The Common Instrument Middleware Architecture: Overview of Goals and Implementation

Indiana University Computer Science Technical Report No.TR616

Tharaka Devadithya
Computer Science Department
Indiana University
Bloomington, IN
Email: tdevadit@cs.indiana.edu

Kenneth Chiu
Department of Computer Science
State University of New York (SUNY)
Binghamton, NY
Email: kchiu@cs.binghamton.edu

Donald F. McMullen
Pervasive Technology Laboratories
Indiana University
Bloomington, IN
Email: mcmullen@indiana.edu

***Abstract*— Instruments and sensors and their accompanying actuators are essential to the conduct of scientific research. In many cases they provide observations in electronic format and can be connected to computer networks with varying degrees of remote interactivity. These devices vary in their architectures and type of data they capture and may generate data at various rates. In this paper we present an overview of the design goals and initial implementation of the Common Instrument Middleware Architecture (CIMA), a framework for making instruments and sensors network accessible in a standards-based, uniform way, and for for interacting remotely with instruments and the data they produce. Some of the issues CIMA addresses include: flexibility in 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.** [1]

I. Introduction

Grid computing [1] is proving to be a useful paradigm for organizing and harnessing distributed resources. By provisioning the fruits of fundamental computer science research as services for such as scheduling and authentication, it is bringing distributed computing into the everyday lives of working scientists. As the convergence between web services and grid computing continues in standards such as the WS-Resource Framework, we expect to see this trend continue, aided by the complementary nature of web services and grid computing.

With this goal, current grid computing research [2][3] understandably focuses primarily on the marshalling of computation and data, and their integration at loci of analysis and synthesis. This focus has spawned the notion of computation and data Grids, respectively. Less well investigated, however, are the sources themselves of data, such as scientific instruments and sensors. These are still largely off-line to downstream grid components, and are poorly integrated as grid entities. This is acceptable if data is viewed as a static resource after whose archival only does scientific activity begin.

In reality, however, the collection of data is as much a part of the scientific process as its analysis. Data collection is not a rote procedure, and often interacts profoundly with interpretation and analysis, whether by human or machine. Ignoring this interaction can lead to inefficient use of computational and human resources, and limits the development of new cyberinfrastructure techniques such as Dynamic Data-Driven Application Simulations (DDDAS) [4], autonomic computing, and software agents.The disadvantages of keeping instruments off the grid are further exacerbated by three trends in scientific research: (1) increasing investments in geographically extended, international collaborations organized around large shared instrument resources, (2) increasing real-time use of instruments by remote researchers both for first-look activities and pipelined data handling, and (3) increasing deployments of large-scale sensor networks.

We thus see a need for bringing instruments[2] on the grid as first-class members, and the Instrument Middleware Project seeks to facilitate this task by researching and developing a set of standards and software components. Together these will form the Common Instrument Middleware Architecture (CIMA), which can be then used to grid-enable a variety of instruments, ranging from large shared resources to tiny wireless controllers such as the Berkeley Mote sensor package [5][6][7], as well as embedded PC-104 and VME-based controller systems.

By promulgating a common set of concepts and interfaces, we hope to increase interoperability between instruments and software. This interoperability will extend along a number of different axes. For example, data analysis software can be insulated from different versions of functionally similar instruments, thereby increasing the flexibility and durability of instrument software. Common interfaces will also promote interdisciplinary collaboration by facilitating compatibility between applications and instruments developed by different communities. A common instrument middleware will also

[1]This work is partially supported by NSF SCI-0330568.

[2]We consider each instrument to consist of a set of sensors arranged hierarchically and physically grouped together on a platform with spatial location and orientation. However, we will make references to instruments and sensors interchangeably.

extend the accessibility of instruments to new classes of users, such as high schools and minority-serving institutions.

CIMA is based on the emerging Open Grid Services Architecture (OGSA) [8] being developed by the Global Grid Forum (GGF) [9]. OGSA includes the Open Grid Services Infrastructure (OGSI) [8], which is used to define a common interface to all OGSA Grid services, and the OGSA Grid Data Service Specification [10], which defines an interface to access data. OGSI uses the web Service Definition Language (WSDL) [11] to specify interfaces to services.

