

Efficient data access for Open Modeling Interface (OpenMI) components

Tom Bulatewicz, Daniel Andresen

How to cite this presentation

If you make reference to this version of the manuscript, use the following information:

Bulatewicz, T., & Andresen, D. (2011, July). Efficient data access for Open Modeling Interface (OpenMI) components. Retrieved from <http://krex.ksu.edu>

Citation of Unpublished Symposium

Citation: Bulatewicz, T., & Andresen, D. (2011, July). Efficient data access for Open Modeling Interface (OpenMI) components. Paper presented at the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, NV.

This item was retrieved from the K-State Research Exchange (K-REx), the institutional repository of Kansas State University. K-REx is available at <http://krex.ksu.edu>

Efficient data access for Open Modeling Interface (OpenMI) components

Tom Bulatewicz, Daniel Andresen

Dept. of Computing and Information Sciences, Kansas State University, Manhattan, KS, USA

Abstract—Data management for linked (or coupled) simulation models can be a challenging task when deploying to grid environments. In cases where the linked models conform to a standard interface for data input and output, general-purpose data providers can be used to supply data to the models from online sources, reducing the complexity of the deployment. We have developed a data provider component that conforms to the Open Modeling Interface (OpenMI) that is suitable for use on computational grids. Through the application of three techniques, caching, prefetching, and pipelining, the component efficiently retrieves data from standards-based web services and delivers the data to OpenMI-compliant models. Each technique resulted in varying performance improvements both within a single simulation and across multiple simulations concurrently executing on a cluster. In this paper we report on the design of the component and the evaluation of its performance.

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

1. Introduction

Computer models require input data in order to perform simulations. This data may originate from a variety of sources, may vary in space and time, and may be used during the initialization of a simulation and during its execution. In the case of linked (or coupled) models that execute independently and cooperate to collectively perform a simulation, each model requires its own input data. Models often have unique input data formats requiring inputs to be individually prepared for each model prior to the simulation run and often results in duplication of the data that is common between models. These input datasets must be deployed with the models to the execution environment, such as a computational grid. To obviate the need to prepare input data in model-specific formats and to increase the portability of datasets between models, standard data formats have been developed (e.g. netCDF [1]). To obviate the need to deploy datasets to the execution environment and to provide access to real time measurement data, data distribution frameworks have been developed that allow models to access online data sources (e.g. via web services). In the general case, models must have the capability to use these standard data formats and data distribution frameworks. In the case of models that are software components with well-defined

input/output interfaces (e.g. CCA [2]), data access using these standards can be implemented in general-purpose data components which can be linked to model components. Such data provider components play an important role in any linked modeling environment.

The Open Modeling Interface (OpenMI) [3] provides 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 corresponds to the exchange items) at runtime.

The *GetValues* function has three parameters and returns a set of values as illustrated in Figure 1. The parameters

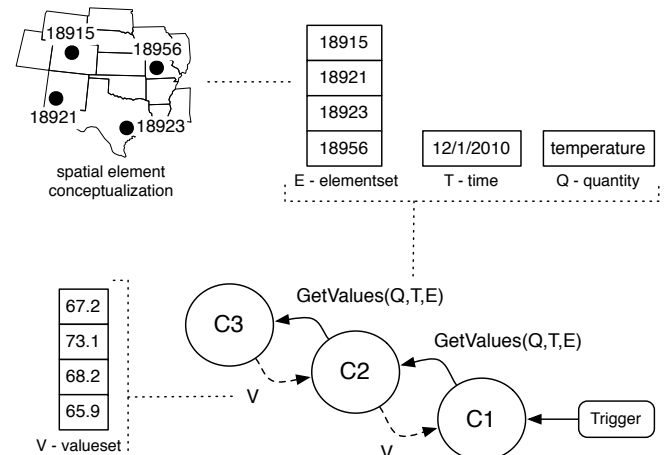


Fig. 1: OpenMI pull-based execution. Solid lines indicate function calls and dashed lines indicate the flow of data.

collectively identify a specific set of values that represent the state of a quantity at a single point in time over some spatial distribution. The quantity is described by a textual identifier, the time by a modified-Julian date, and the spatial distribution by a list of spatial elements called an *elementset*. An elementset is a list of geo-referenced spatial elements such as a point, line segment, or polygon. The *GetValues* function returns a list of floating-point values called a *valueset*. The values in a valueset describe the state of a single quantity at a specific point in time where each individual value corresponds to a different spatial element (based on array index).

