious (and hence boring and error-prone) to do so. Fortunately, there is a better solution. The designers of CORBA have solved this problem by inventing the *interface definition language* (IDL).

Interface Definition Language and object implementation

IDL defines only the interface (i.e., method signatures) of a distributed object; it does not define the object's implementation. IDL syntax closely resembles C++, and an IDL interface definition looks much like a C++ header file. IDL supports interface inheritance and function overloading, structures, enumerations, and most of the primitive data types found in C++. IDL does not support procedural definitions of any kind.

To develop a new type of distributed object, the programmer first describes the object's interface in IDL. The IDL file is then passed through an IDL compiler, which automatically generates a set of classes in the appropriate language (most often C++ or Java). These classes define the proxy and server sides of the distributed object, and handle all of the implementation details corresponding to those in figure 3. The only work left for the programmer is to implement the server methods (which may be a very simple task or very complex, depending on the application).

The "magic" in all of this is the IDL compiler. But the IDL language specification does not come with a compiler, as the implementation details may be different for different operating systems. The compiler is always included with a CORBA *Object Request Broker* (ORB) that actually implements the system.

We have adopted the IDL approach described above. Since we do not have an ORB for QNX, we have written our own IDL compiler, using *lex* and *yacc* [10]. The compiler generates classes which implement the steps illustrated in figure 3.

Let's look at a concrete example. Figure 4 shows the IDL for a distributed object called *TailCone*. *TailCone's* responsibility is to accept control surface and propeller commands from a client (e.g., *DynamicControl*), and set the hardware to the desired values. Our example interface has a single method, called *commandl* (The example shown is simplified; the actual *TailCone* has several additional methods.)

```
/* Simplified TailCone example */
interface TailCone {

  // Set hardware to specified values
  int command(in double propOmega,
              in double elevatorAngle,
              in double rudderAngle); }
```

Figure 4: IDL for the TailCone distributed object.

Note that IDL specifies either "in", "out", or "inout" for method arguments. When the file TailCone.idl is input to our IDL compiler, several C++ source files are generated. Figure 5 shows the hierarchy of the generated classes. The *TailConeProxy* class is descended from our framework's *ServerProxy* class. *TailConeProxy* is a concrete class, has a *command()* method, and can be directly instantiated by a client that wishes to control the vehicle's tailcone. *TailConeSkeleton* is descended from the framework's *Server* class. *TailConeSkeleton* is abstract and cannot be directly instantiated; it contains the pure virtual *command()* method. The programmer who is implementing the *TailCone* distributed object must subclass *TailConeSkeleton* into *TailConeServer*; this subclass must implement the pure virtual *commandl()* method. Thus *TailConeServer* would actually communicate with the tailcone hardware.

In summary, implementation of a new type of distributed object requires the following steps:

1. Define the object's interface in IDL.
2. Compile the IDL file; this generates the proxy class and server skeleton class
3. Subclass the skeleton; the subclass must implement the methods defined by the IDL.
4. Write a program that instantiates a server object and invokes its run() method (inherited from the Server base class) . The server is now waiting for client requests.
5. Clients can now instantiate a proxy object of the appropriate type, and invoke its methods.

Note that steps 1 through 4 need only be carried out by the person who is *implementing* a new distributed object. Application programmers who wish to use the object need only complete step 5.