Currently CIMA is implemented in support of an X-ray crystallography application at the Indiana University Molecular Structure Center (IUMSC) and several other crystallographic laboratories. CIMA interfaces to a robotic optical telescope are also being developed for the observatory at the Morgan-Monroe Station located at Morgan-Monroe State Forest (MMSF).

The paper is organized as follows. Section II gives the design goals of CIMA, while in section III the approach taken to achieve these goals are described. Sections IV and V present the architecture and implementation, respectively. A case study is given in section VI and future work on CIMA is proposed in section VII. Section VIII briefly describes the related work while section IX concludes with a summary of the work.

II. Design Goals

We intend CIMA to be usable in a wide variety of scientific scenarios, across a wide variety of instruments and sensors. The goals were derived from considering a representative subset of plausible scenarios.

Significant to the design of CIMA is the difference between the two similar, but distinct scenarios: *remote access* and *distributed operation*. Remote access allows a scientist working off-site to access the instrument. Full support for remote access would allow such a scientist to perform all the tasks that she could perform if she were on-site. Fundamentally, however, the instrument is still administratively, institutionally, and technologically conventional; with all operational aspects such as control, data, and analysis primarily handled at a centralized location.

Distributed operation, on the other hand, is a more profound development in scientific instruments. In distributed operation, the functions of the conventional instrument site itself are distributed within a virtual organization that may consist of the scientist's institution, a minimal instrument site, and third parties such as a data warehousing site. This has the potential to significantly change the way instruments are developed and encourage innovation, by essentially allowing many researchers to simultaneously investigate new software tools and infrastructure. By applying the appropriate grid technologies, each of these researchers can essentially interact with the instrument as if she 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.

Boot-strappable. 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 is very useful in collaborations, where one research group needs access to *grid-enabled* instruments maintained by another group through the architecture. 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 was achieved by using a *parcel*, which is a XML document of the data and meta data involved. More details of the "parcel" can be found in Section IV-E.

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. Sensors may need to be deployed at locations subject to electrical and processing power constraints. For example, a seismic sensor located underwater in deep sea will have all these constraints in addition to bandwidth limitations. While it is unlikely that a computer will be associated with each of such sensors, computers should be situated as closely as possible. The computer located in such a remote area may have limited processing power and memory. Therefore, CIMA implementations should require a minimum of computing, storage, and network resources. Although the usual limiting resource is power, network bandwidth constraints or intermittent connectivity may create secondary requirements for additional short term or persistent storage at the sensor. This creates tradeoffs in memory allocation between data buffers and program address space.

Support for intermediaries. Intermediaries are important for signal processing or buffering functions between a sensor and the consumer. This off-loads the work of serving multiple consumers from the sensor to an intermediary. The intermediary can have one input stream from the sensor and multiple output streams for the consumers. Intermediary also can act as security gateways for the sensor node by allowing only particular intermediaries to connect to it. Also, filters can be implemented at the intermediate nodes.

III. Approach

We have taken several approaches to meet the goals. These are outlined in the following sub sections.

A. Layered Specification

To provide reusability and interoperability of instrument interfaces, we strive for layered specifications. For example, a lower-layer specification corresponding to a pressure sensor should be reusable with that corresponding to a temperature sensor, with minimal modifications, for example. Then an application written for one sensor would have a fair degree of functionality (i.e., require minimal code changes) even with another sensor. The ultimate goal of this approach is to promote the reuse of code components between applications.

B. Plug-Ins

While the processing required for sending and receiving data is consistent among different type of instruments, different instruments and sensors may require different code to read; and the construction and interpretation of data messages would be specific to each of them. We use plug-ins to perform these specific functions. Therefore, a plug-in would be required at the data originating point as well as at the final destination.

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, would need to have a plug-in to perform such tasks.

C. Loose Coupling

Loose-coupling encourages interoperability by minimizing the dependencies between the system components, such as sensor, data consumer, and intermediaries. This loose-coupling is achieved by implementing a document-oriented message passing model. Each message, whether data or control, would be an XML document containing the data along with some metadata required to interpret them.

