Efficient data collection from Open Modeling Interface (OpenMI) components

Tom Bulatewicz, Daniel Andresen
{tombz@ksu.edu, dan@ksu.edu}
Dept. of Computing and Information Sciences, Kansas State University, Manhattan, KS, USA

Abstract— *The management of output data from simulation models can be simplified in grid environments by automating and standardizing the way in which they are collected and stored. In the context of component-based computer models with well-defined input-output interfaces, general-purpose data collector components can be linked to model components to retrieve output data and deliver them to online repositories via web services. We have developed a distributed data collector component that adheres to the Open Modeling Interface (OpenMI). The component buffers data to minimize the impact on a simulation's runtime and shares the buffer across compute nodes for load-balancing and cooperative delivery of data to web services. The buffering capability resulted in minimal runtimes within a single simulation and reduced data delivery latencies for concurrently executing simulations across a cluster. In this paper we report on the design and performance of the component.*

Keywords: OpenMI, data, web services, modeling, simulation

1. Introduction

The output data produced by environmental computer simulations often provides a starting point for investigation into the phenomena being studied. The data may need to be archived, aggregated, processed, and analyzed statistically or geographically before they can be visualized and interpreted. These may be performed by an individual or as part of a collaborative effort between groups within or across institutions.

Model output traditionally takes the form of local data files that may be of a custom format or adhere to simple standards such has comma separated values and extensible markup language or more complex standards such as netCDF and various database formats. Managing output files can be challenging, particularly in a grid environment in which simulations execute on multiple machines across a compute cluster.

As an alternative to data files, model output data can be immediately relayed to an Internet-connected data storage service that serves as a single repository for the data. This facilitates sharing of the data over the Internet and automates common tasks such as data archiving. In the general case, this capability is added to a model program by either modifying the model source code or incorporating intermediary software (often via scripting). In the case of model components with well-defined input-output interfaces, a general-purpose data collector component can be attached to any model component to retrieve the output data. Such components can mitigate data management challenges in any linked modeling context.

The Open Modeling Interface (OpenMI) [1] defines a standard way for software components to exchange data with each other and coordinate their execution. It defines a set of capabilities that a component must possess in order for it to be linkable to other components. These capabilities are both descriptive, to support the task of specifying component interactions at the domain level, and functional, to support the execution of a set of linked components. To fulfill the descriptive requirements, a component must be capable of providing a list (via a function call) of the domain quantities that it can provide and those that it uses as input, along with the units and spatial distribution of each. These are called *output exchange items* and *input exchange items*, and in the case of model components there is typically one output item for each quantity that it simulates and one input item for each of its inputs. To fulfill the functional requirements, a component must possess a *GetValues* function through which it provides data (that correspond to the exchange items) at runtime.

The GetValues function has three parameters that collectively identify a quantity at a single point in time at one or more locations as illustrated in Figure 1. A quantity (labeled Q in the figure) is represented by an object with several properties such as a textual identifier and units information. A time T is represented by a simple date object. A list of locations E is represented by an *elementset* object that contains a collection of *element* objects that each have a textual identifier, spatial shape (point, line, or polygon) and geographic coordinates. The GetValues function returns an array of real numbers called a *valueset* V such that each number corresponds to an element (based on its index) and collectively represents the state of a quantity at a point in time.

The GetValues function not only provides a means for the exchange of data between a group of linked components (called a *composition*) but it also provides a means for their coordinated execution at runtime. A special component

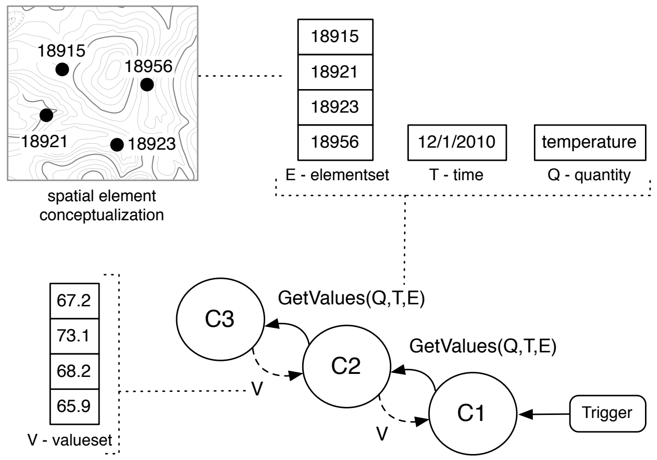

Fig. 1: OpenMI pull-based execution. Solid lines indicate function calls and dashed lines indicate the flow of data.

