



A distributed data component for the Open Modeling Interface



T. Bulatewicz^{a,*}, D. Andresen^a, S. Auvenshine^b, J. Peterson^c, D.R. Steward^b

^a Dept. of Computing and Information Sciences, Kansas State University, 234 Nichols Hall, Manhattan, KS 66502, USA

^b Dept. of Civil Engineering, Kansas State University, Manhattan, KS 66502, USA

^c Dept. of Agricultural Economics, Kansas State University, Manhattan, KS 66502, USA

ARTICLE INFO

Article history:

Received 22 July 2013

Received in revised form

1 February 2014

Accepted 28 February 2014

Available online 27 March 2014

Keywords:

OpenMI

Data management

Web services

Integrated modeling

ABSTRACT

As the volume of collected data continues to increase in the environmental sciences, so too does the need for effective means for accessing those data. We have developed an Open Modeling Interface (OpenMI) data component that retrieves input data for model components from environmental information systems and delivers output data to those systems. The adoption of standards for both model component input–output interfaces and web services make it possible for the component to be reconfigured for use with different linked models and various online systems. The data component employs three techniques tailored to the unique design of the OpenMI that enable efficient operation: caching, prefetching, and buffering, making it capable of scaling to large numbers of simultaneous simulations executing on a computational grid. We present the design of the component, an evaluation of its performance, and a case study demonstrating how it can be incorporated into modeling studies.

© 2014 The Authors. Published by Elsevier Ltd. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/3.0/>).

Software availability

Software name: DataComponent

Developer: GROWE/Kansas State University

Contact address: 234 Nichols Hall, Kansas State University, Manhattan, KS, 66502, 785-532-6350

E-mail: tombz@ksu.edu

Year first available: 2013

Hardware required: Architecture independent

Required software: Windows/Linux

Program language: C#

Program size: 2 MB

Availability: Download available under MIT License at: www.github.com/CNH-Hyper-Extractive/data-component

Cost: Free

1. Introduction

The rate at which data are being captured, created, and stored, is continuing to accelerate in the environmental sciences. The potential impact of these data on scientific discovery can only be

* Corresponding author.

E-mail addresses: tombz@ksu.edu (T. Bulatewicz), dan@ksu.edu (D. Andresen), auvenshi@ksu.edu (S. Auvenshine), jpeters@ksu.edu (J. Peterson), steward@ksu.edu (D.R. Steward).

realized if these data can be effectively utilized. This presents new challenges for data management and has inspired a number of initiatives for environmental information systems and cyberinfrastructure to meet this need (Horsburgh et al., 2009; Demir and Krajewski, 2013; Pettit et al., 2013; Mason et al., 2014; DataONE). These systems provide capabilities for organizing, storing, manipulating, and accessing data. Access to data is provided by web portals or desktop applications that enable scientists to search, analyze, and visualize data, as well as through machine interfaces, such as web services, that enable software systems to access these data.

This creates new opportunities for closer integration between simulation models and environmental information systems. Models can consume data directly from these systems and publish data to them. The often time-consuming tasks performed by a scientist to collect data from a myriad of sources, convert and transform the data, and assemble them into a set of model-specific input files (or databases), are no longer necessary. It is similarly no longer necessary to aggregate model output files, convert and transform them, and upload the final data sets.

In grid environments, in which computational resources consist of large clusters of machines, such integration is equally compelling. The greater scale at which these tasks must be performed, along with the additional tasks of deploying model input files to the compute resources and collecting the output files, typically requires a combination of manual tasks and ad-hoc automation techniques such as scripting. The development and maintenance of such

scripts for both processing the model-specific input and output files and for communicating with online services can involve a substantial effort. By enabling models to exchange data sets directly with environmental information systems the scientist is freed of these tasks. This also simplifies the complexity of the provisioning of models when executed in an automated fashion as part of workflow systems (Deelman et al., 2009) and service-oriented architectures (Nativi et al., 2013; Goodall et al., 2013).

In the context of component-based modeling and simulation in which compositions of models (and models themselves) are assembled from software components with standard input–output interfaces, the opportunities for integration with cyberinfrastructure are even greater because the data access capabilities can be encapsulated into data components that can be composed with other components. It is advantageous to separate data access functionality from the implementations of the models because it allows for more efficient operation (e.g. avoiding duplicate data retrieval by different components) and minimizes the software complexity of the model components. Through the adoption of standards, in both model component input–output interfaces (see Jagers (2010) for a review) and web service application programming interfaces, general-purpose data components can facilitate the exchange of data between model components and environmental information systems. Such components complement other software tools that support data access to these systems (such as the desktop applications ONEDrive for DataONE and CUAHSI-HIS's HydroGet).

We have developed a distributed data component that conforms to the Open Modeling Interface (OpenMI) (Gegersen et al., 2007), which both provides model components with input data retrieved from standards-based web services and delivers model output data to such services on each time step. By operating on a time step basis, the data component enables model components to consume dynamically-changing input data, such as measurement data from sensor networks, and to distribute output data in real-time. This also supports computational steering scenarios in which model output is monitored and inputs are manipulated as necessary as a simulation is being performed. The data component employs three techniques tailored to the unique design of the OpenMI that enable efficient operation: caching, buffering, and prefetching. This work unifies our previous efforts (Bulatewicz and Andresen, 2011, 2012) and includes improvements to the software design that achieve a significant increase in scalability. It also provides an integral part of an interdisciplinary modeling study in which we are integrating models of groundwater, economic decision making, and crop production to investigate the impact of policy on irrigated agricultural systems. The following sections position this work within the context of existing research and introduce the aspects of the OpenMI relevant to understanding the design and implementation of the data component. We then present the design of the data component in Section 2, an evaluation of its performance in Section 3, and a demonstration of how it may be incorporated into an integrated modeling study in Section 4.

1.1. Related work

This work lies at the intersection of component-based modeling, web services, and grid computing. The synergy between web services and modeling and simulation was recognized quickly as web standards emerged (Chandrasekaran et al., 2002). Web services can provide a means for both remotely controlling the execution of computer models running on servers or computational grids (Castronova et al., 2013a; Goodall et al., 2011; Horak et al., 2008; Pullen et al., 2005) and enabling desktop or grid-based models to exchange input and output data with online services. In the latter

case an online service may be composed of a suite of Internet applications and/or a collection of databases.