D. Ontology

One 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.

E. Hierarchical

Instruments are hierarchical in nature. An example would be a crystallography application consisting of a positioning system (goniostat), a CCD array for imaging, and a cryo-cooling unit to preserve the crystal being studied. The entire ensemble can be considered as a hierarchical instrument containing a detector, positioning system and environmental controls. Another example from optical astronomy would be to partition the observatory into dome controls, environmental conditions, telescope positioning, optics selection, and imaging detectors.

Client applications are simplified if they can access one stream of data as opposed to multiple streams. This could 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 plug-ins at each parent instrument. The plug-in aggregates data from its child instruments, re-sending the composite as if all the data is being generated by the parent.

F. Push and Pull Models

Data messages may be "pulled" on demand at the cost of a request-response cycle, or they can be directly "pushed" when scheduled (or as available) from the sensor to the receiver using one-way messages. Both models are useful, depending on the application requirements, with the pull model usually being more convenient, but the push model usually being more efficient. Since the push model does not require a request-response, multiple messages can be batched into single system call, for example. CIMA supports both models.

In the push model, the consumer maintain some type of end-point to which the sensor can stream-in data. Since we are grid-enabling sensors, the most natural choice for this endpoint would be a grid service endpoint.

However, if the consumer is only interested in receiving a single value (current value) from the sensor (e.g,. current temperature), then a pull-model would be more suitable.

IV. Architecture

A. Instrument Model

Our current instrument model is shown in Figure 1. An instrument consists of one or more sensors. Each sensor may serve zero or more consumers. A consumer can receive data from one or more sensors. The communication between a sensor and a consumer forms a virtual link, which we call a *channel*. We are currently also designing a set of default ports.

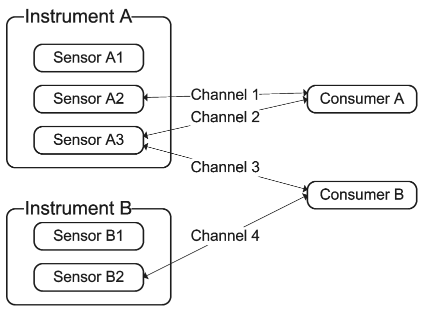

Fig. 1. Instrument Model

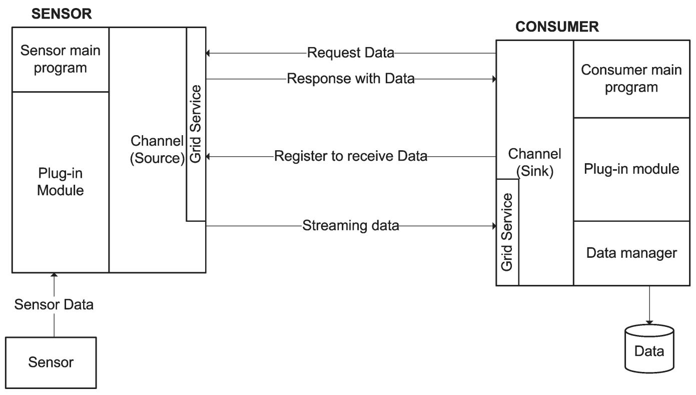

Fig. 2. Communication via the Channel

B. Channels

The application consists of *channels* and *plug-in modules*. A channel provides a generic framework for the communication while the plug-ins implement application-specific functionalities.

As shown in Figure 2, the channel handles all the communications between the sensor and the data consumer. Specific plug-in modules, both at the sensor end and the consumer end, implement the sensor-specific behavior.

The Channel has two modes of operation, namely the Source (sensor end) and the Sink (consumer end). Each mode runs a grid service instance for receiving messages (control and data) from the other. The Channel Source's grid service mainly handles control information from the data consumer, such as registering with the sensor to receive sensor data and unregistering to stop receiving data. It also responds to one-time requests for sensor data. The grid service at the Channel Sink receives streaming data and status messages such as "sensor data not available" from the sensor.

A data consumer can chose 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.

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 methods is used when the rate at which data being generated is not known.

