Title: Integrating Instruments and Sensors into the Grid with CIMA Web Services

Authors: Donald F. McMullen, Pervasive Technology Labs at Indiana University, Bloomington, IN USA; Kenneth Chiu, State University of New York, Binghamton, NY USA

Abstract: As network attached instruments and sensors become available new opportunities for expanding their accessibility, usefulness and throughput present themselves. One of the keys to integrating instruments into computing and storage grids is to provide a standard methodology for developing interfaces that existing and future instruments can provide and data acquisition and reduction applications can rely on. In this paper we present an overview of the Common Instrument Middleware Architecture (CIMA), a Web Services based approach to making instruments and sensors network accessible in a standards-based, uniform way, and for interacting remotely with instruments and the data they produce. Some of the issues CIMA addresses include: standardization of the network protocol for interacting with instruments and sensors, flexibility in the underlying network transport, efficient and high throughput data transport, the availability (or lack of) computational, storage and networking resources at the instrument or sensor platform, evolution of instrument design, and reuse of data acquisition and processing codes.

Key words: middleware, instruments, sensors, Grid, Web Services

Introduction

In contemporary scientific thought observed data is considered primarily as a computational commodity or raw material, to be turned into knowledge through reduction, fusion and analysis. As such, considerable value has been placed on the latter part of this "value-add chain", in the reduction and analysis phase, which can be performed off-line with respect to the instrument that collected the data and outside any possible interaction between the analyzer of the data and the instrument itself. Big data projects organized around instruments such as detectors at the Large Hadron Collider have yielded useful application test beds such as GriPhyN [1] and iVDGL [2], that focus on data somewhat after it has been acquired from the instrument. These projects by design largely ignore the instrument from whence the data comes. Unfortunately, with many major

instrument facilities, researchers with the expertise to judge the quality of appropriateness of the data may not have access to the proper tools to perform this analysis until well after the data has been collected, and often the data are not available in the critical initial stages of the experiment. This focus on data as separate from its source and the operations necessary to control instruments in order to generate, acquire and filter data have left a gap in the Grid. To fill this gap with middleware that makes instruments first class citizens of the Grid we need to solve a number of difficult problems: creating a broadly applicable architecture for middleware in a variety of instrument settings; a diversity of control software on many platforms including embedded system controllers; different rates of data production; aggregates of large numbers of sensors (*e.g.*, a seismic array); heterogeneity of sensors within an instrument; real-time fusion of heterogeneous information from many sources to create one "instrument reading" (*e.g.*, state variables from several sites in a climate monitoring application); and support for pipelined or workflow oriented data reduction and analysis.

To complicate the picture even further, many specialized instrument resources are of unique construction either by virtue of intrinsic design, or by variability in product characteristics between product generations or among vendors when sensors are obtained off-the-shelf from manufacturers. This one characteristic causes continual re-invention of driver software and frequently results in redesign of data output formats from the instrument. As a consequence, software at later stages in the reduction and analysis pipeline is potentially subject to rewriting when any changes are made to the instrument, even if changes are remedial with no new capabilities being added.

One way to deal with the problem of software dependence on the details of hardware design is to provide a middleware layer to abstract control of, and data acquisition from, instruments. A primary design goal of this middleware is to facilitate integration of instruments into current computing and storage Grids, and to leverage Grid-based services with the eventual aim of composing these into a workflow that reflects a particular line of scientific inquiry.

A common middleware layer and meta-standards for instruments will allow instrument users to build, test and deploy new experiment software, and to transparently reuse existing software when instruments are modified or upgraded. An instrument middleware layer will also allow instrument developers to open up their products to take advantage of a broader range of control application and data sinks (data fusion and analysis software).

Instruments and Sensors as Grid Services

Our approach to developing instrument middleware for the Grid leverages existing and emerging standards in Web Services, in particular the Open Grid Services Architecture (OGSA), the Globus adaptation of Web Services.

A Web Service is a network accessible interface to some application functionality based on a standard messaging framework. The interfaces, described in Web Services Definition Language [3] (WSDL) are fundamentally message-based but can be interpreted in terms of procedure calls or document exchanges. The service's interface is platform and language "agnostic" and messages can be delivered by any protocol for which a binding has been defined, such as SOAP over HTTP, SMTP or message oriented protocols like Java Messaging Services [4] or NaradaBrokering [5]. Services are defined as collections of ports each with an abstract definition, the port type, and a concrete definition, the binding, at a Web Services *endpoint* or URL pointing to the service.

In addition to the description, packaging and transport elements the Web Services model is extended by a number of other W3C efforts [6] to provide service discovery (UDDI and WS-Inspection); identity, security and encryption (SAML, XML-DSig, XML-Enc); and workflow (BPEL4WS [7]).