The *GetValues* function not only provides a means for the exchange of data between a set of linked components (called a *composition*) but it also provides a means for their coordinated execution at runtime. A special component called a *trigger* begins by invoking *GetValues* on one of the components. The first time *GetValues* is invoked on a component it begins executing (e.g. performing time steps) from its starting simulation time. The component executes until it requires input values for one of the quantities on one of its links, at which point it invokes *GetValues* on the component at the providing end of the link and blocks. The component blocks until the call to *GetValues* completes and the values are returned, at which point it continues its execution. The components take turns executing and *pull* data from each other until the simulation completes. A component only performs time steps as-needed in direct response to a call to its *GetValues* function.

Compositions can be created in a highly automated way. Using visual software tools, a scientist chooses a set of components of interest, interactively specifies the links between them, and then executes the simulation. Each *link* maps an output exchange quantity from one component to the input exchange quantity of another component and links can be uni-directional or bi-directional.

In this work we present the design and evaluation of a general-purpose Data Provider Component (DPC) that is capable of delivering data from online sources to OpenMI components. We describe the design and implementation of the DPC 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

2.1 System Overview

Figure 2 illustrates the movement of data through a distributed data delivery system for linked model components. Compositions of linked components execute on cluster nodes. Each composition includes a DPC that retrieves data from web services and provides it to the other components. DPCs within compositions that are running on different cluster nodes share data with each other. When a component

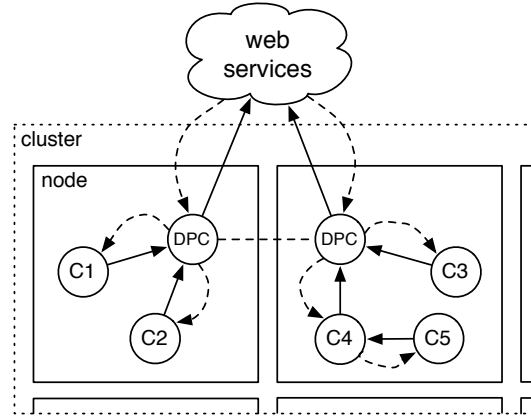


Fig. 2: System overview.

needs input values it invokes *GetValues* on the DPC for the needed quantity, time, and spatial elements and the DPC returns the appropriate valueset. The DPC calls web services for a specific quantity identifier, time, and list of location identifiers (which correspond to elements) and then extracts the valueset from the response. Thus the content of a web service call mirrors that of a *GetValues* call. We assume that each spatial element has a unique identifier, although this is not enforced by the OpenMI. Any web service that can be queried for a quantity, time, and list of locations and returns a list of values could be used as a data source for the DPC.

Within the context of water resources, there are several standards for data models to store observations data and web services to access them [4]. The WaterOneFlow [5] web service API has recently been utilized by several government agencies (<http://hiscentral.cuahsi.org>) and has a *GetValues* method that can be queried for an individual quantity in a single location or region for a timespan. Time series data is returned in XML that conforms to the WaterML schema [6].

The initial implementation of the DPC supports SOAP calls to WaterOneFlow web services and to a custom variant that allows multiple quantities and locations to be queried in a single call which is not currently supported by the WaterOneFlow API but is necessary to evaluate the performance of the DPC. The implementation supports parsing WaterML responses via SAX. An example of a request and response is shown in Figure 3. The DPC's configuration file defines its output exchange items (i.e. quantities and elementsets) and the information about each web service, such as the URL, API, response format, and which quantities can be queried. The implementation can be extended to support other web services and response formats.