called a *trigger* begins by calling GetValues on one of the components. When GetValues is called on a component, it executes as many time steps as necessary to advance to the requested point in simulation time and returns a valueset corresponding to that time. Thus a component only executes time steps as-needed to respond to a GetValues call. If it needs input from another component in order to execute a time step it calls GetValues on that component and blocks until a valueset is returned. The components take turns executing synchronously and *pull* data from each other until the simulation completes.

Compositions of linked components can be created and executed using visual software tools. A scientist chooses a set of components, and for each one, assigns each output exchange item to another component's input exchange item. These assignments are called *links* and there may be multiple links between two components and may be in the same or opposite directions.

In this work we present the design and evaluation of a general-purpose Data Collector Component (DCC) that is capable of collecting data from OpenMI components and delivering them to online repositories. We describe the design and implementation of the DCC in the following section and present our experimental results in Section 3. We review related work in Section 4 and present our conclusions in Section 5.

2. Methods

Figure 2 illustrates the movement of data through a distributed data collection system for linked model components. Compositions of linked components execute on cluster nodes. Each composition includes a DCC that collects data from the components and delivers them to web services. Any web service that is capable of accepting data items consisting of a quantity identifier, date, list of location identifiers, and list of values, can be used.

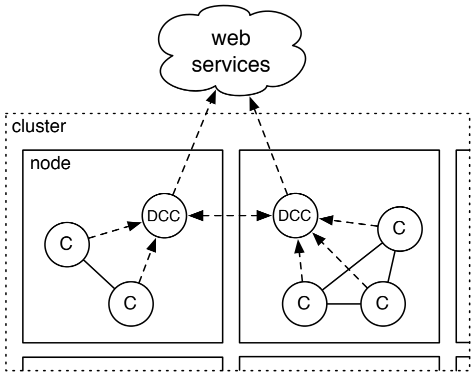

Fig. 2: System overview.

2.1 Web Services

The Open Geospatial Consortium [2] publishes interface standards for location-based information and services to support interoperability. The Geographic Markup Language (GML) [3] standard defines XML schemas for geospatial information including observations data, which is capable of describing model output from OpenMI components. The prototype implementation of the DCC uses this XML schema to represent the data as they are sent to web services. An example of an XML document that conforms to the schema is given in Figure 3 (we used a more succinct gml:id attribute in place of a gml:location element). Additional XML schemas, such as Observations and Measurements [4] could be incorporated into the DCC as well.

```
<gml:Observation gml:id="18951">
  <gml:validTime>
    <gml:TimeInstant>
      <gml:timePosition>2010-12-01T12:00:00</gml:timePosition>
    </gml:TimeInstant>
  </gml:validTime>
  <gml:resultOf>
    <app:Temperature>67.2</app:Temperature>
  </gml:resultOf>
</gml:Observation>
```

Fig. 3: GML description of a single value.

If a DCC were to make a web service call each time it collects a valueset then the execution of the simulation would be paused for the duration of the call due to the synchronous execution of components. In addition, sending a single valueset in each web service call may result in inefficient network utilization when the network latency is comparable to the transmission time of the valueset.

In the ideal case the collection of valuesets would not increase the runtime of a simulation and a sufficient number of valuesets would be transferred in each web service call to achieve efficient network utilization. To these ends the DCC utilizes a buffer that enables the sending of valuesets to be asynchronous with respect to their collection and allows for the coalescing of multiple valuesets into a single web service call. The buffer is shared among all DCC's across a cluster to provide a total buffer size that is greater than the local buffer size of each individual and to allow cooperation in delivering the buffered data.