The Web Services model is an extremely attractive one for scientific computing where issues of contract complexity (requirements for using a service) are minimal. Several years ago a proposal was made to integrate the Web Services model into Globus [8]. As a step in realizing a Web Services-based Grid the Globus group has issued specifications making Globus more useful for casting applications as Web Services. The "Open Grid Services Infrastructure Reference

Implementation" that implements the "Open Grid Services Infrastructure" (OGSI) [9] specification define *grid services* as extensions to W3C Web Services. Early in 2004 in response to evolution in the WSDL specification and the availability of other WS specifications the OGSI was re-factored into the Web Services Resource Framework (WS-RF) [10]. WS-RF provides a much cleaner and more standards-compliant means for defining stateful Web Services and interacting with them through operations on *resource properties.*

Requirements for Grid Instrument Middleware

Grid computing has brought a degree of coherence to the development effort in large-scale computing projects, and has enhanced the ability to focus distributed computing and storage resources on a single large-scale application. Instruments, however, have been either treated on an *ad hoc* basis or are proxied by the files they generate. CIMA aims to bring instruments within the Grid as first class participants via a middleware architecture that interoperates both syntactically and semantically with Grid standards, yet still satisfies the performance, compatibility, and reliability demands of the underlying hardware it mediates.

CIMA addresses requirements for functional transparency, resource-oriented stateful services, protocol independence and efficient communications as described below.

Functional transparency. Each function of the instrument must be completely and accurately represented by the Grid interfaces provided by the middleware. Grid applications must be able to develop a complete operational model of the instrument from minimal knowledge.

Resource oriented stateful services. Instrument control and acquisition details are typically represented procedurally in the instrument's API, or in the user interface of the vendor-supplied control application. Representational State Transfer (REST) [11] provides an alternative to a procedural API that is more interoperable, extensible, and scalable. The REST approach defines a small number of operations such as the HTTP actions PUT and GET on a large number of resources (*e.g.*, URLs in the analogy to HTTP). OGSI has adopted some REST tenets in the form of Service Data Elements (SDEs). CIMA uses SDE or resource property constructs to expose an instrument's

characteristics as metadata so that an application can discover not only the existence and network address of an instrument, but also the channels provided to the application, their meaning, metrics, and use.

Protocol independence. Though SOAP is used as the common protocol, it is not appropriate for every situation. Performance is modest [12] and may be limiting, so middleware must be able to use other protocols. Also, it may be the case that our middleware is connected to the instrument via a wire protocol of some kind other than IP, in which case, we need a standard way of incorporating vendor-specific protocols into our acquisition software without making the acquisition software vendor-specific.

Efficient communications. For situations requiring high communications throughput, the variety of message-exchange patterns provided by message-oriented middleware may be more appropriate than a Remote Procedure Call approach. For the most demanding applications even messages may incur too much overhead since each message is typically treated as a datagram and separately routed in the middleware layer. In situations where each message is small, but the rate is high a more efficient approach is to use data *channels* modeled after the virtual circuit concept. Data is streamed through the channel, with much of the per datum overhead shifted to the relatively infrequent channel set-up and tear-down steps.

The Common Instrument Middleware Architecture

Design Goals

We intend CIMA to be used in a wide variety of instruments and sensors so our design goals were derived from a representative set of use cases including two similar, but distinct scenarios: *remote access* and *distributed operation*. Remote access provides a user with a simulacrum of an instrument's controls and displays without significantly changing the way a user interacts with the controls or data.

In contrast, in distributed operation the data and control channels of the core instrument facility are distributed within a virtual organization that may consist of the scientist's institution, off-

site computational resources and application servers, and storage sites such as a data warehousing facility. This has the potential to significantly change the way instruments are used and encourage innovation, by allowing many researchers to simultaneously interact with an instrument facility, and investigate new software tools and infrastructure. Under ideal conditions many researchers could interact with an instrument facility as if each were the sole user, and explore innovations that in a conventional, centralized setting would have an unacceptable impact on other users.

Some of the requirements for distributed operation that we have identified for inclusion in CIMA are listed below.

Zero configuration for applications. A central design requirement was that CIMA applications must be able to develop an operational model of the instrument from a minimum of external knowledge, which requires that each function of the instrument be completely and accurately described. This requirement will encourage the kind of loose-coupling that promotes interoperability, thus reducing the burden of managing and administering a large variety of instruments.

Interoperable. Interoperability may be required in collaborations, where one research group needs access to "grid-enabled" instruments maintained by another group. In order to achieve this, the specification of a sensor should be complete enough so that third party applications can access it without additional information. Also, minor changes to sensor functionality should not require deep code changes in acquisition or analysis packages. Another goal in CIMA was to make the functionality independent of the data structures being used. This is achieved by using a *parcel*, which is a XML document of the data and meta-data involved. The parcel data structure is described below.

