Software Infrastructure and Applications for the Monterey Ocean Observing System: Design and Implementation

Thomas C. O'Reilly
Kent L. Headley
Robert A. Herlien
Michael Risi

Daniel Davis
Duane R. Edgington
Kevin Gomes
Timothy Meese

John B. Graybeal
Mark Chaffey

Monterey Bay Aquarium Research Institute
7700 Sandholdt Road
Moss Landing, CA 95039-9644
oreilly@mbari.org

Abstract – MBARI's MOOS moored network is a state-of-the-art ocean observatory for interdisciplinary science, consisting of interconnected "host nodes" on the ocean surface, midwater, and seafloor. Each node can accommodate a wide variety of instruments. We describe some of the challenges posed by the system's functional requirements, and how those challenges are addressed by the design of the system's onboard hardware and software. Integration of diverse instruments and their protocols into the network poses major system integration and maintenance challenges, particularly in the areas of instrument installation and control, data retrieval, and data interpretation. All of these functions must be reliably performed in an often hostile environment on networked at-sea platforms that are generally power limited. We describe a distributed software architecture and implementation that addresses these challenges.

I. THE MBARI OCEAN OBSERVING SYSTEM

Several elements of a moored ocean observatory network have recently been deployed in Monterey Bay, as part of the Monterey Ocean Observing System (MOOS). The MOOS moored observatory is a portable multidisciplinary science platform, with capability to acquire data from a network of instrumented nodes on the sea surface, in the midwater, and on the seafloor [1,5].

In this paper we *describe Software Infrastructure and Applications for MOOS*, or *SIAM*. SIAM software provides autonomous onboard system management, instrument data acquisition and onboard logging, remote instrument control and telemetry to shore, automatic "plug and work" integration of instruments and data streams, and other features. SIAM does not provide long term data archival, processing, and distribution on shore; those functions are provided by MBARI's Shore Side Data System (SSDS) [3].

MOOS hardware and electrical designs provide a very powerful platform for SIAM software, and of course impose restrictions as well. References [1,5] describe the MOOS requirements and design in detail; we summarize some key features below.

A. MOOS network

The moored observatory consists of a network of surface, midwater and seafloor computing platforms or *nodes*. The networked architecture enables MOOS's integrated remote control, data acquisition and telemetry throughout the water column and sea floor. The network connections are provided by a variety of media. Either a satellite modem or terrestrial line-of-sight radio provides the wireless link between surface node and shore. 10 Base-T copper or fiber optic links provide the segments between nodes on the moored network. The primary link between surface and seafloor is the armored electro-optical-mechanical (EOM) cable. The EOM cable distributes power to the seafloor nodes, and also provides 10-Base-T connectivity.

B. MOOS Mooring Controller

Each node is comprised of an instrumented MOOS Mooring Controller (MMC). The MMC is a very capable yet relatively low-power embedded system that represents a refinement of the controller first developed in 2001 [10,5].

Each MMC supports connection and control of twelve instruments with serial communications (RS-232 or RS-422) and 12 VDC power. Several additional communications interfaces are provided on the MMC, including Ethernet, RS-485, and two serial ports used for wireless communications via satellite or RF modem.

SIAM supports a rich set of MMC hardware features, including software controlled galvanic isolation of communications and power circuits, as well as access to hardware diagnostic information, such as voltage, current, temperature and over-current conditions for each of the instrument ports.

In comparison to many embedded controllers, the MMC is resource-rich, hosting embedded Linux, which provides the TCP/IP stack and support for the Java virtual machine that form the software underpinnings for SIAM.

C. Power

The MOOS surface node is equipped with solar arrays, wind generator, and storage batteries that provide at least 20 watts continuous average power year-round to the system when deployed in temperate latitudes. The MOOS power subsystem and EOM cable distribute power from the surface to other nodes in the network.

D. Wireless communications

MOOS provides several wireless communications paths between the surface node and shore. When

0-7803-8669-8/04/$20.00

deployed further than about 50 kilometers offshore, a Globalstar satellite modem can be installed into the primary wireless interface. Globalstar supports a data rate of 7800 bps. Due to the cost and restricted bandwidth of Globalstar, terrestrial line-of-sight RF is preferred as the primary link, mooring-to-shore distances permitting. For this scenario, a 900 MHz Freewave ISM band RF modem that supports 38.4 Kbps speeds is used.

When a Globalstar modem is connected to the primary communications interface, the secondary interface can still be outfitted with a Freewave radio for relatively high-speed ship-to-mooring communications at a range of a few kilometers or less. The secondary link can be extremely useful in support of ship-based maintenance activities, such as retrieval of high-volume data and software upgrades. These activities can be performed even in high sea-states, as there is no need to physically climb onto the mooring and plug in a cable.

Finally, a pager on the surface node, is tied to the MMC reset line, providing a remote node reset capability.

II. SIAM REQUIREMENTS, DESIGN AND IMPLEMENTATION APPROACH

A. Requirements

The system software design is driven by high-level requirements levied by science, engineering, and operational users. Key requirements include the following:

1. Many kinds of instruments must be supported on MOOS. The MOOS science team has already identified more than fifty types of instruments to be integrated into MOOS, including commercial off-the-shelf sensors, as well as "custom" devices developed at MBARI and elsewhere.