C. Communication Protocols

We have used SOAP [12], since it is the most widely accepted standard for web Services. The current implementation of the channel uses gSOAP [13] to handle the serialization/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 uses Antelope [14] 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 [15] 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.

D. Data Structures

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 a 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. This would enable the Application Programmers to use the appropriate methods 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.

Another problem with this approach surfaces if there is an intermediary between the sensor and the end consumer, as shown in Figure 3. A given sensor or a consumer may implement only the methods corresponding to the data types

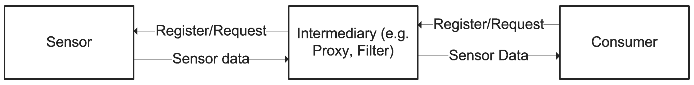

Fig. 3. Intermediary between sensor and end consumer

they use. In contrast, the intermediary will have to implement the methods corresponding to all the data types. The effort for this would not be worth it if the intermediary's task is only forwarding 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. We came up with the "Parcel" (section IV-E) structure, which is an XML document, as a solution to this problem.

E. Parcel

The data out of a channel should be presented in a manner such that intermediaries can handle it without specific knowledge of the data. To enable this, we wrap data in an abstraction known as a parcel.

A Parcel is an XML document, which contains data and meta data about a message being sent. It may contain control information or data.

A Parcel may contain the following elements:

- **Type** is a 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** is 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** is the actual parcel data, if the location is *inline*.

All the fields except for location and body (if location is *inline*) are optional.

Following is an example of a *control message*, where a consumer registers with a sensor to receive streaming data every 5 seconds.

```
<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>
```

Following is an example of a *data message*, containing temperature information.

```
<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>
```

F. Sensor Ontology

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. 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 approaches and 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 datatypes 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 instrument model consists of several levels as illustrated 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. Instruments are devices designed to provide a specific set of functionality, such as the goniostat or CCD in the example. Within the instrument there may be several sensors which provide observations of measurable quantities and actuators which control

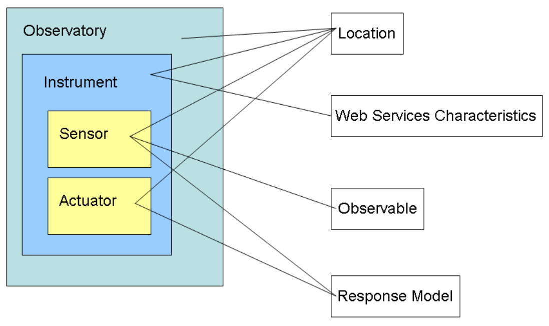

Fig. 4. Overview of the CIMA instrument ontology model

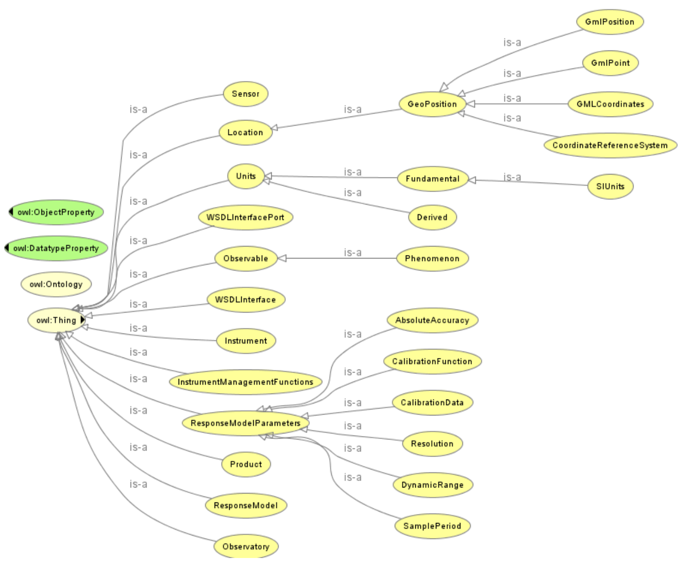

Fig. 5. Some of the OWL classes used to describe a CIMA instrument

the instrument. In the example, the individual thermocouples and hygrometers are sensors. The goniostat positioning system is an example of an actuator.