Efficient. Some instruments and sensors, especially when aggregated, may generate data at high rates. Efficient transport is thus important for CIMA. If the data rate is higher than the rate at which the system can transfer them, then there could be data loss or system crashes. Even though buffering could be used to handle mismatches between the data rates for a short period, it will not be possible to operate indefinitely, since the buffers would overflow.

Lightweight implementations. Sensors may need to be deployed at locations subject to electrical and processing power constraints with parallel limitations on computing power, memory and secondary storage. The design of CIMA should take these potential resource limitations in to account.

Data flow and ad hoc processing pipelines. CIMA should implicitly support the notion of composable data flows through a concept of *intermediaries*, or code modules that forward data on an instrument's channel. Intermediaries can be used as signal processing filters or buffers between a sensor and the consumer, and can off-load the work of serving multiple consumers from a single sensor. Intermediaries can also act as security gateways for the sensor node by allowing only particular intermediaries to connect to it.

CIMA Implementation

Instruments are hierarchical in nature. For example, sensors and actuators in an optical astronomy observatory consist of dome controls, telescope positioning, optics selection, imaging detectors, and environmental conditions such as temperature, humidity, cloud cover, moon phase and location. These can be logically (or physically) grouped into hierarchies from which data can be aggregated or for which control commands affect multiple sensors or actuators. A platform positioning command, for instance, may affect the position of all sensors mounted on the platform.

Client applications are simplified if they can access one stream of data as opposed to multiple streams. This can be achieved if the top level instrument can aggregate the data from lower level instruments. The parent instrument should be able to provide information about its children such as their interfaces, data rates and other meta-data, so that a client application may query the parent and retrieve them. In CIMA, instruments may be arranged as a hierarchy. In this arrangement a parent instrument is considered to be composed of multiple child instruments in a nested manner, with no limit to the depth of nesting or to the actual location of the child components. This is achieved using software modules or *plug-ins* at each parent instrument that aggregate data from its child instruments and re-send the composite as if all the data is being generated by the parent.

One current shortcoming of instruments and sensors is that the applications that use them (e.g., data acquisition codes) must have a complete operational model of the instruments and sensors they work with built in as lines of code. This makes maintaining investments in these codes difficult and expensive when the underlying instrument hardware is improved. A primary design goal for this project is to externalize the instrument description so that applications can build an operational model "on the fly". This approach makes it possible to preserve investments in data acquisition codes as instrument hardware evolves, and to allow the same code to be used with several similar types of instruments or sensors. This is particularly important in situations where the instrument or sensors and the related acquisition and analysis codes are in their early stages of development and undergoing rapid change.

Architecture

Instrument Model

Our current instrument model is shown in Figure 1. An instrument consists of a logical group of one or more sensors and actuators. Each sensor may serve zero or more consumers and each actuator may be used by one or more drivers (if serial use is not enforced). A consumer can receive data from one or more sensors through a *channel*.

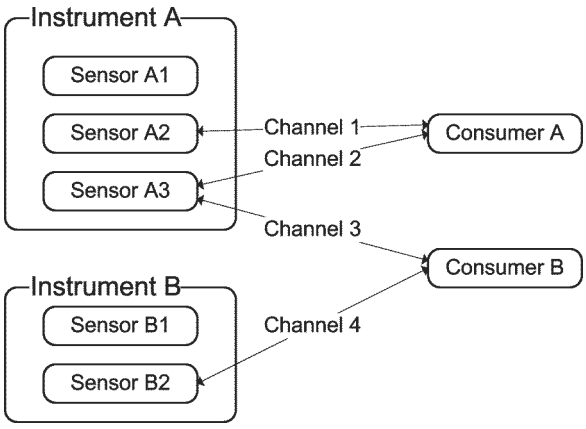

Figure 1. Instrument Model.

Channels

The application consists of *channels* and *plug-in modules*. Channels provide a generic framework for communication between components while plug-ins implement application-specific functionality. The operations needed at the source and sink to send and receive messages may be the same across many types of instruments, while the code needed to construct or parse messages specific to a particular sensor are unique to that hardware. Plug-ins perform these instrument-

specific data handling functions. Because the content of messages varies depending on the underlying hardware, specific plug-ins may be needed at source and sink. Although plug-ins can be "hard coded" for a particular type of sensor, one of the uses for the instrument description embedded in the instrument's grid service is to help applications to parse data messages from, and build control messages to CIMA-enabled instruments and sensors.

Intermediaries, which just forward every incoming message, do not need to have a plug-in as they are not required to interpret the message. However, an intermediary that needs to perform some processing of the message, such as calculate the average of data values and send only the result, may need to have a plug-in to perform such tasks. The use of ontologies and semantic mediation technologies [19] can also be used to build generic modules for such functions that can operate across instrument types.