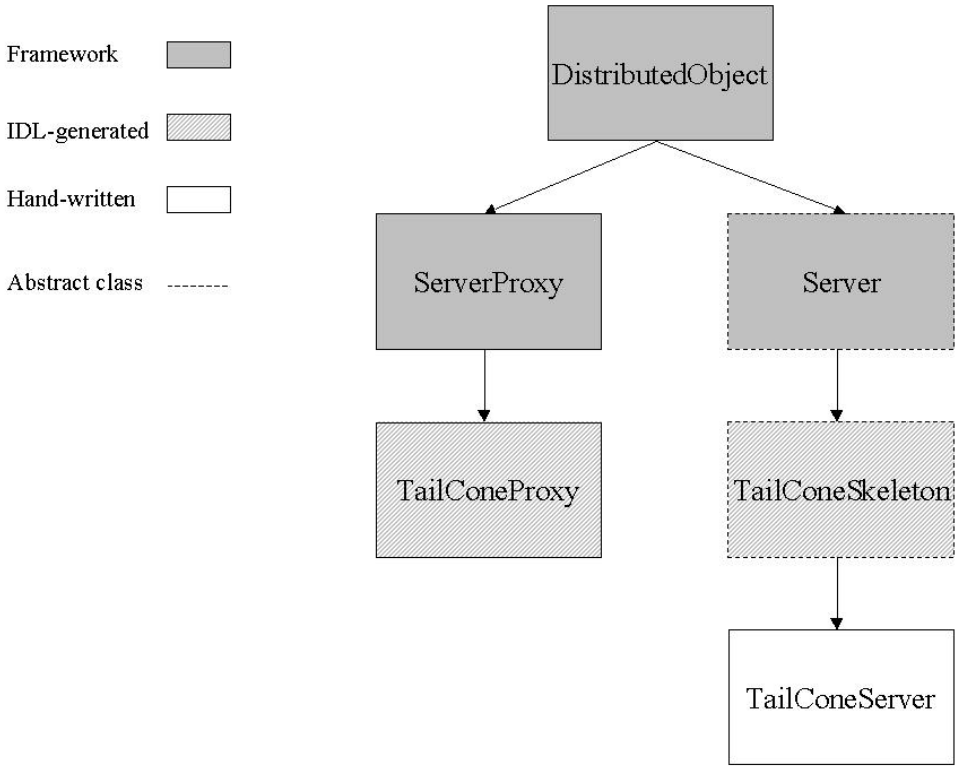

Figure 5: Hierarchy of TailCone distributed object classes

Object configuration

Figure 6 shows an *object message diagram* for the "core" distributed objects in the vehicle system. Each box in the diagram represents a distributed object instance. The arrows indicate method calls; the arrow originates at the client and terminates on the object whose method is being invoked (i.e. this is not a classical data flow diagram). A nuance not discussed earlier is that the implementation of a distributed object may itself be multi-tasking. For example, the *Navigation* implementation consists of a server task, and another task that periodically samples the navigation sensor objects. The periodic task supplies navigation data to the server via shared memory. On each cycle, the periodic navigation task also fires an event, which is propagated through the *Navigation* server to any subscribers. In our current vehicle configuration, the *DynamicControl* object subscribes to this event. On event notification *DynamicControl* calculates settings for the *TailCone* object, based on the latest input from the *MissionPlanner* and current vehicle state from *Navigation*. The effect is that the execution of the *DynamicControl* and *TailCone* tasks are synchronized to the periodic navigation task (which runs at 5 Hz). The configuration shown in figure 6 is just one of many possibilities. The distributed object model, including the event notification mechanism, makes it straightforward to experiment with various schemes.