The class structure of a portion of the ontology is shown in Figure 5. In addition to the Observatory, Instrument, Sensor and Actuator classes there are others to describe the nature of the sensor's location, observables, their units and the details of how a sensor is accessed. Each sensor and actuator has an associated response model that provides information about the accuracy, dynamic range, resolution, and calibration information. In the current model each instrument is represented through a web Services interface with associated WSDL. The instrument's WSDL along with details of ports and operations is available as instances of the WSDLInterface and WSDLInterfacePort classes in the ontology.

Section VI-B illustrates how a thermocouple might be represented.

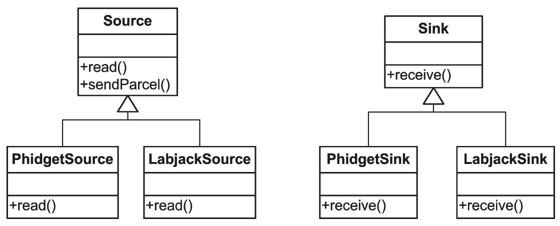

Fig. 6. Source and Sink classes

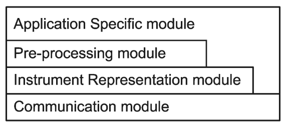

Fig. 7. Categorizing modules into layers

V. Implementation

A. Plug-ins

Plug-ins are implemented using polymorphism. Source and Sink are base classes having the virtual methods *read()* and *receive()*, respectively (Figure 6). 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 [16] 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 would be a trivial task.

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.

B. Streaming Mode

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 push mode of operation. Entries are inserted into the table when consumers register with the sensor and removed when they un-register.

C. Modules

In order to make components as generic and reusable as possible, we separated the functionality among several modules. The modules were categorized in to different layers as shown in Figure 7.

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*, as the name suggests, provides the functionalities to represent an instrument. It provides 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. Another example is if the IR represents a camera, it would provide an API to get the path name of the current image file.

In most cases, the raw data read from the IR would need to be modified to some other value or format. 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 it over. In the case of an IR for binary data, the binary file may need to be Base64 encoded [17].

The *Application Specific module* is the least generic one among all the modules. It provides functionalities 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.

VI. Case Studies

A. Crystallography System

The Crystallography system consists of a CCD image detector and several sensors measuring environmental conditions, such as temperature, pressure and humidity. There is also a sensor to monitor the level of liquid nitrogen, which is used for cooling the crystals. All these detectors and sensors are each accessed by an Instrument Representation (IR) in CIMA.

A crystallographer registers with each of the IR's in order to receive CCD images and the corresponding sensor readings. The IR's will then start sending the required information at the requested rate and to the requested endpoint. These are stored in a database at the receiver's end and are made accessible from a browser via a web portal.

B. Thermocouple Example

This section illustrates how a thermocouple might be described in RDF.