2. Deployed instruments must be controllable from shore. With this capability, scientists on shore can interactively modify deployed instrument data acquisition schedules and other parameters such as sensor gains, and engineers can remotely diagnose problems. This capability also helps to enable autonomous event detection and response, described below.

3. The system must be able to *autonomously* detect events of interest and respond to them. The science team believes this feature will enable new kinds of experiments. Some events of interest occur on time scales that are shorter than mooring-to-shore communications latency. For example mass wasting or seismic events may last only minutes or even seconds, requiring a quick at-sea detection and response. Other events may occur on longer time scales – perhaps hours or days – but might occur when humans are not actively examining the data stream for the event (e.g. late at night). An autonomous on-shore component could detect these longer-duration events in the instrument data streams, notify the relevant personnel, and perhaps automatically respond using the remote instrument control capability described earlier.

4. All acquired instrument data must be stored locally on the deployed instrument platform, until such time as they are retrieved to ship or shore. Science data must be reliably associated with metadata that describes the instrument and its state at the time the data was acquired.

5. The process of adding and removing nodes and instruments from the deployed system must be simple, robust, and scalable to moored networks containing tens of instruments. In past systems deployed by MBARI, changing a mooring instrument payload has been a largely manual, time-consuming, and hence error-prone process. The difficulties inherent in this manual process will become even greater with the introduction of moored networks, since each moored network may consist of about five nodes, and each node may host about 10 instruments. Thus each moored network may contain 50 or more instruments at a given time.

6. The system architecture must enable interaction with other kinds of devices and systems, such as autonomous underwater vehicles, other kinds of moorings or cable-to-shore systems,

B. Design approach

Building on the concepts described by [2], SIAM has been designed and implemented as a *distributed object* system, meaning that SIAM software objects reside on multiple MMC nodes and on shore, and interact with one another across the network. Distributed systems are generally more complex to implement than single-processor "monolithic" systems, but offer very powerful benefits in a networked system such as MOOS. Benefits of the distributed approach include scalability, open and well-defined component interfaces, better fault-tolerance, resource sharing, and ability to incorporate heterogeneous software and hardware components [4].

A *service* is a software component that can perform a set of well-defined operations on behalf of other components known as *clients*. The service can be accessed by clients anywhere on the network, through the service's *distributed object proxy*. The distributed object's interface succinctly defines the operations or *methods* that the service can perform on behalf of clients, including the input and output data types. The proxy code executes on the client's machine, but has a communication channel to the service's host machine. When a client invokes a method on a service through the proxy, the client specifies the method's input parameter values; the proxy internally converts the input parameters to a message that is then sent across the network to the service. The service performs the specified operation, and sends the output values as a network message back to its proxy on the client machine, where the values are passed back to the client's address space. The client only deals directly with the distributed object interface, and is isolated from such details as how parameters are converted to messages and how the service actually carries out the requested operation. This greatly reduces the complexity of client code, from whose standpoint access to the service across the network is quite straightforward.

This approach was regarded by the SIAM team as well-suited to the formidable challenges posed by MOOS requirements. In particular, the distributed object approach enables remote control and data retrieval from instruments (requirement #2), as well as autonomous interaction between networked components (requirement #3), and eventually could enable interactions with other kinds of platforms such as underwater vehicles (requirement #6).

C. Implementation approach

Several languages and protocols can be used to implement distributed object systems. At the time MOOS software development began in earnest (mid 2001), Java and C++ were the strongest implementation language candidates, while CORBA and RMI (both based on TCP/IP) were the strongest distributed object protocol candidates. MBARI engineers examined these alternatives in detail. Evaluation criteria included ease of robust code development, code portability, availability of development tools, performance and resource "footprint" on the MMC node, and perceived future viability of the various solutions. Some of the criteria can be objectively evaluated (e.g. resource footprint), while others (such as ease-of-software development) are more subjective in nature. The team also developed several prototypes to aid in evaluation.

The team selected Java as the implementation language and RMI as the distributed object protocol, for several reasons. In contrast to C++, the Java programming language is quite easy to learn, and is fully object-oriented. Java is also strongly typed, and many programming bugs are caught at compile time rather than run time. Java requires a larger resource footprint than C++; however the memory and persistent storage of the MMC had already been scaled to accommodate the eventuality of Java as an implementation choice. Regarding the oft-cited concern of Java execution performance, major improvements in JVM and JIT technology since 1996 have largely resolved those issues, although *startup* speed of a Java application can be considerably slower than an "equivalent" program written in C++ [9]. The non-deterministic nature of most Java garbage collection implementations can cause unacceptable jitter in "hard real-time" applications having timing requirements of a few milliseconds; however no such stringent timing requirements have been identified for core SIAM components.