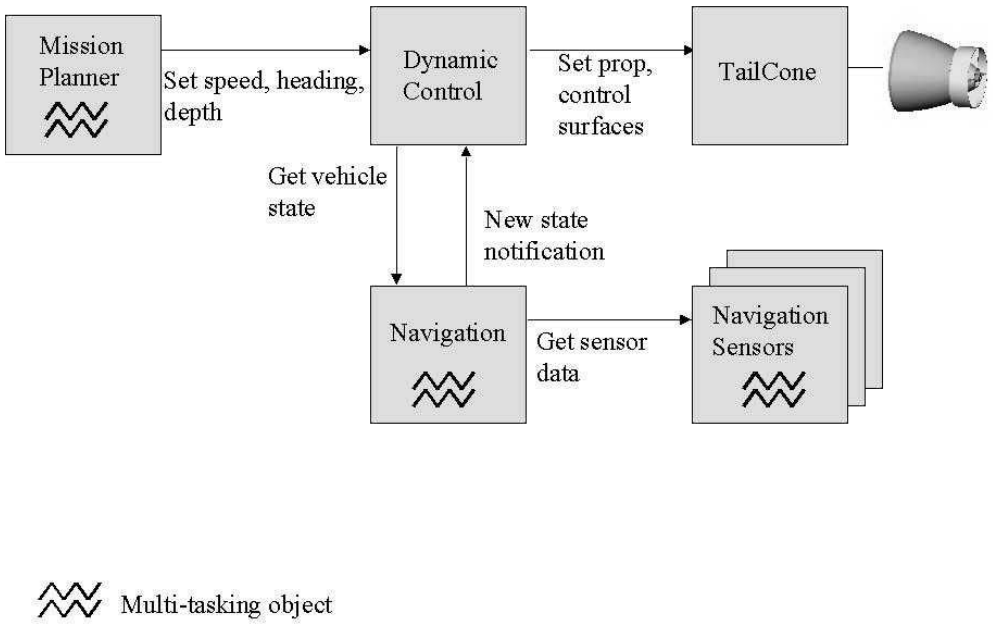

Figure 6: Distributed object message diagram for the *Dorado* vehicle

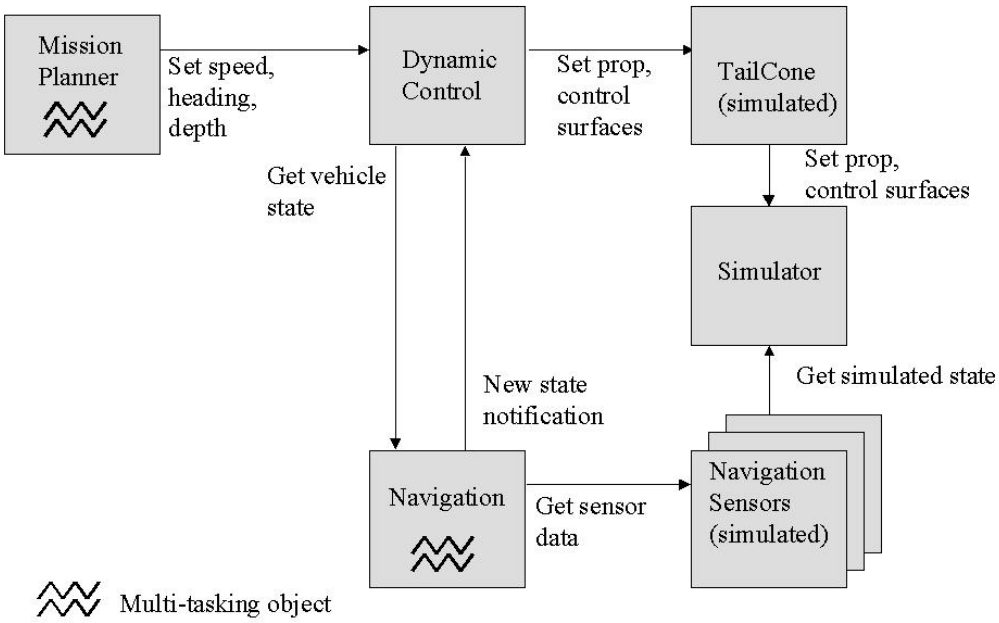

Figure 7: Distributed object message diagram for *Dorado* in "simulated" mode

Implementation, test and validation

Standalone testing of object oriented software is generally straightforward, given concise interface definitions. Small "software harness" test programs were written and used to test the various framework components on an individual basis.

For early system testing, we developed a *Simulator* distributed object. The *Simulator* was implemented by porting *Odyssey's* 'C' simulation code. Given vehicle hydrodynamics, control parameters and control inputs, *Simulator* computes vehicle state as a function of time. For the *TailCone* and sensor objects we implemented servers that communicate with the *Simulator* instead of hardware devices. The *TailCone* and sensor distributed interfaces did not change, however, so *Navigation* and *DynamicControl* are completely "unaware" that they are communicating with a simulation rather than the real world. The object configuration in "simulated" mode is shown in figure 7. The end result is that we are able to test the actual "flight code" for *Navigation*, *MissionPlanner*, and *DynamicControl* in a simulated environment.

Implementation of the design was impressively rapid. In fact, software development outpaced vehicle hardware development. It soon became apparent that the software would be ready for sea-trials months before the *Dorado* tailcone hardware was ready. To prevent loss of momentum in the software effort, it was decided to use an MIT *Odyssey* vehicle as a software test-bed. An *Odyssey* was procured and given a "brain transplant"; the vehicle's VME computer was replaced with a *Dorado* PC-104 stack running QNX. This hybrid vehicle was christened *Dorodyssey-1*. Navigation and science sensors similar to those planned for *Dorado* were installed. Finally, the *Dorado* software was installed on the main vehicle computer. Virtually all of the distributed objects shown in figure 6 are interchangeable between *Dorodyssey* and *Dorado*. The only object whose implementation differs is the *TailCone* server, as the *Odyssey* hardware is quite different from the new MBARI tailcone. But even this implementation was straightforward. A new class, *OdysseyTailCone*, was subclassed from the IDL-generated *TailConeSkeleton* class. Existing *Odyssey* tailcone 'C' code from MIT was then ported into the new class. Note that only the *TailCone implementation* differs, but not the distributed object interface. Thus the other components in figure 6 required no modification whatsoever.