Consider the following example, where a thermocouple is used to measure temperature, through a voltage measurement and a calibration curve. This calibration, and hence the measured temperature, may be derived from standard tables for the type of thermocouple used, or through a carefully performed manual calibration against a primary standard. The description of such a thermocouple in RDF might look something like the following:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!-- Default namespace is "http://cima.org/" -->
<rdf:RDF
 xmlns="http://cima.org/"
 xmlns:rdf="&rdf;"
 xmlns:rdfs="&rdfs;">
  <Description
   my_key="CIMA:A7DB-2287-E7DB-6745"
   my_name="TC7443"/>
  <WSDL
   uri="http://kaplab.iu.edu/TC_service.wsdl"
   uri_namespace="http://kaplab.iu.edu
                /ns/TC_service">
    <binding_template xml:space='preserve'>
      <!-- CDATA BLOCK OF WSDL -->
    </binding_template>
  </WSDL>
  <Characteristics
   idesc="K-type thermocouple on 16 bit A/D"/>
  <Calibration
   Port_name="thermocouple_reading"
   type="vector">
    <value>
      30, 1.179, 35, 1.397, 40, 1.606,
      45, 1.813, 50, 2.020, 55, 2.225
    </value>
  </Calibration>
  <port
   data_format="u16"
   name="thermocouple_reading"
   signal="voltage"
   port_direction="OUT"
   port_type="INTEGER"/>
</rdf:RDF>
```

The schema consists of four main parts: a description that identifies the specific instrument platform, information about how to use the service (a WSDL document for the service or a URI pointer to it), calibration information for this instrument, and the characteristics of the data produced by each channel. The latter may seem redundant to the WSDL document but it provides a place to put semantic information about the instrument's control and data channels and additional type information which may be necessary if WSDL types are too opaque for application code to parse.

After the application has queried the instrument for a description, it can parse the description to extract an operational model of the underlying instrument and information about how to interact with the instrument's service ports. In this example the application finds that there is one data (OUT) port that provides a voltage signal as a 16 bit unsigned integer. Furthermore the application can consult the CIMA RDF Schema to determine what voltage means and how best to handle this data in a computational or user interface context.

VII. Future Work

A main concern in any distributed systems would be security. The data would need to be protected from unauthorized access if they involve sensitive research data or if there are privacy issues, such as in the case of a series of images of a laboratory environment.

At a minimum, transport level security will be provided using Secure Socket Layers [18]. The consumers would need to sign their messages requesting for sensor data. The sensor would also sign the messages with data, to ensure the authenticity of the sender, as well as the integrity of messages.

Another additional feature would be filtering data depending on some criteria. The criteria will be based on the data values, the sender, and/or the recipient. For example, a consumer may be interested in receiving temperature values only if the values are greater than 100 F.

Filter conditions can be specified and processed using XPath [19]. Since the data messages are XML documents, an XPath processor can easily filter the messages given the query.

VIII. Related Work

Distributed real-time systems using TAO CORBA [20]. The TAO ORB is a leading effort to incorporate real-time functionality into CORBA, especially to support avionics applications. As such TAO emphasizes bounded (low) latency communications and fault tolerance. Although CORBA is a general specification, many implementation-specific details make interoperability difficult. We believe that web services provide a more open and interoperable basis for building distributed computing systems, and that performance issues can be addressed through binary XML standards.

Architecture for Accessing Data Streams on the Grid [21]. Plale has developed a flexible architecture for real-time access to streaming data. She develops a taxonomy for data streams which can be used to determine when a data stream system can be characterized as a data resource accessible through a Grid Data Service. She then realizes such a service through the dQUOB [22] real-time query system. We envision that CIMA can be used to shield such a system from the peculiarities of individual instruments.

EPICS [23]. The Experimental Physics and Industrial Control System (EPICS), developed at the Accelerator Technology (AT-8) group at Los Alamos National Lab) and the Advanced Photon Source (APS) at Argonne National Lab, consists of an architecture for building scalable control systems and a collection of code and documentation comprising a software toolkit. EPICS is based on the idea of virtual channels between acquisition code and the underlying hardware. Although well designed for high data rate applications its complexity has limited use outside of accelerator facilities. A preliminary mapping of EPICS process variables to a CIMA interface has been designed.