The RMI distributed object protocol is tightly integrated into Java and so is quite straightforward to use at an application level, as it "hides" most details of network transport. A drawback of RMI is that by default it interacts only with components written in Java. CORBA on the other hand can interact with components written in many languages, but tends to be comparatively verbose, whether used from Java or C++. The network transport associated with distributed object protocol is important to evaluate, especially for the low-bandwidth wireless link between mooring and shore. We have measured transport overhead for both CORBA and RMI, and have found the two protocols to be similar in this respect.

In mid 2001, several Java solutions suitable to the MMC target were available. Since that time, support for Java in embedded systems has diminished significantly. Likely causes for this loss of support include dissatisfaction with Java performance and footprint requirements on small consumer electronics, as well as the general slowdown in the electronics market over the past few years. As of this writing, a few Java/RMI alternatives are currently available for the MMC, and there appears to be renewed interest by vendors to use Java in cellular phones and other small devices [7,8].

III. SIAM ARCHITECTURE

A. Distributed components

Fig. 1 diagrammatically shows the major distributed objects in a SIAM moored network.

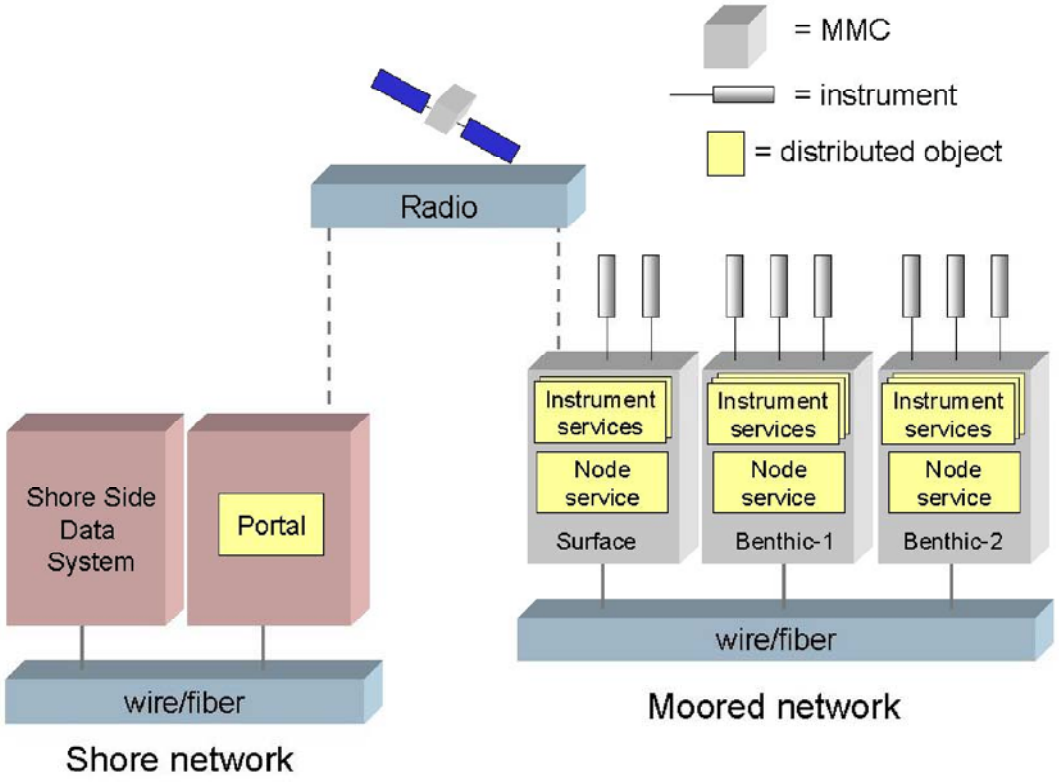

Fig. 1: SIAM distributed object architecture

SIAM distributed objects include:

- Node services: Each physical MMC node hosts a node service that executes on the MMC. Clients interact with a node through methods in the *Node* distributed object interface. These include methods to retrieve the node's IP address, get a list of the node's instrument services (see below), install or remove the instrument service on a specified port, and obtain a list of events scheduled to occur on the node (such as sample acquisitions).

- Instrument services: For each physical instrument installed on a node a corresponding instrument service resides on that node. Clients interact with an instrument through methods in its instrument service object. These include methods to acquire a sample from the instrument, retrieve all logged samples for a specified time window, get the instrument's state and other metadata, turn device power on or off, and retrieve diagnostic information from the instrument serial port.

- The *portal* resides on a shore-based workstation, and has physical access to the intermittent wireless network connecting shore to the deployed moored network. When a wireless connection is available, the portal retrieves data streams from a moored network through the node and instrument services objects, and distributes the data to the Shore Side Data System for further processing, distribution, and long-term archiving.

B. Standard instrument interface

As noted earlier, SIAM is required to deal with many kinds of instruments (requirement #1). Most oceanographic instruments have an RS232 interface through which a "host" computer (the MMC, for example) sends commands and receives instrument data. However there are no widely recognized instrument software protocols; thus each kind of instrument has a unique command set that must be used to interact with the device through its serial port. This overabundance of interfaces poses serious problems to system management.