In January 2000, *Dorodyssey-1* underwent extensive sea-trials in Monterey Bay. During these tests, the new system architecture was validated, as were the individual vehicle software components and subsystems. During five days of intensive testing, no major software "bugs" were discovered.

In addition to being an invaluable test-bed, the *Dorodyssey* vehicle has proved to be extremely useful as a science platform as well. As of this writing, several more *Odysseys* have been converted to the *Dorodyssey* configuration by MIT's AUV Lab. In August 2000, two *Dorodysseys* are participating in a two-week study of upwelling phenomena in Monterey Bay.

Conclusions

We have found that our design and resulting software that meet all of the requirements described earlier. Thus far, the software has proven to be quite robust, both in the lab and at sea. We are able to verify that hard real-time control requirements are being met through use of tools supplied by QNX and developed by

us. The distributed object approach has been particularly successful in providing a system that is maintainable, extensible, reconfigurable, scalable, and portable. For example, the adaptation of the *Dorado* software to a *Dorodyssey* configuration was remarkably straightforward.

A crucial measure of a design's success is the ease (or difficulty) of developing new applications within the design framework. Our system is not a procedural/OO "hybrid", but is object-oriented from the ground up. A diverse and geographically distributed group of people are currently developing software for the vehicle, including developers from MBARI, MIT, and Bluefin Robotics Corporation of Cambridge Massachusetts. The object-oriented experience level of these developers ranges from "very little" to "very experienced". We have learned what is perhaps already obvious; while experienced OO developers learn the system fairly quickly, there is a steep learning curve for those who are new to C++. Accurate documentation of the system's C++ classes is extremely useful, for both experienced and novice C++ developers. Maintaining accurate documentation is itself a challenge, and we are currently evaluating several ways of automating this process.

Finally we note that the AUV program is but one component of an ocean observing system currently under development at MBARI. Hopefully our *Dorado* design experience - and perhaps actual *Dorado* software components - can be applied to other vehicles and data acquisition systems.

Acknowledgements

Andrew Pearce, formerly of MBARI and now at Caw Networks, was a major contributor to our design. I'd like to thank Jim Bellingham, Bill Kirkwood, and Duane Edgington for their valuable advice, guidance, and support. The following people actually develop in the architectural framework or run the vehicles, and I'm grateful to them for their feedback and suggestions; Doug Au, Rob McEwen, Drew Gashler, and Nicole Suoja of MBARI, John Rieffel of MIT, and Aaron Marsh and Don Green of Bluefin Robotics Inc. Many thanks to Berta Pires of Vicinity Corp for proofing this paper.

References

[1] Bellingham, J.G., Humphrey, D. [1990]. Using layered control for supervisory control of underwater vehicles. *Conference Proceedings, ROV '90,* Marine Technology Society, pp 175-181.

[2] Bellingham, J.G., Leonard, J. [1994]. Task configuration with layered control. *Conference Proceedings, IARP Mobile Robots for Subsea Environments*, May 3-6, Monterey, CA

[3] Booch, Grady [1991]. *Object Oriented Design With Applications*, Benjamin/Cummings Publishing Co, Redwood City, CA, ISBN 0-8053-0091-0

[4] Douglas, Bruce P. [1998]. *Real-Time UML*, Addison-Wesley, Reading, MA, ISBN 0-201-32579-9

[5] Herity, Dominic [1998]. C++ in embedded systems: myth and reality. *Embedded Systems Programming*, vol 11, no 2, pp 48-71.

[6] Chagnon, Glenn [1996]. Flexibility by design. *Embedded Systems Programming*, vol 9, no 4,

[7] Eckel, Bruce [1990]. C++ for embedded systems. *Embedded Systems Programming*, vol 3, no 1

[8] Gallmeister, Bill [1995]. *POSIX.4: Programming for the Real World*. O'Reilly and Associates, Sebastopol, CA, ISBN 1-56592-0740-0

[9] Murphy, Niall [1998]. Introduction to CORBA for embedded systems. *Embedded Systems Programming*, vol 11, no 11

[10] Levine, J, Mason, T., Brown, D. [1992]. *Lex and Yacc, 2nd Edition*. O'Reilly and Associates, Sebastopol, CA, ISBN 1-56592-000-7