If the DPC were to make a web service call on each invocation of *GetValues* then the rest of the composition would be paused for the duration of the call, adding to the overall runtime of the simulation. In addition, invocations by different components for the same values would result in

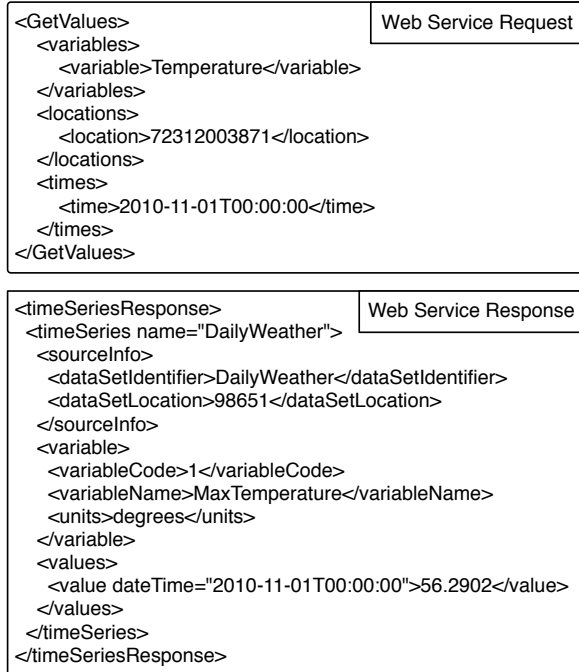


Fig. 3: Example web service request and response (attributes and SOAP envelope removed for clarity).

duplicate web service calls which would also increase the overall runtime and inefficiently utilize network bandwidth and other resources. To minimize the effect that the DPC has on the execution of the composition in terms of runtime and resource use, the DPC must (1) minimize the time it takes for `GetValues` to return a valueset and (2) minimize the number of times a valueset is retrieved from a web service.

In the ideal case there would be zero wait time and each valueset would be retrieved from a web service once. To these ends, the DPC utilizes three strategies: caching, prefetching, and pipelining. All valuesets are retrieved from web services once and are then cached so that they are immediately available for subsequent invocations of `GetValues` by other components (within and across compositions) and subsequent executions of the composition. Since components typically advance forward through simulation time, valuesets are prefetched so that they are available in the cache before they are requested. Multiple web service calls are performed simultaneously in a pipelined fashion to maximize use of available network bandwidth.

The DPC consists of a fetching module and a caching module as illustrated in Figure 4. The fetching module identifies the valuesets that the other components need, retrieves them from the web services, and then stores them in the cache. The caching module handles calls to `GetValues` by retrieving the appropriate values from the cache, assembling the valueset, and returning it to the calling component.

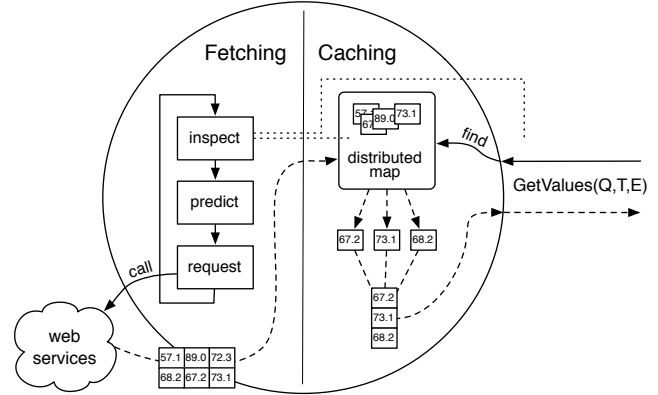


Fig. 4: Operation of the data provider component. Solid lines indicate function calls and dashed lines indicate the flow of data.

2.2 Caching

During the execution of a composition, several components within a single composition may request the same valuesets from a DPC. Components in independently executing compositions on different cluster nodes may request the same valuesets from different DPCs. In both cases it is advantageous for the DPCs to cache the valuesets that they retrieve from the web services and to share those valuesets across all the DPCs that are executing simultaneously in different compositions across a cluster. The same valuesets may be needed on subsequent executions of the same composition so it is also advantageous for the cached valuesets to be persisted between executions.

To serve these needs, a clustering, scalable data distribution platform (Hazelcast [7]) is utilized by the caching module to store the values retrieved from the web services. Each DPC has an instance of the platform peer that is managed by a set of threads within the same process as the DPC. Instances dynamically cluster and discover peers via multicast and communicate via TCP/IP. Thus, there are no servers involved and each DPC is self-sufficient and shares data directly with other DPCs. The data structure used to store the values is a distributed map. Entries in the cache are evenly partitioned onto the currently executing instances across the cluster (i.e. DPCs). Each instance uses a private database file [8] to persist the cache entries between executions.