The implementation of the DCC consists of a buffering module and a delivery module as illustrated in Figure 4. The buffering module collects valuesets from components and stores them in the shared buffer. The delivery module removes valuesets from the buffer and delivers them to web services. The behavior of these modules is dictated by three

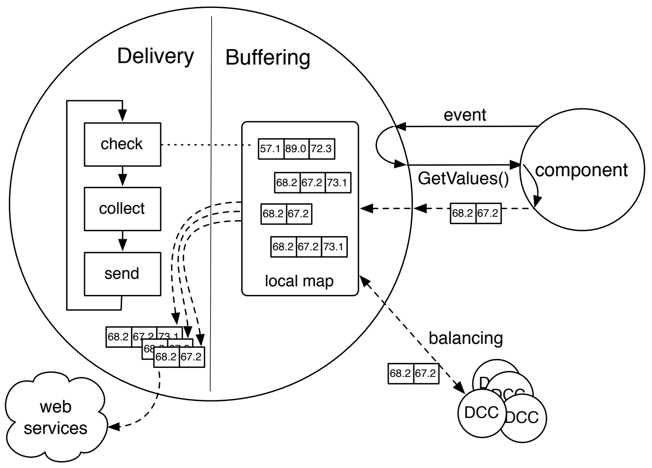

Fig. 4: Operation of the data collector component. Solid lines indicate function calls and dashed lines indicate the flow of data.

parameters: maximum local buffer size, maximum residence time, and minimum delivery size. These are described in the following sections.

2.2 Buffering

A DCC may be linked to one or more components within a composition. Components raise an event each time output data is produced, typically after each time step (the DataChanged event). The DCC listens for this event and in response calls the component's GetValues function to obtain the newly created valueset and places it into the buffer.

The shared buffer is provided by an open source data distribution platform (Hazelcast [5]) that is compiled into, and runs as a set of background threads within, each DCC. It manages a distributed map data structure, dynamically discovering peers via multicast and communicating via TCP/IP. The entries in the map are evenly distributed among all peers and each peer holds a portion of the entries in a local map. Entries are keyed by a universally unique identifier and are variably sized and include the quantity identifier (string), timestamp (long), elementset identifier (string), expiration date (long), valueset data (byte array), and valueset data size (long). The memory overhead of storing an entry in the distributed map data structure is approximately 260 bytes.

The amount of memory dedicated to the local map is dictated by the maximum local buffer size parameter, thus the total memory available to the distributed map is the sum of all peers' local maximums. If the addition of an entry into the buffer would cause the local map's size to become greater that the maximum, then the buffering module waits until there is available space, during which the execution of the simulation is blocked. The buffering module relies on the delivery module to remove entries from the buffer and send them.

2.3 Delivery

The removal of entries from the buffer by the delivery manager is dictated by two parameters: minimum delivery size and maximum residence time. The minimum delivery size provides a means to regulate network efficiency. The delivery manager attempts to remove enough valuesets from the buffer to meet the minimum delivery size before sending them in a single web service call. This may cause entries to remain in the buffer for extended periods of time. This may be acceptable in cases in which the data is being archived, but in cases where the data is consumed as the simulation is being carried out, it may be necessary to place a constraint on the duration that an entry may reside in the buffer before it is delivered. This is controlled by the maximum residence time parameter, which places a limit on the length of time an entry remains in the buffer. Each entry in the buffer has an expiration date that is calculated based on the creation date and maximum residence time. The maximum residence time has higher priority than the minimum delivery size, so in some cases a web service call may contain a small amount of data in order to enforce the time constraint.

The minimum delivery size is specified in terms of the number of values per web service call rather then the number of bytes of serialized XML because different web services may utilize different XML schemas and the latter would require *a priori* knowledge of the serialized XML size for any valueset. Identifying the serialized XML size for a valueset would require either (1) the serialized XML size per value to be known or (2) valuesets to be temporarily serialized to XML as the buffer is being inspected. The former would require calculating the serialized XML size per value for each XML schema and the latter would consume additional system resources.