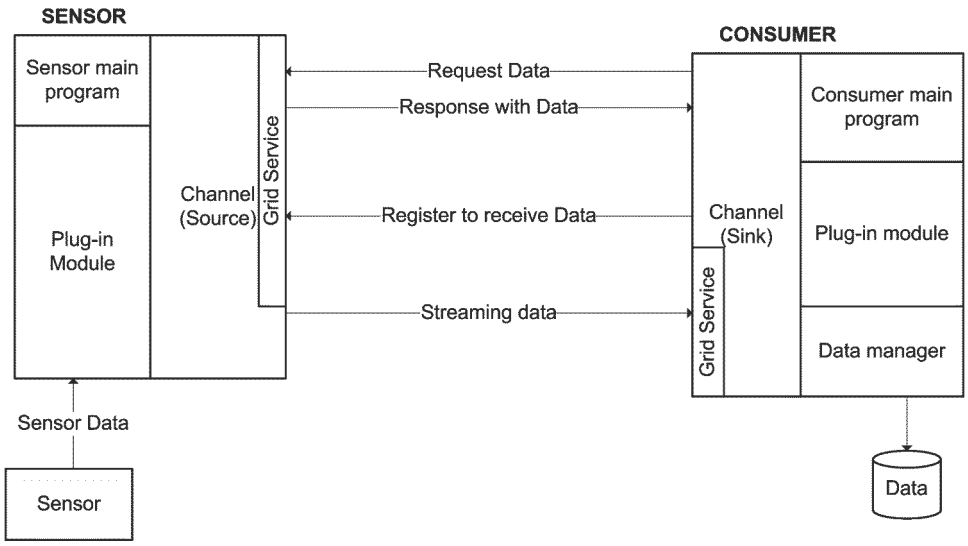

Figure 2. Communication via the Channel.

As shown in Figure 2, the channel handles all the communications between the sensor (or intermediary) and the data consumer. Specific plug-in modules, both at the sensor end and the consumer end, implement the sensor-specific behavior. A channel is bounded at the instrument and the consumer by a *source* grid service (*e.g.* the sensor end) and a *sink* grid service (consumer end). The grid service at a channel source primarily handles control information from the data consumer, such as registering with the sensor to receive sensor data and un-registering to stop receiving data. It also responds to one-time requests for sensor data and other functions including returning a

description of its instrument and other grid-related housekeeping functions. The grid service at the channel sink receives streaming data and status messages such as "data not available" from the sensor.

A data consumer can choose to receive a single data value (request-response or pull model) or continuously receive data values (streaming or push model). The consumer must have a grid service at its end for the push model, while this is not a requirement in the case of the pull model. In the pull model, the consumer sends a request for sensor data and the sensor responds with the current value of its data. However, in push model, first the consumer registers with the sensor to receive data, indicating the data rate required and its port number to which it wants the data to be sent. The sensor then starts a thread which will continuously poll its sensor data at the requested rate and send them to the requested port at the consumer. The thread will continue to run until it receives an un-register request from the consumer or after a given number of attempts to send data fails. The push model for channels is an example of the *multi-response* service interaction pattern as described in [20].

In push model, the consumer also specifies the interval between two data messages. The sensor then starts sending messages at the rate determined by this interval. For example, a temperature reading can be sent every five seconds. However, if the consumer registers with a zero interval, then data is sent as and when they are available. This method is used when the rate at which data being generated is not known.

Communication Protocols

We have used SOAP [13], since it is the most widely accepted standard for Web Services. The current implementation of the channel uses gSOAP [14] to handle the serialization and deserialization of SOAP messages, and the communication. HTTP is used as the transport layer.

In addition to using HTTP, we have also developed prototype systems which use Antelope [15] and Binary XML for Scientific Applications (BXSA), respectively for the transport layer. Antelope provides an Object Ring Buffer (ORB), which enables buffering between the instrument

and the ultimate receiver. The buffer is useful to compensate for mismatches between the sender's and receiver's data rates as well as to store data temporarily in cases where the receiver goes offline for short periods.

In BXSA, an XML infoset [16] is sent as binary data, as opposed to the usual textual format. This could significantly improve performance when sending large amounts of numerical data. Also, BXSA provides the same interfaces available for accessing textual XML, eliminating the need for separate APIs, data model and a type system. We are looking at possibilities of using BXSA with CIMA.

The Parcel Data Structure

Different sensors generate data in different formats. For example, a temperature sensor data would typically be a double precision value while in the case of an image detector, it would be a binary file of the image. One approach for accommodating different types of data formats is to provide generic methods such as *SendDouble()*, *SendBinary()*, *SendString()*, etc. corresponding to each data type. Application programmers would then use the appropriate method depending on their sensor data. This approach has several disadvantages. The first is that the API becomes dense with a method having to be implemented for each data type. Also, providing support for a new data type would require adding new methods, thereby changing the interface.