The valuesets that are retrieved from the web services are decomposed into individual values and stored in the distributed map. Storing the individual values allows the map to assemble different valuesets ad-hoc as they are requested by model components, maximizing the reusability of data and the effectiveness of the cache. Alternatively the complete valuesets could be stored as single entries in the map but this would limit the reusability of the data to cases where subsequent requests are for the same combination of

quantity, time, and elementset.

The DPC may be linked to several model components that use different time steps, different input element sets, and different units. It is the responsibility of the providing component (the DPC) to apply necessary transformations (units, spatial, temporal) to meet the input of the requesting component (the model). To maximize the effectiveness of the cache the DPC caches the values retrieved from the web service and when a component requests a valueset the appropriate values are extracted from the cache, transformed, and provided to the model. Alternatively the DPC could cache the transformed valuesets but this would limit the reusability of the data to cases where subsequent requests are for the same combination of quantity, time, elementset, and units.

Entries in the distributed map are tuples of the form $\langle \text{value}, \text{availability} \rangle$ and are keyed by a textual string that uniquely identifies a value by the concatenation of its time, quantity identifier, and element identifier. The value is a floating point number and the availability flag is a boolean that indicates whether the value has been retrieved (true) or a web service call for the value is in progress (false). The contents of the map are persisted to the database upon completion of a composition run and then restored into the map upon startup.

When `GetValues` is called by a component, the DPC checks to see if the value for each element of the requested elementset exists in the cache by creating the key and then performing a get operation on the distributed map using the key. If values for any of the elements are missing then the valueset is not available and the cache waits for a period of time before checking again (during which the composition is blocked). The cache relies on the fetching module to populate the map with values.

2.3 Fetching

The simulation of physical processes (especially those for which the OpenMI was initially designed) typically involve the calculation of output quantities over a simulation period. The components step through simulation time and periodically request values from each other. The frequency at which `GetValues` is invoked on the DPC by other components is likely not the most efficient frequency for the DPC to call the web services. For this reason the DPC prefetches valuesets to minimize the time that the other components must wait when calling `GetValues` on the DPC. Multiple web service calls are issued simultaneously in a pipelined fashion to take advantage of multi-core and multi-host web services.

Throughout the execution of a composition the components are at approximately the same point in simulation time. This is because components typically require input data that reflects their current simulation time which requires that the components providing the inputs advance to that same point in simulation time. Prefetching is thus most effective when

it is done such that the data for all components is prefetched to the same future point in simulation time.

Prefetching relies on knowledge of what data will be needed before it is requested. It is not possible for the DPC to obtain this information directly from the other components as this functionality is not supported by the OpenMI. The DPC predicts what valuesets will be requested in the future by observing what valuesets have been requested in the past. Components that use a fixed-length time step request data from the DPC at fixed intervals making it possible for the DPC to identify these components and determine the length of their time steps. In such cases the DPC can accurately predict the valuesets that will be requested in the future. It is more difficult for the DPC to predict the data needs of components that use a variable-length time step and is not addressed in this work.

Web service calls request different combinations of quantity, time, and spatial elements, and thus these calls may be coalesced along any or all of these three dimensions. Since our goal is to minimize wait time, the coalescing strategy should group together similar requests as much as possible without inducing additional wait time. Of the three dimensions, only spatial coalescing is guaranteed to not incur any unneeded wait time because the DPC cannot provide a valueset to a model component unless all values for the entire elementset have been retrieved. Coalescing by time or quantity may result in a model component waiting a longer period of time for a valueset that is part of a larger request than it would have if the valueset was requested individually. For this reason the DPC only coalesces web service requests spatially such that each valueset, corresponding to a single quantity and time over a complete set of spatial elements, is requested in each web service call.

2.3.1 Runtime Operation

The fetching module manages a *fetch thread* for each web service. It is responsible for identifying the valuesets that must be retrieved from the web service and issuing the web service calls to retrieve them. When the state of the fetch thread changes (as a result of a call starting or completing) or a cache miss occurs, the fetch thread attempts to identify and download as many valuesets as possible. A single quantity may be provided along several links at different temporal intervals or different spatial elementsets, so each link must be checked for necessary valuesets individually. Each attempt makes a series of passes through the links until no valuesets are found or until there are no available resources. On each pass, the earliest valueset (either by request or predicted) needed by a link that is not already in the cache is identified and a web service call is started (at which time placeholder entries (availability = false) for the values are created in the cache to indicate that they are being retrieved). Multiple DPCs may request intersecting valuesets in which case only partial valuesets are requested from the web service.