The following algorithm is used by the delivery manager. The delivery thread periodically iterates over the entries in the local map and determines (1) whether there are any entries that have expired and (2) whether there are enough entries to meet the minimum delivery size. If either case is true, the delivery thread iterates over the local map to identify the entry with the lowest expiration date and removes it. It repeatedly iterates over the entries, removing the entry with the lowest expiration date, until either (1) enough entries have been collected to meet the minimum delivery size, or (2) the local map is empty. Only the entries in the local map are iterated in order to avoid global operations and improve scalability. The valuesets within the entries are deserialized from their byte array representation, coalesced, serialized into XML, and then sent in a single web service call. The process repeats until both the simulation is completed and the number of entries delivered is equal to or greater than the number of entries inserted into the local buffer. The latter ensures that each DCC delivers a fair share of the entries and that only DCC's with excess capacity deliver more entries then they collect.

When a valueset is delivered to a web service it must include information about the location that each value represents. This information is not stored inside the buffer entries because it would be redundant as the location information is identical for all valuesets that correspond to a common elementset. Elementsets are static during a simulation run so there is typically a high ratio of valuesets to elementsets. The buffer entries only store the elementset's identifier and the actual elementset information is stored in a separate distributed map. In this way a DCC can lookup the complete elementset information for any valueset before it is delivered.

3. Experimental Results

We conducted a performance study using an onsite Linux-based Beowulf cluster. The compute nodes had 2 quad-core 2.3 GHz Opteron 2376 processors with 8 GB of memory and the server node had a quad-core 2.7 Ghz processor and 8 GB of memory. All nodes were connected via gigabit ethernet. The software components were implemented in Java using the Alterra OpenMI 1.4 SDK and the web service was SOAP-based and implemented in PHP hosted by the Apache HTTP server within a Windows virtual machine.

To represent a model component we created a *producer* component that used a fixed-length time step of 1 day and would sleep for a fixed amount of time between time steps to mimic the time spent calculating a time step. On each time step a single valueset was generated and collected by the DCC linked to it.

3.1 Minimum Delivery Size

The minimum delivery size that maximizes throughput to the web service is dependent on several factors including network latency, available bandwidth, and software performance. We conducted a series of measurements to empirically identify the ideal delivery size for our experimental configuration.

To establish a baseline of the expected performance of the DCC, we independently measured the maximum throughput from a benchmark Java application to the Apache/PHP web service and found it to be 47 MB/s. This performance was only possible when the ChunkedStreamingMode of the java.net.HttpURLConnection object was enabled, which prevents the complete POST request from being buffered in memory and streams the data directly from disk to the connection's input stream (although not all web servers support this mode).

We configured a single composition consisting of a producer component linked to a DCC and measured the throughput from the DCC to the web service in a series of simulations. In each simulation the producer generated a number of valuesets that were collected by the DCC and individually sent to the web service in separate calls. The average throughput (over all sends in each simulation) achieved in each simulation is shown in Figure 5.

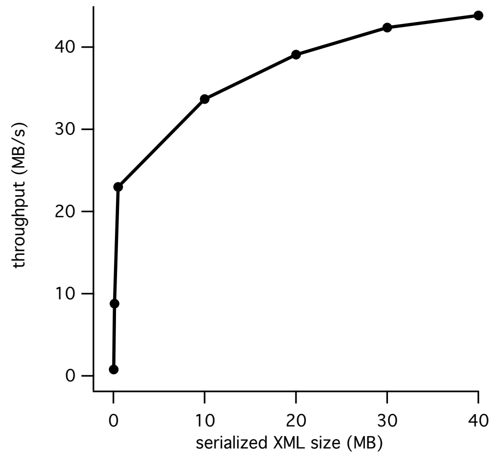

Fig. 5: Impact of minimum delivery size on throughput.

As the data size increased, the throughput increased as well until the maximum throughput was reached, at which point larger data sizes no longer improved the throughput. To achieve at least 50% of the maximum throughput it was necessary to set the minimum delivery size to 2.6×10^5 values which ensures that each web service call contains at least 11 MB of serialized XML data.

3.2 Maximum Residence Time

The maximum residence time imposes a limit on the amount of time that an entry may reside in the buffer. When expired entries are detected in the buffer, all expired entries are removed from the buffer, along with any additional entries necessary to meet the minimum delivery size, and are sent in a single web service call. Thus some entries may be removed from the buffer before they expire resulting in an average residence time that is less than the maximum residence time.