Astronomical Instrument Markup Language (AIML) [24]. AIML is a NASA project to create an XML DTD for the HAWC airborne camera and related systems [25]. The aim was to create a representation of the control and data systems primarily as a specifications document to coordinate work between hardware and software engineering groups. AIML was also used to develop simulations of the hardware and to generate user interfaces. The vocabulary used in the AIML DTD was drawn primarily from the hardware engineering effort and included a large percentage of project-specific terminology, but the project has laid some useful foundations for developing general ontologies for instruments.

Universal Plug and Play (UPnP) [26][27]. UPnP is a Microsoft standard to allow devices to interact over an IP network using a zero configuration approach. Basics include using DHCP to acquire an address, a network registry scheme for service discovery based on pre-assigned device codes, and an on-line documentation system to allow users to map device codes to capabilities. The Salutation Consortium [28] is similar effort by Japanese gadget makers and NIST.

IX. Summary

The wide usage of instruments and sensors has spawned the need for re-usable middleware architectures, with the ability to connect instruments with minimal configurations. In this paper we presented the Common Instrument Middleware Architecture (CIMA), which provides a framework for such requirements.

The main objective CIMA was grid-enabling instruments. We have implemented Instrument Representations (IRs) with grid service interfaces. The instruments are self describing with the aid of ontologies.

CIMA is currently implemented in beta testing mode in a Crystallography application at the Indiana University Molecular Structure Center with applications to other instruments and sensors in development.

References

[1] C. Kesselman and I. Foster, *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1999.
[2] "Globus Project," http://www.globus.org.
[3] A. Grimshaw, "Legion," http://www.cs.virginia.edu/~legion.
[4] F. Darema, "Dynamic Data-Drive Application Simulation," http://www.dddas.org.
[5] A. Woo, "Mote Documentation and Development Information," http://www.cs.berkeley.edu/~awoo/smartdust/.
[6] K. S. J. Pister, "SMART DUST: Autonomous sensing and communication in a cubic millimeter," http://robotics.eecs.berkeley.edu/~pister/SmartDust/.
[7] J. M. Kahn, R. H. Katz, and K. S. J. Pister, "Mobile Networking for Smart Dust," in *Proc. of ACM/IEEE Intl. Conf. on Mobile Computing and Networking (MobiCom 99), Seattle, WA*, 1999.
[8] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "The physiology of the grid: An open grid services architecture for distributed systems integration," in *Open Grid Service Infrastructure WG*, 2002.
[9] "Global Grid Forum," http://www.gridforum.org/.
[10] M. Antonioletti, "Grid Data Service Specification," https://forge.gridforum.org/projects/dais-wg/document/Grid_Data_Service_Specification-ggf9/en/1.
[11] E. Christensen et. al., "Web Services Description Language (WSDL) 1.1," http://www.w3.org/TR/wsdl, 2001.
[12] W3C, "Simple Object Access Protocol (SOAP) 1.1," http://www.w3.org/TR/2000/NOTE-SOAP-20000508/, 2000.
[13] "gSOAP," http://www.cs.fsu.edu/~engelen/soap.html.
[14] "Antelope," http://www.brtt.com/.
[15] W3C, "Xml information set (second edition)," http://www.w3.org/TR/xml-infoset/, 2003.
[16] "Labjack," http://www.labjack.com/.
[17] "The Base16, Base32, and Base64 Data Encodings," http://www.ietf.org/rfc/rfc3548.txt, 2003.
[18] "Secure Socker Layers (SSL)," http://wp.netscape.com/eng/ssl3/.
[19] W3C, "XML Path Language (XPath)," http://www.w3.org/TR/1999/REC-xpath-19991116, 1999.
[20] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design of the TAO Real-Time Object Request Broker," *Computer Communications, Elsivier Science, vol. 21*, 1998.
[21] B. Plale, "Architecture for Accessing Data Streams on the Grid," in *2nd European Across Grids Conference (AxGrids)*, 2004.
[22] B. Plale and K. Schwan, "Dynamic querying of streaming data with the dQUOB system," *IEEE Transactions in Parallel and Distributed Systems. 14(4)*, pp. 422–432, April 2003.
[23] S. A. Lewis, "Overview of the Experimental Physics and Industrial Control System: EPICS," http://csg.lbl.gov/EPICS/OverView.pdf, 2000, lawrence Berkeley National Laboratory.
[24] J. Breed, "Astronomical Instrument Markup Language," http://pioneer.gsfc.nasa.gov/public/aiml/, 2000, NASA GSFC.
[25] ——, "High Resolution Airborne Wideband Camera," http://pioneer.gsfc.nasa.gov/public/hawc/, 2000, NASA GSFC.

[26] “Universal Plug and Play Device Architecture,” http://www.upnp.org/download/UPnPDA10_20000613.htm, 2000, microsoft Corporation.
[27] “Understanding Universal Plug and Play: A White Paper,” http://www.upnp.org/download/UPNP_UnderstandingUPNP.doc, 2000, microsoft Corporation.
[28] “The Salutation Consortium,” http://www.salutation.org/, 2003.