One class of online services that is well-suited for exchanging data with computer models is workflow management systems which are frameworks to setup, execute, and monitor scientific workflows, such as Taverna (Hull et al., 2006), VisTrails (Callahan et al., 2006), and Kepler (Altintas et al., 2004). Such systems could provide workflows that pre-process or post-process model data or conduct simulations whose input or output data is utilized by models. Another class of online services is data-centric and provides data storage (e.g. archiving) and retrieval (e.g. public access or sharing within or across institutions). Examples include the Integrated Rule-Oriented Data System (iRODS) (Rajasekar et al., 2006) which is a file-based distributed data storage system, the Consortium of Universities for the Advancement of Hydrologic Science, Inc. (CUAHSI) Hydrologic Information System (HIS) (Maidment, 2008; Tarboton et al., 2009) which facilitates the management of hydrologic data, DataONE (DataONE) which provides cyberinfrastructure for persistent Earth observation data, and Globus Online (Globus Online, 2013a) which provides online managed data storage based on GridFTP (Globus Toolkit, 2013b).

Web services provide a means for these online application and data services to achieve interoperability with one another and with client applications running on desktop computers and compute clusters. Standards for web services and the data encodings they use make it possible for independent applications to interpret exchanged data in a meaningful way. In the context of environmental modeling in which data is spatial-temporal in nature, the standards published by the Open Geospatial Consortium (OGC) for location-based information and services are of particular relevance. For example, the Web Feature Service (WFS) Standard (Vretanos, 2010) defines how geospatial data may be accessed from a web service and utilizes the Geographic Markup Language (GML) (Portele, 2007) Standard. Within the domain of hydrology, the CUAHSI HIS WaterOneFlow web service Application Programming Interface (Horsburgh et al., 2009) defines how time series hydrological observations data may be accessed and utilizes the Water Markup Language (WaterML) encoding standard (Zaslavsky et al., 2007).

The fundamental data model upon which these services and encodings are based (consisting of quantities, times, and locations) is generally compatible with the data model employed by the OpenMI for the exchange of data between components, making interoperability between services and components possible (Castronova et al., 2013b). Several OpenMI components have been developed that retrieve time series data from WFS web services (OpenMI Association, 2010). In a related work, Castronova et al. (2013b) enabled a desktop application to retrieve input data from WaterOneFlow web services and store them in a local database which could then be accessed by model components via a general-purpose data-access component.

Our work complements these efforts in two ways. First, our data component is not only capable of retrieving data from web services but delivering data to them as well. Second, the data component is not limited to use on desktop computers but may also be used on high-performance compute clusters. The prototype implementation is compatible with WaterOneFlow web services and is being extended to support additional standards. In our previous work (Bulatewicz and Andresen, 2011, 2012) we developed independent components for retrieving data from web services and delivering data to them. This work unifies our earlier efforts into a single component and includes fundamental changes to the software design to scale to significantly higher numbers of simultaneously executing simulations.

1.2. The Open Modeling Interface

The Open Modeling Interface (OpenMI) Standard (Gregersen et al., 2007) defines how software components may exchange spatial-temporal data with one another and coordinate their execution. Components that possess the capabilities defined by the interface can be linked together and exchange data, typically on each time step, as they carry out simulations. These capabilities are implemented as functions (specifically, object methods and properties) within the source code of a component that either provide descriptive information about the component (such as its inputs and outputs) or support its execution (such as performing initialization or exchanging data).

Each input and output is formalized as an *exchange item* that describes the properties of a variable (referred to as a *quantity* by the OpenMI) such as its name, units, and spatial distribution. The way in which a variable is spatially distributed is formalized as an *element set* that is composed of a list of *elements* each of which has a textual identifier, spatial shape (point, line, or polygon) and geographic coordinates. When configuring a linked model, called a *composition*, a scientist uses a visual software tool (the OpenMI Configuration Editor application – OmiEd) to choose a set of components and assign each input exchange item of a component to an output exchange item of another component. These assignments are called *links* and there may be multiple links between two components and these links may be in the same or opposite directions. At runtime a component requests data from other components along each input link, typically before performing each time step. The request is made by calling the *GetValues* function of each linked component specifying a date and time (or range) at which the data is needed, as illustrated in Fig. 1. The *GetValues* function returns a list of real numbers called a *value set* where each number represents the state of the variable at a different spatial location for the requested point or span of time. As such, each call to *GetValues* may be considered to be a request for the state of a variable for a list of spatial locations at a point or span of time and the response to be the list of numbers returned.

Although the *GetValues* function allows a value set to be requested that corresponds to either an instantaneous point in time or to a span of time, in this work we only consider the prior case, as the added complexity of the latter is outside the scope of this work. Value sets may only contain numeric values (categorical values are

not supported) as this is consistent with version 1 of the OpenMI Standard, which this work is based upon.

In addition to facilitating the exchange of data between components, the *GetValues* function provides implicit coordinated execution of components at runtime. The execution of a linked model is initiated when one of the components begins executing. On each time step the component invokes *GetValues* on each component linked to it to obtain all the necessary input value sets for the time step, pausing its execution during each invocation. When *GetValues* is invoked on a component, it executes as many time steps as necessary to advance to the requested point in simulation time and returns a value set corresponding to that time. Thus a component only executes time steps on-demand in response to the invocation of its *GetValues* function by another component and may itself invoke *GetValues* on other components prior to performing each of its time steps. In this way components take turns executing and pull data from one another until the initiating component's simulation is completed.

Components are typically model programs that consume input data and produce simulated output data, but they can serve other purposes as well. Examples include data conversion or transformation, data visualization, access to databases, and access to online data services as in the case of our data component.

2. Methods

2.1. Overview

The purpose of the data component is to serve as an intermediary between online data services and model components, by providing models with input data retrieved from web services and delivering model output data to web services. The design of the data component was guided by the following requirements:

1. To be general-purpose
2. To minimize the runtime of a simulation
3. To be scalable

Our design balances these three competing objectives making the data component broadly applicable and suitable for use on both desktop computers and compute clusters.

The first requirement of the data component is that it is general-purpose such that its inputs and outputs can be defined, and redefined, by a scientist as necessary for different sets of model components. The input and output exchange items of the data component reflect the quantities exposed by a web service: any quantities that a web service can provide or accept can be configured as exchange items of the data component. This is possible because the OpenMI defines the way in which data is exchanged between software components and web service standards define the way in which data is exchanged with online services. Together these standards make it possible for the data component to serve as a data relay between model components and web services.