To investigate the effect of the maximum residence time, we measured the average residence time of entries that were added to the buffer at a regular interval. The testing configuration included a single producer component that generated a valueset at a regular interval that was shorter than the maximum residence time in each case causing several entries to be added to the buffer before one of them expires. The buffer was configured such that entries were only removed as a result of an expiration (unlimited maximum buffer size) and when an expiration occurs all entries were removed from the buffer and sent in a single web service call (minimum delivery size set to maximum).

The results indicate that the average residence time was one-half the maximum residence time in all cases. The expiration of the first entry added to the buffer causes all the entries to be removed and delivered, all of which spent differing amounts of time in the buffer. In general, the sum S of the residence time rt of n entries added to the buffer at a fixed interval i is:

$$\begin{aligned} S_n &= rt_1 + (rt_1 + i) + .. + (rt_1 + (n-1)i) \\ &= n/2(rt_1 + rt_n) \end{aligned} \tag{1}$$

The first entry added to the buffer has the maximum residence time RT_{max} and the last one added has 0 residence time, thus the average residence time RT_{avg} is given by:

$$RT_{avg} = (n/2(0 + RT_{max}))/n = RT_{max}/2 \tag{2}$$

With respect to network efficiency, setting a low maximum residence time resulted in inefficient bandwidth usage because the minimum delivery size was not met. In such cases, network efficiency can be improved by increasing the maximum residence time.

3.3 Buffering

The primary purpose of the buffer is to provide a means for the asynchronous delivery of data to minimize the impact of the data collection on the simulation runtime. To evaluate the buffer's utility in this respect we measured the simulation runtime of a single composition consisting of a producer component linked to a DCC with varying buffer sizes.

The producer generated valuesets of 100 KB at a fixed interval and the DCC had an unlimited maximum residence time and a minimum delivery size equivalent to the size of the valueset so that valuesets were delivered to the web service in individual calls. We imposed a latency on the web service of twice the interval so that data was added to the buffer at exactly twice rate at which it was removed causing the amount of buffered data to increase throughout the simulation.

We measured the *simulation time* as the time spent by the producer executing time steps. In general the minimum buffer size BS_{min} to ensure no impact on runtime is given by:

$$BS_{min} = (InRate - OutRate) \times T_{sim} \tag{3}$$

where $InRate$ is the rate at which data is added to the buffer, $OutRate$ is the rate at which data is removed, and T_{sim} is the simulation time. In this experiment the expected minimum buffer size was (20.0 KB/s - 10.0 KB/s) $\times$ 2500 s = 25.0 MB.

Results are given in Figure 6. Given a sufficient buffer size the runtime remains constant with minimal overhead added by the data collection and sending (adding only the time to retrieve the valueset from the component and insert it into the buffer, both of which occur in memory). The results are consistent with the expected minimum buffer size as speedup becomes constant as the buffer size approaches 24 MB.

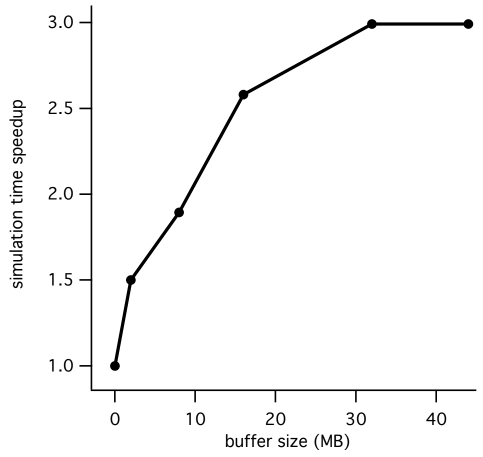

Fig. 6: Speedup for varying buffer sizes.

3.4 Distributed Cooperation

The entries in the local buffer of each DCC are evenly distributed among all the active DCC's on a cluster. This results in entries migrating away from DCC's that are collecting faster then they are delivering, and toward DCC's that are delivering faster then they are collecting, which enables cooperative sending of the data.

To evaluate the effect of cooperation among DCC's we executed concurrent simulations on multiple cluster nodes with differing rates of data collection and data delivery. A single composition consisting of a single DCC and producer executed on each compute node and we measured the *completion time* which we define as the amount of time necessary for all data to be delivered. In the ideal case the completion time is equivalent to the simulation time, but may extend past the simulation time if the simulation completes before all data is delivered.