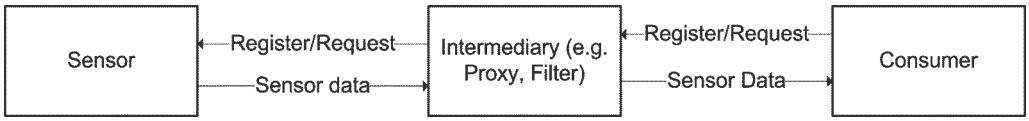

Figure 3. Intermediary between sensor and end consumer.

Another problem with this approach surfaces if there is an intermediary between the sensor and the end consumer, as shown in Figure 3 above. A given sensor or a consumer may implement only the methods corresponding to the data types they use. In contrast, the intermediary will have to implement the methods corresponding to all the data types, the effort for which far outweighs the value if the intermediary's task is only to forward data. Therefore, it was required to have some common data type, which can be made transparent to those that do not need to interpret the data.

This data type is the *parcel* structure, which is an XML document, and potentially an XML Schema Complex Type.

Tag	Description
Type	URN that uniquely identifies the type of the parcel. Application-level parcel handlers will recognize the type of the parcel, and unwrap it. For example, JPEG might be a parcel type.
ID	Handle for this parcel, given as a URI.
Location	Indicates the location of the parcel data. If the data is contained within the parcel, this field would be *inline*.
Encoding	Indicates the encoding of the data. Intermediaries use this to know how the data must be handled. Binary is one encoding.
Body	The actual parcel data, if the location is *inline*.

Table 1. Parcel data structure tags.

```
<Parcel>
<ID>http://<consumer-ip>/<consumer-port>/2005/02/25/0001</ID>
<Type>http://www.cs.indiana.edu/2004/register</Type>
<Descriptor>
<XMLParser>libxml</XMLParser>
</Descriptor>
<Location>inline</Location>
<Encoding>XML</Encoding>
<Body>
<Time>2005-02-25T11:31:22Z</Time>
<Consumer>
<Host>tiger.cs.indiana.edu</Host>
<Port>2000</Port>
</Consumer>
<DataInterval>5</DataInterval>
</Body>
</Parcel>
```

Table 2. Example of a parcel *control message* registering a consumer to receive data every five seconds.

```
<Parcel>
<ID>http://<sensor-ip>/<sensor-port>/2005/02/25/0991</ID>
<Type>http://www.cs.indiana.edu/2004/temperature</Type>
<Descriptor>
<XMLParser>libxml</XMLParser>
</Descriptor>
<Location>inline</Location>
<Encoding>XML</Encoding>
<Body>
<Time>2005-02-25T17:24:30Z</Time>
<Temperature>23.453</Temperature>
</Body>
</Parcel>
```

Table 3. Example of a *data message* containing temperature information.

Parcels

A Parcel may contain the elements shown in Table 1 above. All the fields except for location and body (if location is *inline*) are optional. Tables 2 and 3 above show example parcels for control and data messages. The consumer would send the control message to the sensor and the sensor would return the data message.

Sensor ontology and the instrument description

A key objective of the CIMA approach is to make the instrument or sensor self describing and to push the production of metadata about what the instrument is producing as far toward the instrument as possible. The former objective, self description, assists components downstream in the data acquisition and reduction process to understand and manage the instrument or sensor effectively (e.g., apply appropriate conversions and calibrations). The latter objective, annotating the data coming from an instrument, provides information needed for proper curation of the data. The development of these components is based on a "CIMA ontology" for instruments and sensors, which is based on the OWL Description Logic formalism. OWL-DL was chosen for several reasons: it makes the description amenable to machine reasoning tasks, it facilitates distributed development and extension of the CIMA ontology and, through inferencing, makes it possible to check the consistency of the ontology even across multiple developers and sites. In addition, XML Schema products such as SensorML (http://vast.nsstc.uah.edu/SensorML/) and ISO schema for location (ISO-19115) and time (ISO-19108) can be leveraged as XML Schema data types from within the RDF specification of the ontology, *i.e.*, instances of the CIMA ontology can refer to resources that are XML documents based on these Schemata or can use types from XML Schemata to type RDF resources.

The CIMA descriptive instrument model consists of several levels as illustrated below in Figure 4. At the outer level is the observatory, the location of one or more instruments with related functionality. An example of an observatory is a crystallography bay containing a goniostat, a CCD array, and several temperature probes used to collect data for an X-ray diffraction crystallography

experiment. Within the observatory are one or more instruments. Within the instrument there may be several sensors which provide observations of measurable quantities and actuators which control the instrument. In the example, the individual thermocouples and hygrometers are sensors. The goniostat positioning system is an example of an actuator. The CIMA ontology in OWL-DL consists of classes for the observatory and its components and for attributes of these components, such as physical and network location, service characteristics, observables produced, actuators, and the sensors' and actuators' response models.