The data component is configured (via a file) by specifying the list of input and output quantities that a web service can provide and accept, along with the element set definition of each and the web service URL and type. These quantities become available as input and output exchange items when the data component is added to a composition in the OmiEd application and can be linked to model components in the same way that links are added between model components.

The second requirement of the data component is that it minimizes its impact on the runtime of a simulation, ideally causing no increase. If a data component was to call a web service after each request received from a model component to either obtain input data or send output data, the simulation would be paused during the web service call (due to the synchronous execution of components) and increase the runtime of a simulation. This increase in runtime can be reduced or eliminated by decoupling the calls to the web services from the requests made by the model components. In order to decouple the web service calls from the model component requests, the data component must have the ability to temporarily store model input and output data in a *data store*. Rather than the data component call a web service in response to each request for input data from a model component, it first checks to see if the data is already available in the data store (by matching the variable, element set, and instantaneous timestamp). If it is, then it can be returned to the model component immediately, and if not, it can then be requested from a web service. There are two cases in which the data may already be available in the data store: (1) the data was previously requested by a model component, and (2) the data was retrieved from a web service ahead-of-time. We refer to the prior as *caching* and the latter as *prefetching* and these techniques can reduce, and in some cases

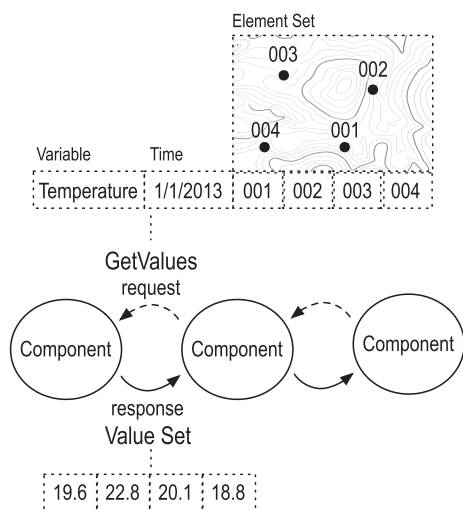


Fig. 1. Lists of real numbers called *value sets* are exchanged between model components.

eliminate, the increase in runtime due to the web service calls. In addition to minimizing the runtime, caching also minimizes the amount of data downloaded from the web services because each input is only retrieved once. The data store is shared among all simulations executing across a compute cluster to maximize the reusability of the cached data. With respect to sending output data, rather than call a web service in response to each request from a model component, the data component immediately stores the output data in the data store and sends it at a later time. We refer to this as *buffering* and it eliminates the increase in runtime otherwise due to sending output data to web services.

The third requirement of the data component is that it is scalable such that many simulations, each containing an instance of the data component, may execute concurrently across a compute cluster with minimal impact to the runtime of the simulations. To these ends we employed two strategies: (1) maximize network efficiency when sending data to web services, and (2) separate the data component software into two tiers.

Network utilization is inefficient when the amount of data being sent is small enough that the network latency is comparable to the transmission time of the data (i.e. the duration of time and amount of data exchanged at the network transport layer for establishing the connection and for sending the data are similar). To ensure that the network bandwidth is used efficiently when sending model output data to web services, the data component sends a sufficient amount of data in each web service call. With respect to retrieving data, which consists of values that each represent a variable at a point in time for a location, the data component could request groups of values in each web service call for spans along any of these three dimensions in each web service call. At one extreme it could make a web service call for each individual value, and at the other extreme it could make a single web service call to obtain all the input values required for a complete simulation. In the prior case the network utilization may be inefficient due to the small data size of a single value, and in the latter case the execution of a simulation would be delayed until the data is retrieved and may require storing a large amount of data for the lifetime of the simulation (in addition it would prohibit both real-time online data access during the simulation and the ability to utilize multi-threaded and multi-hosted web services). Efficient network utilization can be balanced with real-time data access by requesting groups of values in each web service call (essentially coalescing what would otherwise be multiple requests into a single request). Values could be grouped by time, variable, and/or location, depending on the capabilities of a web service. In addition, grouping by time would require the data component to be capable of predicting the simulation times at which model components will request data and grouping by variable would only be possible in cases in which the data component is providing multiple quantities to one or more model components that are sourced from a single web service and requested for the same points in simulation time. We designed the data component such that requests are grouped by location (when supported by the web service) and left grouping by time and variable to be addressed in future work due to the additional complexity.

The data component software is organized into two tiers that separate the management of the data store and communication with web services from the interactions between the components within a composition. This is a more scalable design than our previous work (in which there was a single tier) because the management of the data store requires considerable computer resources (memory, processor, and network) yet accessing the data for providing input data to model components and collecting output data requires few resources. Without separating them, the resource demands of the data store are imposed on each data component thus increasing the resource demands of every simulation. By separating them, the amount of resources dedicated to the management of the data store can be managed separately from those required by the individual simulations. The number of *data managers* that manage the data store can be increased or decreased independently from, and as necessary to support, the number of simultaneous simulations.

An overview of the system is illustrated in Fig. 2. Compositions of linked components perform simulations on the nodes of a cluster. Each composition includes a data component (labeled DC in the figure) whose input and/or output exchange items are linked to model components. Model components request input from data components (by invoking *GetValues*) for a variable at a specific time and element set in the same way as from other model components. The data component in turn requests the input data from a data manager which may obtain the data from the data store or retrieve it from a web service to fulfill the request. Each time a model component produces output data (in response to a *GetValues* request from another model component) the data component is notified. When notified, the data component obtains a copy of the output data (by invoking *GetValues* on the model component) and sends them to a data manager which stores the data for eventual delivery to a web service.