For example, the system might include "utility components" that acquire and log instrument data, synchronize internal instrument clocks, and control the power for all instruments in the moored network. One solution would be to require that each of those utilities have built-in knowledge of every instrument's unique protocol for the required functionality. This is a poor solution, since the utility software is then tightly "coupled" to the specific deployed instrument protocols. Each time a new kind of instrument is added to the system, or an existing instrument protocol is modified, each utility's software must also be modified.

SIAM takes a different approach by defining a standard software interface to be implemented by all distributed instrument services installed on MOOS. This interface includes methods that are generic to all instruments, such as instrument initialization, sample acquisition, and power control. Thus a client may interact with any instrument through its distributed object interface, without "knowing" specifically what kind of instrument it is dealing with or any details of its serial interface software protocol. The instrument service itself is responsible for interacting with the actual device, and translating standard interface method protocols to the protocols recognized by the actual instrument device. However these details are encapsulated by the service, and hidden from the rest of the system. Thus clients can always access any instrument through the standard interface, and are

insulated from the actual service implementation. New kinds of instruments can be added and existing services can be modified without the need to modify any client code.

Fig. 2 is a UML class diagram that illustrates this concept. The standard interface called *Instrument* is implemented by all instrument service classes, such as *CTD*, *GPS*, and *Fluorometer* services. Since all instrument services implement *Instrument*, any client anywhere on the network can remotely control any instrument through its distributed object. The instrument services must be implemented such that when a client invokes any method in the *Instrument* interface, the service ensures that the correct action is taken for the particular instrument. Note that it is also possible to extend the "remote control" functionality of an instrument; in this example interface *ADCP* extends *Instrument* by adding additional methods, perhaps methods to modify the acoustic parameters of the instrument. The *ADCP service* implements the *ADCP* interface, in addition to the generic *Instrument* interface. If a client "knows" that a distributed object represents not just an *Instrument* but an *ADCP, the* client can remotely utilize the extra *ADCP* methods.

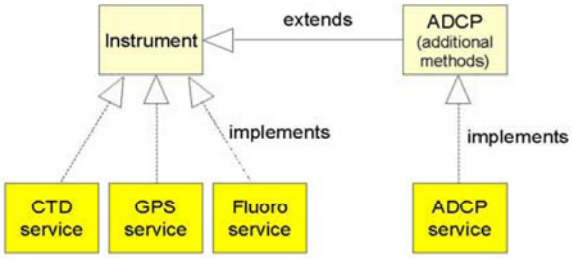

Fig. 2: Instrument interface and service classes

C. Instrument service application framework

Since a service must be developed for each of the many kinds of instruments deployed on MOOS, SIAM has developed a Java *application framework* that greatly simplifies this process. The *InstrumentService* class implements many methods of the *Instrument* interface, as well as other functionality that is common to most instruments, and leaves other methods as abstract (i.e. defined but not implemented). To completely implement a particular instrument service, the programmer extends the *InstrumentService* class, and adds any logic and data necessary to fully implement the *Instrument* interface. This concept is illustrated in Fig. 3.

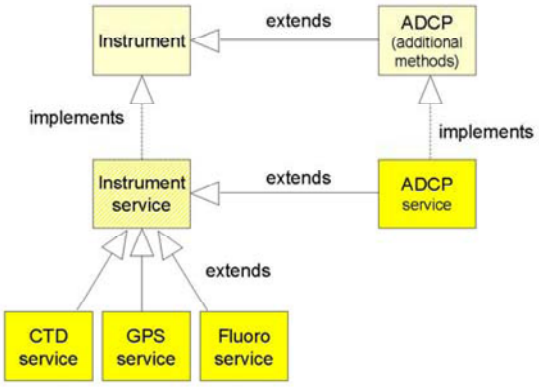

Fig. 3: Instrument interfaces, *InstrumentService* application framework, and implementations

There are many benefits to the application framework approach. Since *InstrumentService* framework class implements most of the functionality needed for a service, relatively minor amounts of subclass code are required to completely implement a specific instrument. We have found that *InstrumentService* typically implements about 75% of a given instrument service. The SIAM development team has focused on making *InstrumentService* robust and efficient; all the instrument service subclasses inherit these virtues. Likewise any improvements to *InstrumentService* are inherited as well.

SIAM's standard instrument interface feature and the instrument service framework approach address the requirement to support many kinds of instruments (requirement #1). Standard instrument interfaces also enable simplified integration of instruments into the moored network by providing generic methods to initialize and control instruments (requirement #5).

D. Instrument pucks

The MBARI instrument puck is a small data storage device that can be physically mated to an instrument's serial port before the instrument is deployed. The puck can be preloaded with information about the instrument, enabling automatic integration of the instrument and its data stream into an observatory [11].