Resource usage maximums can be externally parameterized statically in terms of the maximum number of concurrent web service calls and dynamically in terms of maximum network bandwidth or CPU utilization (via the Java Management Extensions). When the maximum number of concurrent calls is met or the maximum CPU or network bandwidth is reached, prefetching stops for a period of time before it is attempted again.

To keep all links prefetched to approximately the same point in simulation time, each link is prefetched up to a moving limit: $limit = \min\{p + n \times i, e\}$ where p is the earliest time to which all links are prefetched to, i is the longest request interval across all links, and e is the ending time of the composition. The constant n controls how close in simulation time the links are to be prefetched to and should be 1 when the difference in request intervals is large and may be higher when the difference is small.

3. Experimental Results

We conducted a performance study using an onsite Linux-based Beowulf cluster. The compute nodes had 16-core 2 GHz processors and 64 GB of memory and the server node had a 4-core 2.7 GHz processor and 8 GB of memory connected via gigabit ethernet. The software components were implemented in Java (Alterra SDK) conforming to version 1.4 of the OpenMI and the web service was implemented in ASP.NET. The time spent by the web service to generate the response to each request was configurable and the contents of the response contained random numbers.

To represent a model component we created a placeholder component that used a fixed-length time step of 1 day and would sleep for 10 s between time steps to mimic the time spent calculating a time step, which we call the *processing time*. There was a single link between the model component and the DPC and the elementset consisted of 50000 elements. Each composition ran for 60 time steps resulting in a total of 3 million values in the cache at the end of the run.

3.1 Caching

To investigate the effect of the cache we performed two sets of simulations with varying numbers of model components with and without the cache and measured the runtime and amount of data transferred (prefetching was disabled). In the first set, the model components were part of a single composition and linked to a single DPC. In the second set, each model component was in a separate composition running on a different node so each composition consisted of one model component and one DPC. In both sets the model components request the exact same valuesets to maximize the effect of the cache. Since the design of the DPC necessitates a cache, the effect of disabling the cache was approximated by removing each valueset from the cache after it was returned to a model component.

When a component invokes `GetValues` on a DPC the requested valueset is either retrieved from the cache if it exists (a cache hit) or it is retrieved from a web service if it does not (a cache miss). We define *wait time* to be the amount of time that it takes a call to `GetValues` to return a valueset. We define *total wait time* to be the sum of all wait times over the course of a composition run and may be estimated by:

$$W = \sum_{i=1}^m (R_i + N_i + X_i) + \sum_{j=1}^{m+h} L_j \quad (1)$$

where m is the number of cache misses, h is the number of cache hits, R is the average web service response time, N is the average data transfer time, X is average time to parse a response, and L is the cache lookup time. We empirically identified values for N , X and L in our estimation of expected performance.

The effect of the cache is shown in Figure 5 (top). When caching was disabled the wait time increased linearly as the number of components increased (top-left). Enabling the cache within a single composition achieved constant wait time. In the case of distributed compositions there was a super-linear increase in the wait time due to the higher cache lookup time associated with the distributed map. The total data transferred increased linearly with the number of components when the cache was disabled and remained constant at 1.5 GB when the cache was enabled (top-right).

3.2 Prefetching

To evaluate the effect of prefetching we measured the wait time of a single model component linked to a DPC as we varied the web service response time. We used response times that were multiples of the model component's processing time. The DPC prefetched valuesets with a limit of one active web service call at any one time.