2.2. The data store

Data managers are responsible for both communicating with web services and managing the storage of model input and output data in the data store. We implemented a set of software modules that provide the functionality to communicate with web services and utilized an existing software for the data store functionality. The data store is a *key-value store*, which is a non-relational database in which related data is aggregated together and stored as an entry that is accessed via

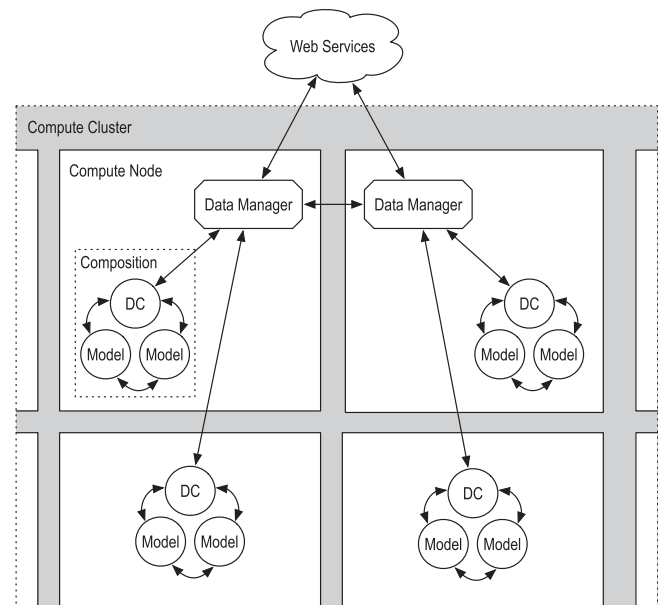


Fig. 2. System overview. Arrows indicate the movement of data.

a unique identifier. We chose to utilize a key-value store because storing data in this way achieves high performance when scaling horizontally (i.e. increasing the number of compute nodes to allow for higher capacity) because the data can be efficiently sharded and replicated across compute nodes (i.e. each node stores a subset and/or copy of the entries).

The data operations that may be performed on a key-value store include inserting entries, accessing entries, and removing entries, typically referred to as *put*, *get*, and *remove*. These operations require a unique *key* to be associated with each entry when inserted into the store and subsequently used to locate the entry for access or removal. Locating an entry based on its key is very efficient, while iterating or searching through all the entries is not, thus the way in which data is aggregated into entries dictates the way in which it may be efficiently accessed and thus the overall performance of a key-value store. The structure of the data exchanged both between components and between the data component and web services is a value set that consists of a list of real numbers representing the state of a variable at a point in time over a set of locations. As the value set is the unit of aggregation of data exchanged, storing each value set as an entry in the key-value store aligns with the way in which the data is accessed by the data component.

Aggregating data as value sets is not the only possibility, as it would also be possible to aggregate data into larger units such as groups of value sets, or into smaller units such as the individual values that make up a value set (as in Bulatewicz and Andresen (2011)). Storing individual values as entries in the key-value store simplifies the process of assembling value sets ad-hoc from entries in the key-value store as they are requested by model components (to avoid the need to call a web service to obtain them) thus maximizing the reusability and the effectiveness of the cache and resulting in no storage of duplicate data. This also results in higher memory usage per entry as each entry incurs a constant overhead (approximately 260 B) that is approximately the data size of a single value. This means that 50% of memory usage is overhead, and greater processor and network usage results as each entry must be inserted and removed from the key-value store individually. Storing value sets as entries in the key-value store (as in Bulatewicz and Andresen (2012)) minimizes overhead in terms of memory, processor, and network, but introduces the possibility of storing duplicate data in the key-value store in the case that the values stored in two value sets intersect. It also requires a more complex process to assemble value sets ad-hoc (see Section 2.3.1). In our earlier work we found that the overhead of storing individual values as entries limited the scalability of the system and thus in this work we designed the data component to store value sets as entries in the key-value store.

Each entry in the key-value store is a variably-sized object consisting of a variable identifier (string), timestamp (string), element set identifier (string), scenario identifier (string), a delivery flag (boolean), array of values (double precision), and value count (long), that are serialized into an array of bytes. The keys used to access the entries in the store are strings formed by the concatenation of the entry's variable identifier, element set identifier, timestamp, and scenario identifier, for example: TemperatureSewardCounty2013-01-01T12:00:00S01. Using keys of this form guarantees uniqueness and makes it possible to efficiently look up a value set from the key-value store for a specific variable, instantaneous point in time, and element set, for a particular scenario identifier. The scenario identifier provides a

way to partition, version, and identify value sets that are created by different linked models or instances thereof. For example, when executing several instances of a linked model, each instance may be assigned a unique scenario identifier so that the input and output value sets of each are distinct. The delivery flag indicates whether the value set is pending delivery to a web service.

When a value set is delivered to a web service, additional information must be provided that indicates the locations the values represent. This information is not stored inside the entries in the key-value store because all the value sets for a particular element set would result in the storage of duplicate data. As element sets are static during a simulation run there is typically a high ratio of value sets to element sets, so the entries only store the element set identifier and the actual element set information is stored separately in the data store. In this way a data store can look up the complete element set information for any value set before delivering it to a web service.

A number of different key-value store database systems could be utilized as the data store, such as Memcached (Memcached, 2013) or Cassandra (Cassandra, 2013). We chose to utilize the Hazelcast distributed data platform (Ozturk, 2010) because in our previous work we found it to be highly efficient and require minimal configuration. Hazelcast is a clustering, scalable, in-memory data platform that is implemented in Java and distributed as a shared library that we compiled into the data manager program. When the data manager is started it creates an instance of the Hazelcast platform peer that runs as a set of threads inside the data manager process. Instances within different data manager processes dynamically form a cluster by discovering one another via multicast and communicating via TCP/IP. Instances thus join and leave the cluster as data manager processes are started and stopped. Each instance has a local memory that is logically organized into one or more global hashmaps data structures whose entries are distributed across the instances of a cluster and it is these distributed hashmaps that make up the data store. The instance running within a data manager is self-contained and the software modules within the data manager may only put, get, and remove entries (i.e. value sets) to and from the data store as illustrated in Fig. 3.

The instances balance the entries in the data store such that they are evenly distributed among the instances executing on a cluster and each instance has approximately the same number of entries in its local memory. For each entry stored in an instance there are copies of the entry stored in a different instance somewhere in the cluster in case an instance fails (the number of copies is configurable in Hazelcast). When instances leave a cluster its entries are migrated to and distributed among the remaining instances. Each instance optionally persists the entries of its local memory to a file between executions.