For MOOS, the puck contains the instrument's service software in the form of Java bytecodes, as well as metadata that describe the instrument. The metadata include several elements:

- A unique "device ID" assigned to the particular instrument, similar to a serial number. This number uniquely identifies the instrument and its data stream.
- XML that describes the instrument, including make and model, serial number, calibration constants, a description of the data "records" produced by the instrument, including field names, types, and units.
- Initial instrument parameters and settings such as sampling rate, and description of instrument power characteristics

When the instrument and its puck are connected to an MMC node's serial port and power is applied, the puck enters "puck mode", allowing the MMC to extract the contents of the puck by issuing the appropriate command in the standard puck protocol. Once the MMC has received the contents of the puck, it issues another standard command to switch the puck into a "pass through" mode, allowing the host platform to communicate directly with the instrument, without further interaction with the puck. The MMC then executes the instrument service Java code that it extracted from the puck, using elements of the extracted metadata as startup parameters for the service. The service exports its distributed object proxy, making the service available to clients, either locally or across the network. Local clients include SIAM components that manage the instrument power, sampling schedule, and other components. Thus the service code and metadata loaded from the puck serve to describe the instrument to the rest of the system, enabling automatic integration (requirement #5).

As described in [5,11] there are a number of possible puck implementations, including an *embedded puck*, in which the puck protocol and storage are integrated within the instrument itself, eliminating the need for external hardware and connectors. In some cases it makes sense to use a "virtual" puck, in which the puck contents, in the form of a file, are pre-installed on the node.

E. Self-describing data streams

Each SIAM instrument service generates a self-describing data stream for its instrument. Self-describing data streams are at the heart of automated data management. A data stream is a time-sequential series of SIAM packets, of which there are several types. *Sensor data packets* contain "raw" data bytes acquired by the service from the instrument through its serial port. *Metadata packets* contain metadata read from the puck, generated by the service, or read from the instrument itself. *Message packets* contain human readable text messages, usually indicating events of interest; message packets can be generated by autonomous clients or by human users. Thus a data stream consists of instrument data interleaved with metadata and messages, which describe the context of the data. Packets from the data streams are periodically retrieved through the instrument service by the onshore portal client and distributed to the Shore Side Data System (Fig. 1). The SSDS uses the data streams' metadata packets to automatically process the sensor data and with minimal or no human intervention, and to provide comprehensive query and visualization functions to users on shore.

As the instrument service operates, it appends packets to the data stream. Using behavior inherited from the SIAM instrument service framework, the instrument service ensures that the first packet produced after startup is a metadata packet describing the data that will follow. This metadata packet contains the XML document read from the puck, the service's state, and state information provided by the instrument device itself.

Later, if the instrument service detects a "significant" change in the data acquisition context, a metadata packet that reflects the new state is generated and appended to the data stream. The SIAM instrument service framework defines generic changes that are considered significant, for example when a sampling rate is changed or the instrument configuration is changed. Instrument service subclasses may extend the definition of significant change.

Each instrument data stream is stored locally on the MMC file system as it is generated by the instrument's service. When the wireless link to shore is available, the onshore portal client requests the latest packets from each instrument service. The service then retrieves the appropriate packets from the stored data stream and returns them to the portal. The instrument's entire data stream is stored on the MMC until explicitly re-initialized by the user, thus ensuring that no data is lost in the event of communication failures (requirement #4).

Each packet in a data stream is given a sequence number as it generated, thus enabling detection of missing, duplicated, or out-of-order packets that may result from wireless link failures or other errors that may occur during transfer across the network. Each packet also includes a reference to the most recent preceding metadata in the stream, thus ensuring that the proper context of each packet can be traced.

IV. SOME RESULTS AND LESSONS LEARNED

Two MOOS mooring surface nodes were deployed in Monterey Bay during the first half of 2004. Key goals of these early deployments are to "shake down" and refine the SIAM system and evaluate its performance. This analysis is ongoing, and as yet there is not a substantial body of quantitative results to report. Qualitatively speaking, the SIAM infrastructure and applications are performing nominally, delivering science and engineering data reliably. In this section we examine and evaluate some of SIAM's novel features in more detail.

A. Distributed object protocol over low-bandwidth wireless links

Two wireless modems – Freewave 900 MHz radio, and Globalstar satellite modem - are currently supported on MOOS. Internet protocols can be utilized over either of these links via *the point-to-point protocol* (ppp). Internet protocols underlie Java RMI, telnet, ssh, ftp, ping, and many other network utilities. The protocols are well understood, robust, and come with many useful features and capabilities. For example, Java RMI is based on TCP-IP protocols that guarantee reliable packet delivery and error detection. Use of uniform underlying protocols throughout the network, including on the wireless link, greatly simplifies the distributed system design. Thus the benefits of utilizing Internet protocols on the wireless link are clear; but what are the costs?

These costs (in power and satellite airtime) are amplified by limitations in wireless bandwidth, intermittence of wireless connections, the overhead of various protocols such as RMI, and the power-costs due to resultant on-air time. The Freewave modem can run at 38.4 Kbps, while Globalstar is much slower, at 7800 bps. Some of the link intermittence is planned, since the wireless modem is periodically switched off to conserve power and satellite airtime charges. Other intermittence is unpredictable, due to atmospheric and other environmental effects.