In the ideal case there is perfect predictive capability and perfect overlap of the time that the DPC spends prefetching and the time that the model component spends calculating time steps. In this case the total wait time is based on the time spent performing concurrent operations C and performing serial operations S :

$$C = \sum_{i=1}^m (R_i + N_i + X_i) \quad (2)$$

$$S = \sum_{k=1}^t P_k + \sum_{i=1}^{m+h} L_i \quad (3)$$

where t is the total number of time steps performed by the model component and P is the processing time of each time step. The total wait time W with prefetching may be estimated by:

$$W = \max(C - S, 0) + \sum_{i=1}^{m+h} L_i \quad (4)$$

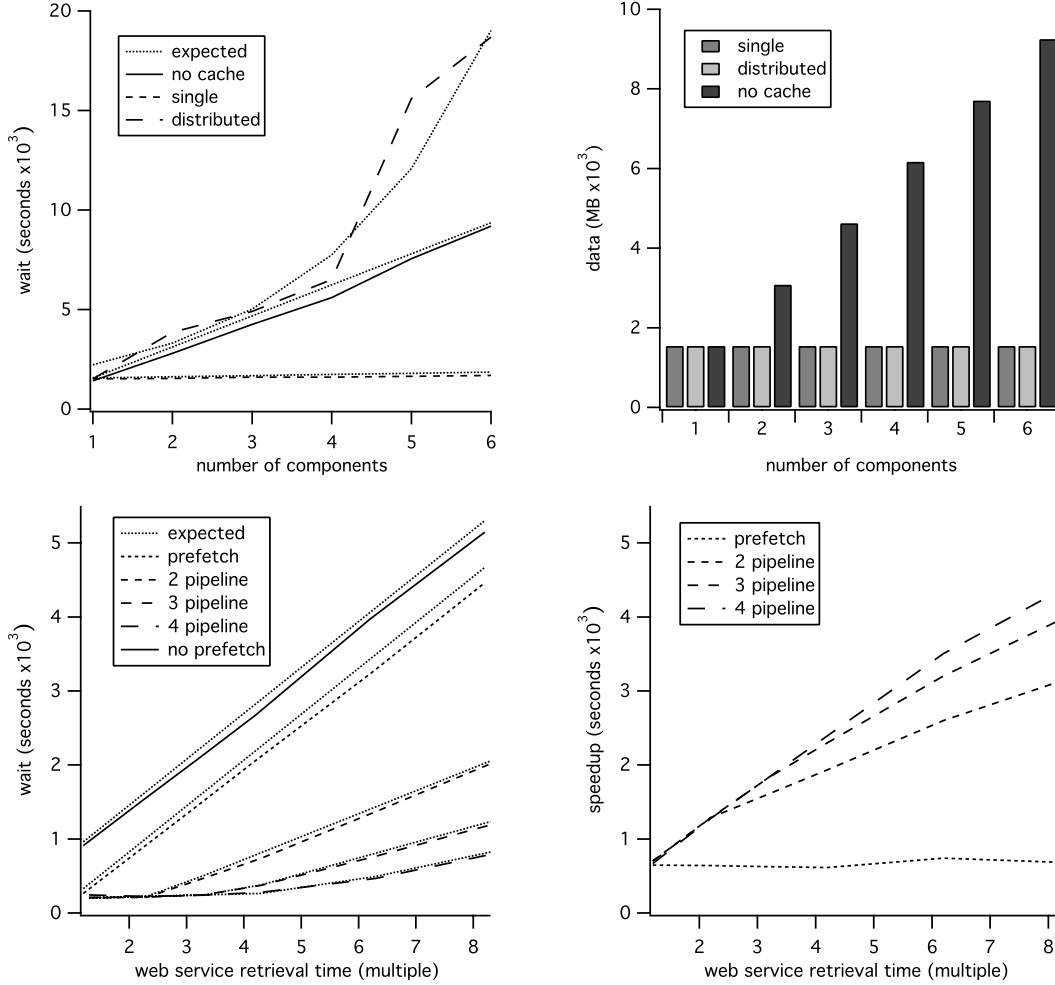


Fig. 5: Performance results: effect of caching (top) and effect of prefetching and pipelining (bottom).

We refer to $R_i + N_i + X_i$ as the *web service retrieval time* as it reflects the total time necessary to retrieve a valueset from a web service.

With prefetching disabled the wait time increased linearly with the web service response time (Figure 5 bottom-left). When prefetching was enabled the total wait time increased at the same rate but was lower by a constant value that corresponded to one web service retrieval time. The speedup in this case was constant (bottom-right).

3.3 Pipelining

To evaluate the effect of multiple concurrent web service calls we measured the wait time of a single model component linked to a DPC as we varied both the number of concurrent requests and the web service response time.