The Hazelcast platform supports *native clients* that may access the data store managed by the cluster of instances. A client connects to an instance and that instance executes put, get, and remove operations on the data store on behalf of the client. As clients do not participate in the storage or management of the entries in the data store, they require few computer resources and many clients may connect to a single instance. The native client shared library is compiled into the data component and runs as a set of threads inside the process in which the data component is running, similar to how the instances run within the data

manager processes. Similarly, the data component's engine (which implements the OpenMI and handles the configuration file) has limited interaction with the client and may only instruct the client to connect and disconnect with an instance and put, get, or remove entries. The client is otherwise isolated from the engine and the client threads maintain a direct and persistent network connection to the instance threads. The data component communicates with the data manager through the Hazelcast client-instance connection using two request queues managed by the instance. The component inserts both requests to retrieve value sets from web services and requests to store value sets in the data store into these queues and the data manager and its software modules process the requests.

2.3. Providing input data to models

2.3.1. Caching

During the execution of a composition, several model components within a single composition may request identical value sets from a data component. In addition, model components which are independently executing compositions on different cluster nodes may request the same value sets from different data components. In both cases it is advantageous for the data components to cache the value sets that they retrieve from the web services and to share those value sets across all the data components that are executing simultaneously in different compositions across a cluster. It is also advantageous for the cached value sets to be persisted between executions as the same value sets may be needed on subsequent executions of a composition.

When GetValues is invoked by a model component on a data component, the data component checks to see if the requested value set exists in the data store by creating the appropriate key and then performing a get operation on the data store using the key. If the data component successfully retrieves the value set from the data store then it is returned to the model component and the execution of the composition continues. If the value set is not in the data store then the data component inserts the key into the request queue. After the insertion is completed, the data component periodically checks the data store until the value set is available (during which the execution of the composition is paused). The data component relies on the retrieval module inside the data manager to obtain the requested value set from a web service and insert it into the data store.

The retrieval module waits for a request to be inserted into the request queue. When a request is inserted by a data component, it is removed by the data manager provided that the amount of data in the local data store has not reached the maximum limit (as configured in the data component). The request queue may only hold a single request at a time and causes data components to wait if they attempt to insert a request when there is already a request in the queue. This prevents the data manager from becoming overwhelmed with requests. The data store is checked for the requested value set in case it was already retrieved while the data component was waiting to insert the request. If it is not, the retrieval module attempts to assemble the requested value set from other value sets that are already in the data store, as explained below.

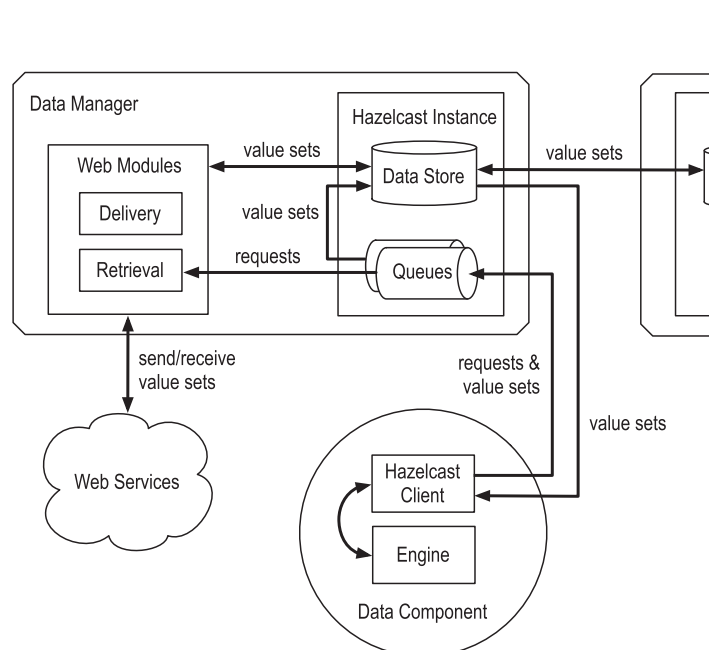


Fig. 3. Interactions between software modules. Arrows indicate the direction of data movement.

The element set of a requested value set may intersect with the element sets of other value sets in the data store. As such, it may be possible to assemble the requested value set by extracting the necessary values from other value sets already in the data store whose element sets intersect with the element set of the requested value set. This maximizes the reusability of the cached data and minimizes the number of web service calls.

The following algorithm is utilized by the retrieval module to assemble value sets in such a way as to minimize the number of get operations performed on the data store. Each element of each element set is compared to the requested element set to determine whether the elements in the requested element set exist in other element sets. If all the elements in the requested element set can be found in other element sets, then the value set map is checked for each source element set to see if a value set for the requested time exists. If it does then the required values are collected from it. If all the values in the requested value set are found then the assembled value set is inserted as a new entry into the data store. This requires one get operation per source element set. In the case of a requested value set whose element set is a subset of another element set whose data is in the value set map, it would require one get operation to obtain the necessary data to assemble the value set. The maximum number of get operations that may be necessary is equal to the size of the value set being requested, which occurs in the case that each value is sourced from a different element set. The amount of time required for assembly varies considerably according to the size of the element set, the degree of overlap with existing element sets, the number of value sets in the cache, and the number of data stores that they are spread across.

If a value set cannot be assembled from the values already in the data store, a web service call task is created for the request and added to a thread pool. Each task generates the appropriate web service request XML, calls the web service, and then parses the response into a value set that is inserted into the data store, as shown in Fig. 4. Multiple web service calls are issued simultaneously in a pipelined fashion to take advantage of multi-core and multi-host web services. The retrieval module limits the number of simultaneous web service calls to the number of connected data components. This limit is necessary because data components may request value sets ahead-of-time (prefetch) which could result in the creation of so many threads that the system resources become exhausted.

2.3.2. Prefetching

The simulation of physical processes (especially those for which the OpenMI is initially designed) typically involves the calculation of output quantities over a simulation time period. A component typically steps forward through simulation time requesting value sets from the data component on each step. To avoid causing a model component to wait for a value set while the data component is retrieving it from a web service, the data component retrieves value sets before they are requested, a technique called *prefetching*.

This notion of prefetching data stands somewhat in contrast to the pull-based execution style of the OpenMI in which computation is only performed as needed. Although the pull-based execution style is efficient in that computation is only performed on demand, the concept of prefetching could be broadly applied to model components in general as a means to reduce simulation runtime through parallel execution. Models could precompute results, perhaps based on estimated or anticipated inputs, while other components are executing and the results immediately returned when requested if the estimation was close enough.