The RMI protocol is discussed in detail by [6]; here we describe the process in general terms. To obtain a distributed object, the client first sends a TCP/IP network message to the machine where the service resides, which returns a message that describes the distributed object's interface and its state. When the client subsequently invokes a method on this distributed object, the object sends a message across the network to the service via TCP/IP, specifying which method is being invoked and any input parameter values. The service executes the method and sends a TCP/IP message back to the distributed object on the client machine, specifying the return and output values. When objects (as opposed to primitive values) are specified as parameters or return values, an additional overhead of "object serialization" is incurred, as the object is converted to a stream of bytes prior to transfer.

As part of our early design verification and development process, we developed a simple test to determine RMI overhead. We defined a simple distributed object interface called *Benchmark*, and a simple class called *TObject* as defined schematically in Fig. 4.

```
/** Simple distributed object interface
*/
interface Benchmark {

  /** No parameters, no return
      value. */
  void emptyTest();

  /** Primitive parameters and
      return value. */
  long primitiveTest(short a,
                     double b,
                     long c);

  /** Returns an object. */
  TObject objTest(byte fillValue);
}

/** Simple object returned by
    Benchmark.objTest() */
class TObject {

  short x;
  long y;
  double z;

  byte array[] = new byte[1000];
}
```

Fig. 4: Benchmark RMI interface

The Benchmark interface consists of three methods. Method emptyTest() takes no parameters and doesn't return a value; this should be the method with lowest overhead. Method primitiveTest() takes three primitive parameters and returns a primitive value. Finally, method objTest() takes a primitive value and returns an object of class TObject. TObject contains a short, long, and double value, and a 1000-byte array which is initialized to random numbers.

We then wrote an RMI service that implements *Benchmark*, and also wrote a client that invokes the three Benchmark methods via RMI.

Our test setup consisted of two Linux workstations interconnected by a DB-9 serial cable; the *point-to-point protocol daemon (pppd*) was run on both machines to establish an IP link across the serial cable. The number of bytes transferred across the serial cable during RMI transactions was measured with an in-line protocol analyzer. We ran tests with ppp compression disabled and enabled.

Table 1: Overhead of RMI distributed object protocol

	Bytes transferred		
	application	ppp	compressed ppp
step 1: get distributed object service = (BenchMark)Naming.lookup(url);	N/A	2839	873
step 2: service.emptyTest();	0	178	71
step 3: long ret = service.primitiveTest(short sval, double dval, long lval);	26	204	74
step 4: TObject obj = service.objTest(byte bval);	1019	1432	1143
step 5: disconnect from service System.exit()	N/A	341	196

Table 1 shows the amount of data transferred between client and service machines during the process of 1) obtaining the service remote object, 2) invoking the object's *emptyTest()* method, 3) *primtiveTest()*, 4) *objTest()*, and finally 5) disconnecting from the service. The transaction is shown in the first column. In the next column is the number of "application" bytes, equal to the number of bytes in method parameters and return values. The next column shows the measured number of bytes actually transferred across the wire at the ppp link layer, with ppp compression disabled. The last column shows the

number of ppp bytes transferred when default ppp compression is enabled.

In step 1, the client obtains the Benchmark service's distributed object across the network, which in this case requires the exchange of 2839 bytes. In step 2, the client invokes the service's *emptyTest()* method. As this method takes no parameters and does not return a value, the 178 ppp bytes transferred represent the combined minimum overhead of RMI and the underlying TCP, IP, and ppp protocols. This interpretation is consistent with the total ppp bytes transferred during invocation of *primitiveTest()* (step 3), which is equal to the application bytes (26) plus the overhead (178 bytes) for a ppp total of 204 bytes. Invocation of *objTest()* incurs additional overhead due to object serialization, which in this case appears to be 235 bytes (1432 – 1019 – 178).

A very useful feature of ppp is its lossless compression capability, results of which are shown in the last column of Table 1; compression results in significant savings in bandwidth. The compression ratio is dependent the data content, so the values shown are examples only. In particular the relatively minor compression (0.8) during invocation of *objTest()* is due to the fact that the contents of the 1000-byte array contained within the returned *TObject* was deliberately randomized by the service before returning it to the client.

Table 2 shows some statistics for telemetry retrieved from the MOOS Test Mooring currently deployed in Monterey Bay, over a period of 108 days. Telemetry was retrieved via Java RMI over the 7800 bps Globalstar satellite telemetry link. The surface node initiates a satellite connection with the shore-based portal at a configurable interval, which for the period shown varied between 30 minutes and one hour. Over a period of 108 days, more than 138 megabytes were retrieved, corresponding to an average download rate of about 1.3 megabytes per day. Note also that the satellite link is subject to intermittence; about 5 percent of the regularly scheduled links were terminated prematurely.

D. Java and rapid development

The use of Java on embedded systems is not widespread in the oceanographic community, and is one of the novel features of SIAM. We took advantage of many of Java's benefits. Java's extremely rich packages (i.e. libraries) save development time by reducing the amount of software needed for low-level infrastructure, allowing developers to focus on application-level implementation. In general, Java packages are robust and well documented. There is also a wide developer base; it is almost always possible to find code to fit your needs when it doesn't exist in the standard Java packages. Since Java is object-oriented, existing classes can be extended to fit particular needs.