When web service calls are made simultaneously the time spent on concurrent operations is reduced by a factor of the number of simultaneous calls. Thus, the total wait time W

with pipelining may be estimated by:

$$W = \max\left(\frac{C}{Q} - S, 0\right) + \sum_{i=1}^{m+h} L_i \quad (5)$$

where Q is the number of simultaneous web service calls.

The wait time remains constant as long as the number of simultaneous web service calls is the same as the factor by which the web service retrieval time is greater than the model component processing time. The wait time increases sub-linearly with the web service retrieval time once the number of simultaneous web service calls is insufficient to complete all of the necessary prefetching during the model component processing time.

4. Related Work

The synergy between web services and modeling and simulation was recognized quickly as web standards emerged [9]. Web services can provide both a means to access data and to control the execution of online models [9],

[10], [11], [12]. Workflow Management Systems (WMS) provide an infrastructure to setup, execute, and monitor scientific workflows composed of web services [13], [14]. Standard data formats [1], [15] and data access systems [16] have been developed to improve data portability.

There has been recent interest in enabling OpenMI components to interoperate with web services. In one effort [17] a *feature type component* was developed that can retrieve point time series data from a server using the OGC Web Feature Services (WFS) standard. The component steps forward through time and when `GetValues` is invoked it returns the value of the feature type that corresponds to the current time. Another component [18] was developed that can retrieve time series data from a proprietary data platform. To obviate the need for a local data server on the machine that is executing a composition, the component serves as a proxy between the components of a composition and the data server. Our work complements these efforts in the context of computational grids.

5. Conclusions

We presented the design of the Data Provider Component (DPC) for OpenMI components and evaluated its performance. The DPC efficiently retrieves data from multiple web services and delivers the data to OpenMI components that are executing on a cluster. We adapted three common-practice optimizations, caching, prefetching, and pipelining, to the unique behavior and constraints of OpenMI components. General-purpose data components simplify the task of deploying linked models to runtime environments and provide a means for integrating real-time measurement data into their simulations.

The DPC consists of a fetching module and a caching module. The fetching module continuously monitors the data requests made by model components and prefetches data from web services in a pipelined fashion. The caching module services the data requests by extracting the appropriate data from a distributed map that is shared among all DPCs across a cluster.

We evaluated the performance of each of the three optimizations: caching, prefetching, and pipelining. Caching within a single composition achieved linear speedup in wait time as the number of model components increased. Distributed caching was less performant in terms of wait time and would be most advantageous in situations with high latency web services. In both cases the amount of data transferred was minimized and remained constant as the number of model components increased. Prefetching achieved constant speedup in wait time as the web service retrieval time increased and pipelining achieved linear speedup given a sufficient number of simultaneous web service calls.

To mitigate some of the challenges of data management for linked simulations, intelligent, efficient data provider components will become an essential part of any OpenMI

linked model. We believe that this work provides a sound basis for the development of such components.

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] R. K. Rew and G. P. Davis, "The Unidata netCDF: Software for scientific data access," in *Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography, and Hydrology*. Anaheim, California: American Meteorology Society, February 1990, pp. 33–40.
- [2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski, "Toward a common component architecture for high-performance scientific computing," in *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, 1999, p. 13.
- [3] J. B. Gregersen, P. J. A. Gijssbers, and S. J. P. Westen, "OpenMI: Open modeling interface," *J. Hydroinform.*, vol. 9(3), pp. 175–191, 2007.
- [4] P. Taylor, "Harmonising standards for water observation data – discussion paper, OGC 09-124r2," *Open Geospatial Consortium Inc.*, 2010.
- [5] T. Whiteaker, "CUAHSI WaterOneFlow workbook, HIS document 5," *CUAHSI*, 2010.
- [6] I. Zaslavsky, D. Valentine, and T. Whiteaker, "CUAHSI WaterML, OGC 07-041r1," *Open Geospatial Consortium Inc.*, 2007.
- [7] T. Ozturk, "Scalable data structures for java," in *Devoxx*, Metropolis Antwerp Belgium, November 2010.
- [8] The HSQL Development Group, B. Simpson, and F. Toussi, "HyperSQL user guide: HyperSQL database engine (HSQLDB)," *The HSQL Development Group*, 2010.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] H. H. Page, "The HDF Group," 2010, <http://www.hdfgroup.org/HDF5/>.
- [16] 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.
- [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>.