Throughout the execution of a composition the components are at approximately the same point in simulation time. This is because each component typically requires input data from the other components that reflect its current simulation time, causing those components to advance to the same point in simulation time. For this reason, all components should be 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 data component to obtain this information directly from model components, as the OpenMI does not support this functionality.

The data component predicts what value sets will be requested in the future by observing what value sets have been requested in the past. Components that use a fixed-length time step request data from the data component at fixed intervals making it possible to identify these components and determine the length of their time steps. In such cases the data component can accurately predict the value sets that will be requested in the future. It is more difficult for the data component to predict the data needs of components that use a variable-length time step and is not addressed in this work. The data component prefetches all links to a common future point in simulation time (number of Julian days) given by: $t = \min(p + i, e)$ where p is the earliest time to which all links have been prefetched, i is the longest request interval (i.e. longest time step) across all links, and e is the ending time of the composition. When prefetching data, it is possible that the requests by the actively executing model component are delayed due to requests made by the waiting components, but that delayed time is accounted for when/if those components use that data.

2.4. Delivering output data to web services

The input exchange items of the data component may be linked to one or more model components within a composition. At initialization, the data component registers to be notified via a *DataChanged* event whenever a model component produces an output value set along any of its input links, which is typically raised after each time step. When the data component receives this notification it invokes the *GetValues* function on the model component to obtain a copy of the value set as shown in Fig. 5.

The data component instructs the Hazelcast client to insert the value set into a data queue within the Hazelcast instance that the client is connected to. This queue can only hold a single value set at a time so if a value set is in the queue, then additional insert attempts will wait until the value set is removed, causing the data component to wait, and in turn causing the model component to wait. The queue serves as a gate to prevent too much data from being inserted into the data store, which would be possible if the client inserted value sets directly into the data store. Whenever a value set is added to the queue, the data manager checks if there is available space in the local data store and if so moves the value set into the data store and sets a flag within the value set that indicates it is pending delivery to a web service. The amount of memory dedicated to the local data store is configurable via the data store configuration file and must be equivalent among all connected data stores (as required by Hazelcast).

The delivery module periodically searches for value sets pending delivery and if there is a sufficient amount of data to be sent such that network resources will be utilized efficiently, then the value sets are sent to the web service. The amount of data that is sent in each web service call is configured in the data store as a number of bytes, called the *delivery size*. The data component estimates the number of value sets to include in each web service call by estimating the size of an encoded value set (as XML) via a constant per-value multiplier specific to each web service.

The periodic search performed by the delivery module is efficient in that only a single pass through the entries in the local data store is necessary. During this pass, entries that are pending delivery are copied into a priority queue ordered by creation date. If the encoded size of the value sets in the queue is greater than the delivery size then a thread pool task is created that serializes the value sets into the appropriate XML encoding and calls the web service. The delivery module ensures that it delivers at least as many entries as it collects before allowing the data manager to shut down to ensure that all managers cooperate fairly in the delivery of data.

The delivery size provides a means for both the regulation of network efficiency and the control of the delay between the collection of a value set and its delivery to a web service. The delivery manager attempts to remove enough value sets from the buffer to meet the 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

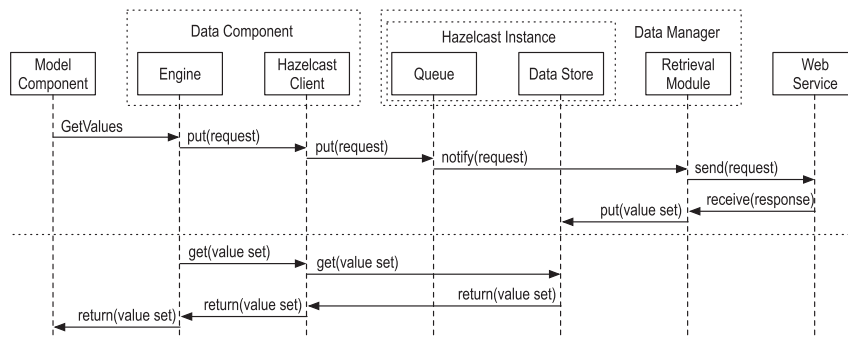


Fig. 4. Sequence diagram of interactions involved in providing data from web services.

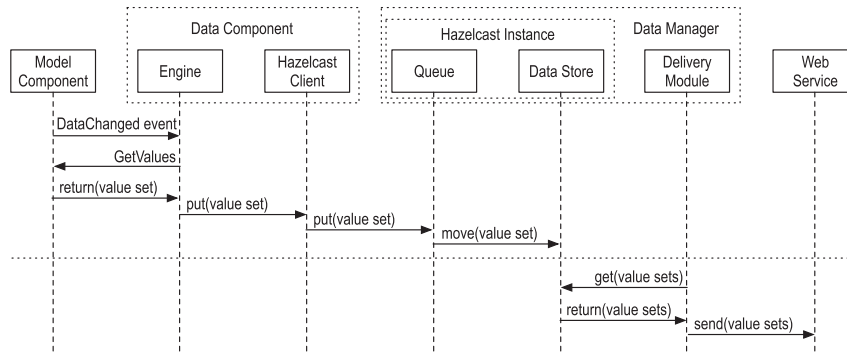


Fig. 5. Sequence diagram of interactions involved in delivering data to web services.

consumed as the simulation is being carried out it may be necessary to minimize the duration that an entry may reside in the buffer before it is delivered, at the expense of efficient network utilization. By setting the delivery size to zero, value sets are sent individually as quickly as possible. Note that the delivery module does not attempt to send more value sets than necessary to meet the delivery size because this would require additional parameterization of a maximum, since data may be inserted into the data store at a rate that is faster than the delivery module is able to remove it, preventing the delivery of data.

The amount of memory available to the data store is finite and once exhausted the operation of the data store will pause, since it relies on the ability to store data. The effectiveness of the cache increases with the amount of data in the cache, so it is beneficial to allow the data store to be filled as much as possible without impeding the basic operation of the data store. For these reasons a thread within the data manager periodically checks the local buffer and if the data size is greater than 90% of the maximum size, then an eviction is performed. The least accessed 15% of the entries are removed from the buffer that (1) have been sent, or (2) been accessed at least once, or (3) have been downloaded and cached. This prevents cached data, which may or may not be used in the future, from preventing data retrieval from web services or collection from components. In the case that the data store is mostly full of unsent data, downloaded data is purged leaving only the unsent data and if no memory is available, a request will remain in the data queue blocking data components from adding more data requests. As data is delivered to the web services, sent data will be purged and data components will again be able to insert requests into the data queue.