Java's extensibility proved very useful in many cases. For example, we encountered particularly serious problems when using RMI over an intermittent wireless link; the problem manifested itself by gradual increase in the number of Java threads on the MMC and subsequent resource exhaustion. We traced the problem to the implementation of the Java socket classes. Though diagnosing the problem required considerable effort, we were able to fix this problem with a minimal amount of "custom" code, by extending Java's *ServerSocket* and *RMISocketFactory* classes.

Among the drawbacks experienced with embedded Java are the limited choice and support of JVMs for embedded platforms. To date, there are only a few JVMs that run on our platform and that support all of the features required by SIAM. At times, perceived deficiencies in the embedded JVM or its packages

Table 2: MOOS Test Mooring telemetry received via Globalstar satellite

Data stream	Sample interval (minutes)	Total samples retrieved	Total bytes retrieved
Triaxys wave sensor	30	4922	1941049
ASIMET short wavlength radiometer	10	15437	5159930
ASIMET long wavelength radiometer	10	15436	5994108
ASIMET wind sensor	10	15398	5793036
ASIMET relative humidity and temperature	10	15437	5422523
Garmin GPS	10	15431	5975157
RDI Workhorse ADCP	60	2574	4303122
Mooring power can	10	15448	52017200
Main environmental	10	15431	2114895
EOM cable diagnostics	10	15461	7379389
Node diagnostics	60	2648	21146895
Total	N/A	133623	138394199

Measurement interval: 108 days (April 23 through August 9, 2004)
Total Globalstar connections: 3487
Total broken connections: 187 (5%)
Average download duration: 127 ± 66 seconds
Total connection time: 182 hours

required us to develop workarounds. The isolation from system-dependent details that Java provides, while beneficial in many respects, can be frustrating in embedded systems. Although Java provides some mechanisms for doing so, it can be awkward to gain access to hardware and software outside of the JVM. Java's RMI works well over stable, hard-wired links, but considerable troubleshooting was required for reliable operation over an intermittent wireless link, as described above. Thus while the use of Java has brought positive results, this choice was not made without compromises.

V. FUTURE DEVELOPMENT

In this section we briefly discuss a few areas of possible future SIAM development.

A. Power management

Currently a SIAM component called *sleep manager* resides on each node; sleep manager seeks to minimize node power usage by putting the MMC's main processor into "sleep mode" when no activities are scheduled in the immediate future. All onboard SIAM components requiring power management implement an interface that permits sleep manager to determine the component's scheduled power needs. Based on this information, sleep manager sets an appropriate wakeup time before going to sleep. (The mechanisms that put the CPU into a sleep state and wake it up again lie outside the JVM, at the Linux driver level, thus requiring non-Java sleep manager components.) In addition, I/O activity on the node's serial ports causes the main processor to wakeup from sleep, providing a mechanism for responding to unpredicted events.

Since a node may lie along a network communication route, it must be prepared to route network traffic at any time. The MMC's *Medusa* communication processor provides this routing functionality, and routes packets even if the node's main processor is asleep [5].

The node currently queries each instrument service for its "power policy" – "continuous", "when sampling", or "never" - and applies power to the instrument accordingly. But at present the node software has no knowledge of how much power the instrument actually consumes, and does not attempt to balance loads or arbitrate power requests among multiple instruments. The facts that MOOS power distribution hardware protects against catastrophic power failure [5], and that our early deployments include few high-power instruments make this rather simple approach feasible at this time. However, to fully use available power, to avoid unpredicted shutdown of individual instruments and nodes, and to ensure robust communications in the moored network, additional software must be developed. We refer to these capabilities *as distributed power management*, as it seems likely that some components will need knowledge from all nodes in the network to effectively manage available power. Furthermore, hardware within the MMC's dual port adapter currently provides an actual measurement of power drawn by an instrument, but SIAM does not presently utilize this information. It is clear that in this networked environment, the power distribution system must be robust enough to curb excessive power use without catastrophic failures. Beyond that, some degree of centralized power monitoring and management will be required, if the system is to fulfill its promise of self-configuration. This implies that the system must be able to dynamically determine for itself the true load presented by its payload, rather than relying on nominal stated values, which may be too conservative (allowing resources to go unused), or too liberal (exceeding the power budget). Much work remains to be done in this area.

B. SIAM on cable-to-shore systems

Cable-to-shore observatories such as the Monterey Accelerated Research System (MARS) offer continuous network connectivity and high power to instruments deployed in the deep ocean. Determining the detailed requirements for a cable-to-shore observatory will require definition and analysis of a wide range of use scenarios, and these tasks are yet to be completed [12]. However for the purpose of discussion, let us assume that cable-to-shore observatory requirements include the following:

- Deployed instruments must be controllable from shore.
- The system must be able to *autonomously* detect events of interest and respond to them.
- Science data must be reliably associated with metadata that describes the instrument and its state at the time the data was acquired.
- System configuration management must be reasonably simple and scalable to large numbers of instruments

We have shown how SIAM's distributed object instrument services, standard instrument interfaces, and self-describing data streams meet the equivalent requirements in the MOOS moored observatory; we believe that similar solutions could work well in a cable-to-shore system. For example, SIAM's instrument services and standard interfaces enable efficient generic mechanisms for "administrative" functions such as instrument power control, data acquisition, and retrieval of diagnostic information. SIAM's "plug-and-work" integration model and self-describing data streams simplify the system configuration process and help ensure quality of science data.