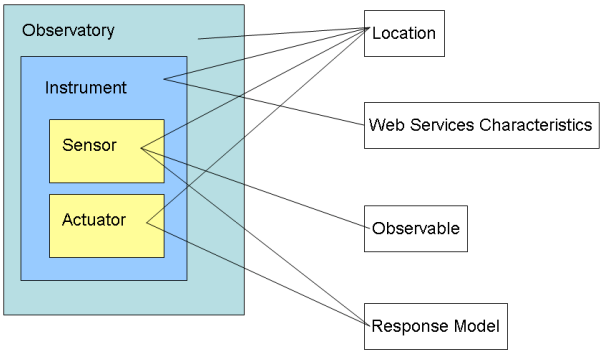

Figure 4. Overview of the CIMA descriptive instrument ontology model.

Implementation

The major code artifacts of a CIMA implementation are *plug-ins*, *modules*, and *channels* (discussed above). Plug-ins map specific hardware (sensors and actuators) to the communications channel and the grid service for the instrument. Modules are components of a layered abstraction for partitioning functionality within a CIMA implementation between communication and interacting with sensors and actuators.

Plug-ins

Plug-ins are implemented using polymorphism. Source and Sink are base classes having the virtual methods *read()* and *receive()*, respectively (Figure 5 below). These methods need to be overridden by the plug-in developer such that they behave appropriately according to the data involved. For example, in a plug-in developed for a Labjack [17] board (LabjackSource class), *read()* constructs and returns a parcel from the data read from the sensor connected to the Labjack

board. A corresponding plug-in is required at the receiver end to extract the data. Once the plug-ins are developed, writing sensor and consumer programs is straightforward.

If an intermediary is only involved in parcel forwarding, a plug-in is not required, since the base class methods provide the required functionality. However, if the intermediary must perform some processing of the data before forwarding, such as inserting the time at which the parcel was received, a plug-in with the required functionality is required.

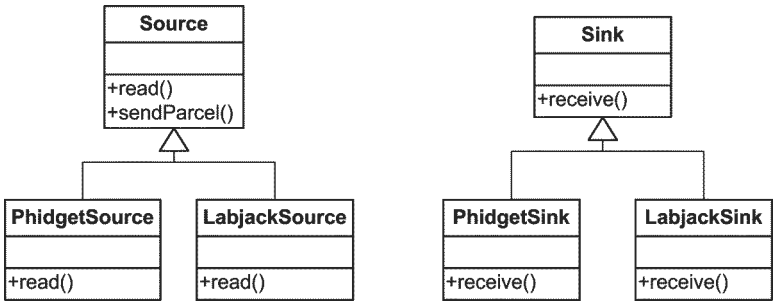

Figure 5. Source and Sink classes.

An in-memory table of consumer information, such as URL and port, is maintained by the channel at the sensor end. This is used in the push mode of operation. Entries are inserted into the table when consumers register with the sensor and removed when they un-register.

Modules

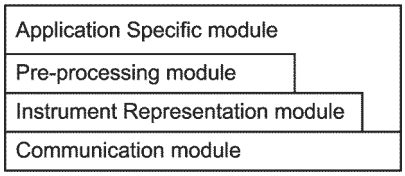

Figure 6. Categorizing modules into layers.

In order to make components as generic and reusable as possible, we separate the functionality among several modules. Modules are categorized into different layers as shown in Figure 6. The *Communication* module mainly handles receiving requests for sensor data and maintaining a list of interested (registered) consumers. It also performs encoding and decoding of messages. The functionality is independent of the application domain and the data being sent.

The *Instrument Representation* (IR) module provides the functions to represent an instrument to potential consumer/drivers and an interface for the other modules to interact with the

physical sensor. For example, an IR for a temperature sensor would have an API to get the current temperature. Likewise if an IR represents a camera, it would provide an API to get a URI for the current image or allow a consumer to register to receive a video stream from the camera.

In most cases, the raw data read from a sensor would need to be modified to some other value or format before the IR can provide it to a consumer. This is handled by the *Pre-processing* module. For example, if the IR for the temperature sensor returns the Kelvin figure, it may need to be converted to the corresponding Celsius value before sending. In the case of an IR for binary data, the binary file may need to be Base64 encoded. [18]

The *Application Specific* module is the least generic of the modules. On the instrument side it would drive a plug-in to acquire data. For a consumer application it provides functionality such as storing the received data and meta-data onto a database, saving binary files at a given location, deciding on which ports to run a grid service on a given machine, etc.