The data component does not provide a means to update the data in the cache, for example, by re-downloading value sets at a prescribed interval because the intended use case is the retrieval of reference data that do not change or do so infrequently. Scenarios in which the data changes frequently would be managed by the user, for example, by disabling persistence so that the cache is empty at the start of each simulation run resulting in new data being downloaded, or by changing the size of the cache to indirectly control the frequency at which data are re-downloaded, since data are expired when the available memory becomes low causing them to be re-downloaded the next time they are requested.

2.5. Adding support for web services

The initial implementation supports WaterOneFlow web services and a simple REST web service used in the performance study. The data manager may be extended to support additional web services through the creation of adapter classes that serve as intermediaries between the fetch/delivery modules and the web services. An adapter class conforms to the *ServiceAdapter* interface defined by the data manager that consists of two primary methods that are responsible for sending a set of value sets to a web service and retrieving a value set from a web service. We envision the development of a complete plugin architecture in which web service adapters can be developed independently and loaded into the data manager at runtime.

3. Performance study

To evaluate the scalability and efficiency of the data component we measured a set of performance metrics for varying numbers of linked models simultaneously executing across a compute cluster.

3.1. Baseline configuration

We created a composition that includes a data component linked to two model components such that the data component

provides input to a *consumer* component and collects output from a *producer* component as illustrated in Fig. 6. The producer and consumer components serve as placeholders for model components and although they are capable of accepting and providing exchange items and advancing through simulation time, they do not perform any calculations but rather pause for 1 s on each time step, which we refer to as the *time step calculation time*. They discard the data they receive as input and produce constant-valued data as output. We configured the components to advance their simulation time by one day on each time step and configured the composition for a time horizon of 7 months, thus each component performs 212 time steps in each simulation. We empirically determined that using a higher number of time steps does not impact the performance results. We configured the data component to deliver the output from the producer component to a web service hosted within the compute cluster and provide input to the consumer from the service. The data component thus provides 212 value sets to the consumer component and collects 212 value sets from the producer component during the execution of a single instance of the composition, which we refer to as a *simulation*.

For the purpose of the performance study, in which a large number of simulations were necessary and consumed a significant amount of time, it was advantageous to use a small element set. It was also important that it be of a reasonable size that an environmental model may use in practice. For these reasons the components exchange value sets that correspond to an element set of 1000 elements, which allowed us to run the necessary number of experimental simulations within the necessary timeframe while also be representative of some environmental models, such as regional watershed models that operate at coarse resolutions for which elements represent polygons.

We used a fixed pause duration to minimize fluctuations in the number of data components waiting on the request queue of the

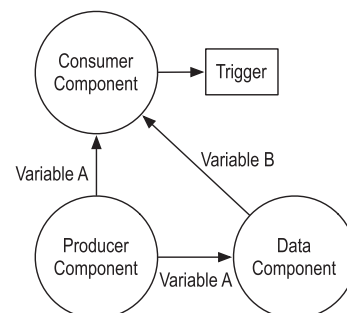


Fig. 6. The composition used in the performance study. Arrows indicate the direction of data transfer between the components.

data managers. This allows for more accurate performance measurement because it avoids arbitrary delays due to the queue being empty at times and very long at other times that may result when the request frequency of model components is variable. The greater the variance in the request frequency the greater the variance in the number of data components waiting on the request queue and thus regular request frequencies result in minimum runtimes.

A simulation begins when the trigger invokes *GetValues* on the consumer component for its starting simulation time. The consumer component in turn calls *GetValues* on both the producer component (which advances its time) and data component which return the requested value sets to the consumer component which then advances its time. The generation of the value set by the producer raises a *DataChanged* event which causes the data component to call *GetValues* on the producer component to obtain the new data. The trigger repeatedly invokes *GetValues* on the consumer until the simulation time of the consumer reaches the configured end time. We refer to the duration of time that an instance of the composition spends executing as the *simulation runtime*. As the components perform time steps sequentially and pause for 1 s on each step the simulation runtime of the baseline configuration would be 424 s if the data component and exchanges between the components incur zero time.

3.2. Execution environment

We executed the simulations on an onsite Linux-based Beowulf compute cluster for the performance study. To limit variability in the results due to differences between the hardware specifications of the compute nodes we utilized a single class of machines that had dual 8-core Intel Xeon E5-2690 processors with 64 GB of memory and were connected via gigabit Ethernet. Of the 133 nodes and 2066 cores in the cluster, this machine class accounted for 34 nodes and 544 cores. A virtualized Windows-based server with a 4-core 2.7 GHz processor and 8 GB of memory hosted the web services and was connected to the compute nodes via gigabit Ethernet.

Access to the cluster nodes is provided via a job scheduler (Sun Grid Engine) to which requests are made for resources (number of processor cores, amount of memory, and maximum runtime) and when they become available a set of scripts provision the nodes as necessary and then execute a set of simulations. The job scheduler executed each set of simulations on multiple nodes utilizing an average of approximately 5 cores on each node. We configured the job scheduler to reserve one core for each data manager and one core for every 4 simulations. Scheduling several simulations on each core made it possible to execute a greater number of simulations than there were cores. We verified that collocating several simulations on a single core did not affect the performance results in our experimental configuration (probably because the producer and consumer components require few computer resources).

The components are implemented in the C# programming language based on the OpenMI 1.4 software development kit and were executed using the OmiEd application (via the command line). We chose version 1.4 of the OpenMI because the model components in our case study rely on libraries based on this version (Bulatewicz et al., 2013; Castronova and Goodall, 2010) although we are actively developing an implementation of the data component for version 2.0 of the OpenMI as well. The data manager is implemented in the Java programming language because this is the language that Hazelcast is implemented in. The web service is implemented in the PHP programming language and is hosted by the Apache HTTP server. We developed a custom REST-based web service and minimal XML data encoding to avoid any bias that a more complex encoding may have on the results. The web service parses the XML in each request and returns constant-valued data in

its response. We configured the data manager such that the backup feature of Hazelcast was disabled so that this feature would not affect our performance measurements.