As currently specified, the seafloor MARS science node will provide power and data connections for instruments, but will not have significant computing capability. Rather, the node's connectors will be treated as "remote" instrument ports, but all generation of instrument-specific commands and all data processing will be carried out on shore [13]. In the MARS system, there are no underwater processors capable of running a Java virtual machine, let alone hosting a SIAM instrument service. However, a major advantage of SIAM architecture is its distributed object nature. The SIAM instrument service could actually run on a Java-capable workstation *on shore*, and communicate with its instrument through the MARS cable using the protocol specified for MARS. In fact, the instrument service design is deliberately insulated from the details of its operating environment. For example, access to the MOOS Mooring Controller's

serial ports is implemented by a Java class that is quite dependent on MMC hardware architecture; but the class implements an abstract *InstrumentPort* interface which is quite generic. The instrument service only "sees" the abstract *InstrumentPort* interface, and hence is insulated from the MMC hardware details. A class called *MARSInstrumentPort* could implement *InstrumentPort* using the appropriate MARS protocols, while having no effect whatsoever on the instrument service code. The SIAM data, metadata, and message packet classes that define a self-describing data stream are also quite independent of hardware. Thus while some SIAM classes, such as *NodeService* are MOOS-specific and would not be applicable to MARS, it is likely that the SIAM *InstrumentService* and data stream classes are quite portable.

We also note that SIAM components might be considered as "optional" for MARS, perhaps coexisting with other MARS software infrastructure. During the coming year we plan to examine the utility and feasibility of porting elements of SIAM to MARS.

ACKNOWLEDGEMENTS

We thank MOOS project manager Keith Raybould of MBARI for his indefatigable support and encouragement. We thank SIAM users in MBARI's Science, Observatory Support, and Support Engineering groups for their invaluable collaboration and feedback, especially Mike Kelley, Francisco Chavez, and Mark "Zorba" Pickerell. We thank the rest of the MOOS engineering team for their electrical and hardware support; particular thanks go to Lance McBride for puck development, Wayne Radochonski for MMC development, and also Ed Mellinger, Andy Hamilton, Jon Erickson, and Johnny Ferreira. Brent Roman and Karen Salamy provided invaluable software development and test support. SIAM work is funded by the David and Lucille Packard Foundation.

REFERENCES

[1] M. Chaffey, E. Mellinger, and W. Paul, "Communication and power to the seafloor: MBARI's ocean observing system mooring concept", *MTS/IEEE Oceans 2001 Conference Proceedings*, November 2001.

[2] T. O'Reilly, D. Edgington, D. Davis et al, "'Smart network' infrastructure for the MBARI ocean observing system", *MTS/IEEE Oceans 2001 Conference Proceedings*, November 2001.

[3] J. Graybeal, K. Gomes, M. McCann, B. Schlining, R. Schramm, D. Wilkin, "MBARI's Operational, extensible data management for ocean observatories", *The Third International Workshop on Scientific Use of Submarine Cables and Related Technologies, Tokyo*, 2003.

[4] Wolfgang Emmerich, *Engineering Distributed Objects*, John Wiley and Sons, 2000

[5] M. Chaffey, L. Bird, J. Erickson, et al, "MBARI's buoy-based seafloor observatory design", *MTS/IEEE Oceans 2004 Conference Proceedings*, in press.

[6] *The Remote Method Invocation Specification*, Sun Microsystems, 2002

[7] "Aplix and Motorola announce advanced software for Java-enabled phones", *The Wi-Fi Technology Forum*, http://www.wi-fitechnology.com/displayarticle1371.html, August 16, 2004

[8] "Enfora partners with Esmertec to enable wireless Java technology in their Gsm/gprs Oem devices", *TMCnet.com*, http://www.tmcnet.com/usubmit/2004/Aug/1067528.htm, August 24, 2004

[9] J.P.Lewis and Ulrich Neumann , "Performance of Java versus C++", http://www.idiom.com/~zilla/Computer/javaCbenchmark.html, 2004

[10] T Meese, D Edgington, W Radochonski, S Jensen, K Headley, "A New Mooring Controller: an evolution of the OASIS Mooring Controller toward a distributed Ocean Observing System," *MTS/IEEE Oceans 2001 Conference Proceedings*, November 2001.

[11] K Headley, D Davis, D Edgington, L McBride, T O'Reilly, M Risi, "Managing sensor networks and metadata in ocean observatories using instrument pucks," *The Third International Workshop on Scientific Use of Submarine Cables and Related Technologies, Tokyo*, 2003.

[12] A. Chave, G. Waterworth, A. Maffei, G. Massion, "Cabled ocean observatory systems", *Marine Technology Society Journal*, vol. 38, no. 2, 2004

[13] *MARS Critical Design Review*, MBARI, unpublished