Implementation Examples

X-Ray Crystallography

One of the primary test applications for the CIMA approach to making instruments remotely accessible is X-Ray diffraction crystallography. In this process a small (less than 1mm) crystal is placed in an X-Ray beam and the 3-D diffraction pattern caused by the crystal's lattice is recorded as a series of images captured on a CCD device. Although the CCD is the primary sensor, a number of other environmental variables including the crystal temperature, X-Ray source cooling status, and ambient temperature and humidity are also relevant. In addition to sensors, the experiment depends on an actuator: the crystal is moved precisely in the X-Ray beam by the goniometer, a three degree of freedom positioning system. The entire ensemble of sensors and actuators can be considered as a hierarchical instrument containing a detector, positioning system and environmental controls.

In CIMA-enabled crystallography the detectors and sensors are accessed by an Instrument Representation (IR) module running on a proxy to the data acquisition system. A consumer

application, in this case a data manager that stores and organizes the output of the instrument, registers with the IR running on the proxy to receive CCD images and the corresponding environmental sensor readings. The IR will then start sending the required information at the requested rate to the requested endpoint on the data manager. In this example the data are stored in a location independent storage manager and are made accessible to users from a browser via a web portal. Other functions such as experiment set-up and monitoring and access control are handled through the portal. Any number of consumer/driver applications in addition to the data manager could be active, allowing the primary user to route frames to analysis packages or share the state of the experiment with additional researchers at other locations.

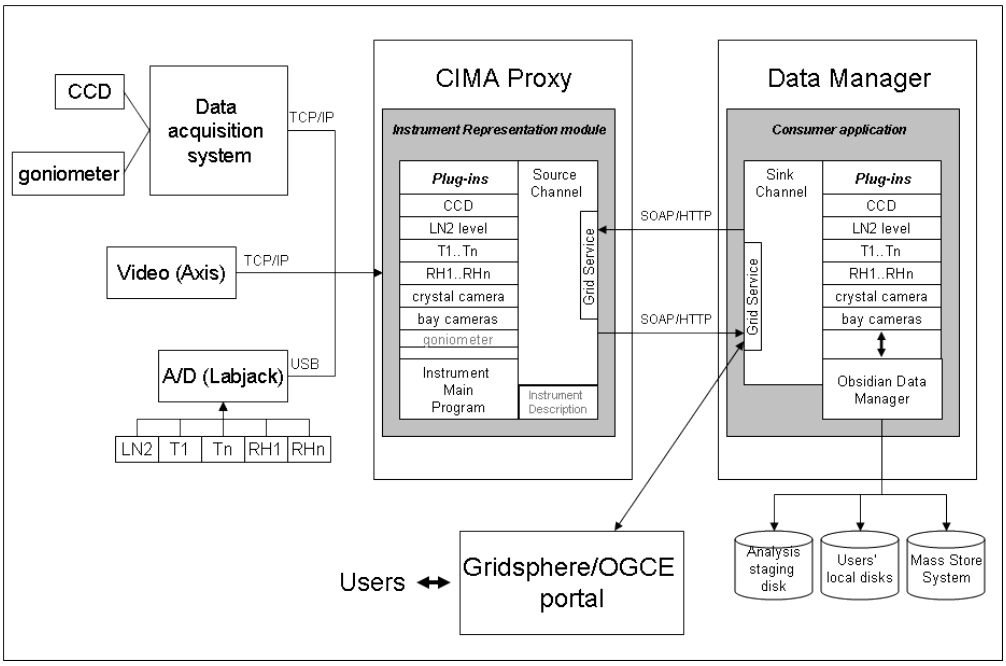

Figure 7. CIMA-enabled X-Ray diffraction crystallography facility.

Robotic telescope

We are also developing CIMA applications in robotic telescopes at the IU Morgan-Monroe State Forest Observatory (MMSFO), a facility that has been collecting long term photometric data on variable stars for nearly 15 years. The operation is completely automated and unattended. Several environmental measurement instruments play a critical role in deciding whether or not to open the dome and use the telescope: wind speed, humidity, precipitation, cloudiness, temperature, and location of the moon all are collected and used. Instrument sensors also are used to detect the

state of the CCD camera that gathers images, the level of liquid nitrogen, the distance above the horizon (to minimize the amount of intervening atmosphere), and other status information.

Once the decision is made to open the dome and perform observations, a list of about 200 target stars is cycled through. Targets close to the zenith are weighted preferentially, provided they make a sufficiently large angle with the moon. The cover to the telescope is opened and then a 4 minute exposure is made, centered on the target star. With the current equipment (optics and CCD) about 12 observations can potentially be made each hour. After targeting and acquisition the CCD image data moves through a sequence of computers and image processing programs, eventually ending in a FITS format file. At this point the specific star or stars of interest in the image are identified and their magnitudes accumulated in a light curve file for each.