Four *fast-production* nodes were configured such that data was produced at a faster rate than it could be delivered, and the other *slow-production* nodes were configured such that data was produced at a slower rate than it was delivered. Given a sufficient number of nodes, the runtime remains constant with minimal overhead added by the data collection and sending (adding only the time to retrieve the valueset from the component and insert it into the buffer, both of which occur in memory). The following equation describes the necessary balance of capacity across a cluster to ensure that simulation runtimes are not affected by the data collection. For N nodes over a given simulation period:

$$0 = \sum_{i=1}^{N} InRate_i - OutRate_i \qquad (4)$$

In this configuration, the 4 fast-production nodes inserted 0.5 entries per second while the remaining slow-production nodes each inserted 0.01 entries per second. All nodes removed entries at the rate of 0.1 entries per second. By Equation 4, 32 slow-production nodes would be sufficient to collectively match the rate of 1.6 entries per second inserted by the four fast-production nodes. This assumes that the slow-production nodes are actively delivering data during the complete simulation period.

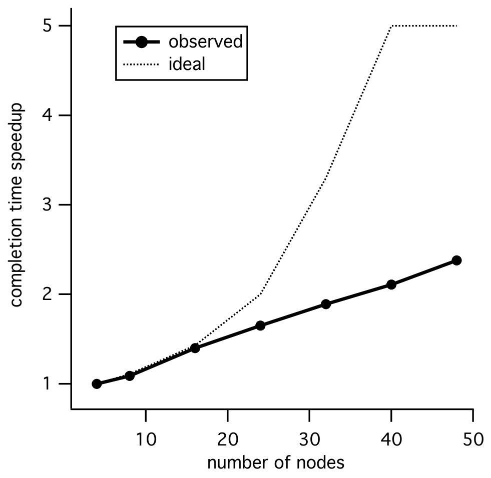

Fig. 7: Speedup for varying numbers of nodes.

We observed near-ideal speedup in the average completion time of the four fast-production nodes for up to 16 nodes as shown in Figure 7. As the number of nodes increased beyond 16, the rate at which entries were added to the distributed buffer was not sufficient to ensure each slow-production node had entries in its local buffer, resulting in the slow-production nodes not sending at all times (as in the ideal case). For example, each slow-production node spent an average of 31% less of its excess time delivering data when there were 48 nodes than when there were 16 nodes. Enabling the cooperation across nodes reduces the completion time, but achieving ideal speedup is contingent on the availability of sufficient data at each node.

4. Related Work

The DCC automates both the task of collecting model output data and transferring them to online services. These are typically performed manually or using custom software [6] that execute general-purpose tools such as GridFTP [7]. Web services have been utilized in modeling and simulation since their conception as both a means to access data and to control the execution of online models [8], [9], [10], [11]. The DCC provides a point of integration between linked models and any Internet-connected data platform that supports web services, including Workflow Management Systems such as Taverna [12] and Vistrails [13] and distributed data storage systems such as iRODS [14] and HIS [15]. It complements existing methods for input data retrieval from online services for OpenMI linked models [16], [17], [18].

5. Conclusions

We presented the design of the Data Collector Component (DCC) for OpenMI components and evaluated its performance. The DCC collects model output data from model components and efficiently delivers them to web services. It utilizes a distributed buffer optimized for the unique behavior and constraints of the OpenMI. General-purpose data collector components simplify the task of collecting model output within a grid environment and facilitate storage and post-processing by online services.

The DCC consists of a buffering module and a delivery module. The buffering module obtains output data from one or more components as they are created and stores them in a distributed buffer that is shared among all DCC's executing on a cluster. The delivery module continuously monitors the buffer and delivers data to web services in a way that balances efficiency and latency.

We evaluated the performance of the DCC and its sensitivity to its three parameters: maximum buffer size, minimum delivery size, and maximum residence time. The minimum delivery size was found to have a significant impact on the throughput and to achieve at least 50% of the maximum throughput each message must contain at least 2.6×10^5 elements in our experimental configuration. The maximum residence time resulted in an average residence time that is one-half the maximum residence time when entries are added to the buffer at a regular interval. We found that buffering resulted in maximum speedup in simulation time (a factor of 3) within a single composition, and distributed cooperation resulted in a speedup in the completion time by a factor of 2.4.

As the importance of data availability, interoperability and transparency continue to rise, so too does the need for software tools to facilitate these. General-purpose tools that intelligently and efficiently collect and deliver data will become an essential part of OpenMI linked models on workstations and grids alike and this work provides a starting point for such tools.

6. Acknowledgments

This work was supported by the National Science Foundation (grants GEO0909515, EPS0919443, EPS1006860). Access to the Beocat compute cluster at the Dept. of Computing and Information Sciences at Kansas State University was appreciated.

References

[1] J. B. Gregersen, P. J. A. Gijsbers, and S. J. P. Westen, "OpenMI: Open modeling interface," *J. Hydroinform.*, vol. 9(3), pp. 175–191, 2007.

[2] OGC, "Open geospatial consortium," 2012, http://www.opengeospatial.org.

[3] C. Portele, "OpenGIS geography markup language (GML) encoding standard," Open Geospatial Consortium, 2007, OGC 07-036.

[4] S. Cox, "Observations and measurements - XML implementation," Open Geospatial Consortium, 2011, OGC 10-025r1.

[5] T. Ozturk, "Scalable data structures for java," in *Devoxx*, Metropolis Antwerp Belgium, November 2010.

[6] M. Papiani, J. L. Wason, A. N. Dunlop, and D. A. Nicole, "A distributed scientific data archive using the web, XML and SQL/MED," in *SIGMOD RECORD*, vol. 28, 1999, pp. 56–62.

[7] G. Toolkit, "GridFTP user's guide," 2012, http://www.globus.org.

[8] S. Chandrasekaran, G. Silver, J. Miller, J. Cardoso, and A. Sheth, "Web service technologies and their synergy with simulation," *Winter Simulation Conference*, vol. 1, pp. 606–615, 2002.

[9] J. M. Pullen, R. Brunton, D. Brutzman, D. Drake, M. Hieb, K. L. Morse, and A. Tolk, "Using web services to integrate heterogeneous simulations in a grid environment," *Future Gener. Comput. Syst.*, vol. 21, pp. 97–106, January 2005.

[10] S. Shasharina, C. Li, R. Pundaleeka, N. Wang, D. Wade-Stein, D. Schissel, and Q. Peng, "HDF5WS – web service for remote access of simulation data," *APS Meeting Abstracts*, p. 2014, October 2006.

[11] J. Horak, A. Orlik, and J. Stromsky, "Web services for distributed and interoperable hydro-information systems," *Hydrol. Earth Syst. Sci.*, vol. 12, pp. 635–644, 2008.

[12] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. Pocock, P. Li, and T. Oinn, "Taverna: a tool for building and running workflows of services," *Nucleic Acids Research*, vol. 34(Web Server issue), pp. 729–732, 2006.

[13] L. Bavoil, S. P. Callahan, C. E. Scheidegger, H. T. Vo, P. J. Crossno, C. T. Silva, and J. Freire, "Vistrails: Enabling interactive multiple-view visualizations," *Visualization Conference, IEEE*, vol. 0, p. 18, 2005.

[14] A. Rajasekar, M. Wan, R. Moore, and W. Schroeder, "A prototype rule-based distributed data management system," in *HPDC workshop on Next Generation Distributed Data Management*, Paris, France, 2006.

[15] T. Whitenack, "CUASHI HIS Central 1.2," 2010.

[16] T. Bulatewicz and D. Andresen, "Efficient data access for open modeling interface (openmi) components," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA) Volume 1, ed. H. R. Arabnia, CSREA Press, Las Vegas, Nevada, USA, July 18-21*, 2011, pp. 822–828.

[17] Q. Harpham, "Future service chain platform," in *First Open Consultation Meeting, Distributed Research Infrastructure For Hydro-Meteorology Study*, Genoa, Italy, October 2010.

[18] KISTERS, "Kisters news," 2010, http://www.kistersnews.com.