The MMSFO site has provided a valuable verification and validation about the software design chosen in CIMA. The same code used in the crystallography application has been used for the instrument representative (WS interface) and the data collector, with changes needed only in the back-end schema to store the data into a relational database. WS interfaces to the CCD image acquisition system are currently in use and interfaces to environmental variables will be available in the near future. This has given us confidence that the CIMA partitioning of instruments and sensors is capable of spanning a wide range of instrument types, and uses for the scientific data.

One of the immediate advantages of making the telescope's operational condition and output available in real time through Web Services is that the usable data produced by the telescope can be increased by a factor of 70 to 80. For the life of the observatory the environmental data was used to make a go/no-go decision on making an observation but was then discarded; directing that data to a data management system will allow determining the quality of any given observation, such as the amount of haziness, the difference in temperature between the telescope mirror and the outside, the nearness of the moon to the target star, *etc.*

More importantly, each CCD frame has 75 to 200 stars other than the target star. That data has not been used because of the difficulty of assuring that the frame is oriented correctly to

unambiguously identify those other stars, a problem which can now be solved with readily-available open source software. For example, telescope output can be used to help locate new variable stars and quasars.

Summary

The wide usage of instruments and sensors has created a need for re-usable middleware architectures, with the ability to represent and access instruments and sensors as grid services. In this paper we discuss the Common Instrument Middleware Architecture (CIMA), which provides a framework and a software architecture for providing remote access and distributed operation to instruments and sensors. CIMA Instrument Representation (IR) modules with grid service interfaces have been developed for X-Ray diffraction crystallography facilities and remote robotic telescopes.

References

1. "GriPhyN - Grid Physics Network," 2003, http://www.griphyn.org/index.php.

2. P. Avery and I. Foster, "The International Virtual Data Grid Laboratory (iVDGL)," 2003, http://www.ivdgl.org.

3. Chinnici, R., *et al.* "Web Services Description Language (WSDL) Version 1.2," World Wide Web Consortium, 2002, http://www.w3.org/TR/wsdl12.

4. Hapner, M. *et al.* "Java Message Service", Version 1.1 April 12, 2002. http://java.sun.com/products/jms/.

5. Fox, G., Pallickara, S. "NaradaBrokering: An Event Based Infrastructure for Building Scaleable Durable Peer-to-Peer Grids." Chapter 22of Grid Computing: Making the Global Infrastructure a Reality Grid. Published by John Wiley, West Sussex, England. 2003.

6. "World Wide Web Consortium," 2003, http://www.w3c.org.

7. Curbera, F., Goland, Y., Klein, J., Leymann, F., Roller, D., Thatte, S., and Weerawarana, S. "Business Process Execution Language for Web Services, Version 1.0," IBM, 2003, http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

8. Foster, I., Kesselman, C., Nick, J., and Tuecke, S. "The Physiology of the Grid: An Open Grid

Services Architecture for Distributed Systems Integration (Draft),” 2002. http://www.globus.org/research/papers/ogsa.pdf.

9. Tuecke, S., *et al.* “Open Grid Services Infrastructure (OGSI) (draft),” Global Grid Forum, 2003, http://www.gridforum.org/ogsi-wg/drafts/draft-ggf-ogsi-gridservice-23_2003-02-17.pdf.
10. Czajkowski, K., *et al.* “The WS-Resource Framework”, Version 1.0, 03/05/2004. http://www.globus.org/wsrf/specs/ws-wsrf.pdf
11. Fielding, R. “Architectural Styles and the Design of Network-based Software Architectures,” Dissertation, University of California at Irvine, 2000.
12. Chiu, K., Govindaraju, M., and Bramley, R. “Investigating the Limits of SOAP Performance for Scientific Computing,” presented at Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC'02), 2002.
13. W3C, “Simple Object Access Protocol (SOAP) 1.1”, 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.
14. “gSOAP”, http://www.cs.fsu.edu/~engelen/soap.html.
15. “Antelope”, http://www.brtt.com/.
16. W3C, “XML Information Set (Second Edition)”, 2003. http://www.w3.org/TR/xml-infoset/.
17. Labjack. http://www.labjack.com.
18. Josefsson, S., Ed. “The Base16, Base32, and Base64 Data Encodings”, 2003. http://www.ietf.org/rfc/rfc3548.txt.
19. Guptay, A., Ludäscher, B., and Martonez, M. “Registering Scientific Information Sources for Semantic Mediation.” In *Proc. 21st International Conference on Conceptual Modeling*. Tampere, Finland.
20. Barros, A., Dumas, M., and ter Hofstede, A. “Service Interaction Patterns: Towards a Reference Framework for Service-based Business Process Interconnection.” Technical Report FIT-TR-2005-02, Faculty of Information Technology, Queensland University of Technology, Brisbane, Australia, March 2005.