The delivery size that maximizes throughput when data is sent from the data store 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 determine an appropriate delivery size for our experimental configuration. We found that the maximum throughput between a benchmark application and the web service was 47 MB/s when at least 50 MB of data was sent. As the XML serialization of a 1000-element value set requires 49 KB, the data store would have to send 1498 value sets in each web service call to achieve maximum throughput. If value sets were collected at a rate of 1 per second then they would be delivered every 25 min. To increase the rate at which value sets were delivered to the web service while still maintaining good network efficiency we decided to use a delivery size of 11 MB which achieves 50% of the maximum network throughput.

3.3. Scalability

To verify that the design of the data component and data manager is efficient and capable of high performance when there are large numbers of data components we measured the average simulation runtime for varying numbers of simulations and data managers. Each simulation used a unique scenario identifier so that its input and output value sets were distinct. The results are presented in Fig. 7 (top). For a given number of data managers, the average simulation runtime increased as the number of simulations increased. This is because the data component pauses a simulation while it is waiting for the data manager to process its requests. When all simulations were supported by a single data manager the rate at which the average simulation runtime increased was most closely described by the function $0.0003 \times (n^{1.5})$ where n is the number of simulations ($R^2 = 0.996$). In the case of 4 data managers the rate at which the runtime increased was most closely described by the function $0.4 \times e^{0.001 \times n}$ ($R^2 = 0.994$). Based on these functions we expect the average simulation runtime to increase at a greater rate when there are more than 1000 simulations.

The number of data managers supporting the simulations had a varying impact on the average simulation runtime. Increasing the number of data managers from 1 to 4 significantly reduced the average simulation runtime, while increasing further to 8 only resulted in a small reduction for higher numbers of simulations. Increasing the number of data managers further to 16 slightly increased the average simulation runtime due to the additional overhead incurred by the management of the distributed data store. We therefore estimate that the ideal ratio for our experimental configuration was approximately 1 data manager per 250 simulations.

The duration of time between when a value set is collected by a data component and when it is delivered to the web service, the *delivery time*, is a function of (1) the rate at which value sets are collected by the data components connected to a data manager, (2) the size of the value sets, and (3) the delivery size that the data manager is configured to use. For our experimental configuration the size of a value set with 1000 elements encoded in XML was 49 KB and the delivery size was 11 MB, so the data manager only sent data to the web service when it found 230 unsent value sets in its local data store. For low numbers of simulations the rate at which value sets were collected and added to the data store was low, resulting in the data manager delaying the sending of the data, as shown in Fig. 7 (bottom-left). Higher numbers of data managers amplified this effect, further reducing the rate at which value sets

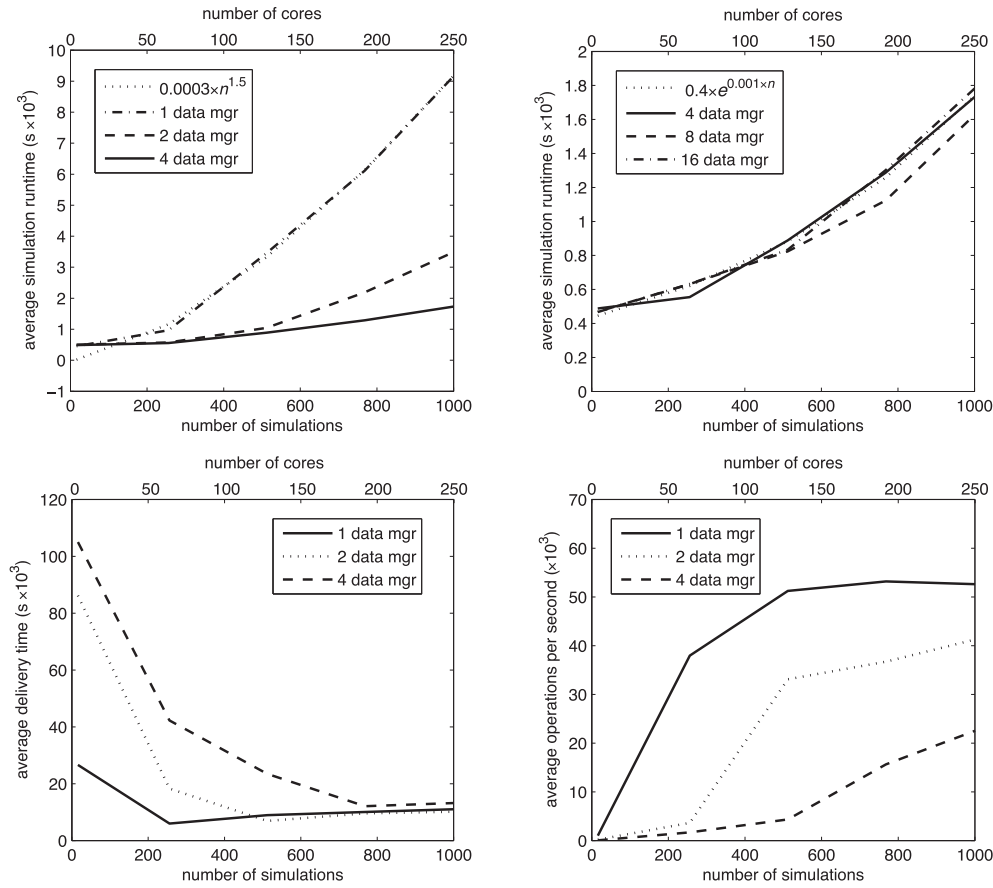


Fig. 7. Scalability results.

were added to each data manager and further increasing the delivery time. For high numbers of simulations, the delivery time was low due to the faster rate at which value sets were collected by data components and added to the data store, which ensured there was always a sufficient number of unsent value sets. In this case the delivery time increased slightly as the number of simulations increased because greater numbers of value sets in the data store increased the search time for locating unsent value sets.

The average number of operations per second (put, get, and remove) performed by Hazelcast on the hashmap that stores the value sets increased with the number of compositions as shown in Fig. 7 (bottom-right). The maximum number of operations per second reached in our experimental configuration was approximately 50,000 when a single data manager was used and was significantly less for greater numbers of data managers.

3.4. Caching

Caching is most effective when the input value sets needed by the simulations are in the data store prior to when they are requested, and the time required to retrieve a value set from a web service is significant. It is least effective when the needed value sets are not in the data store or when they may be quickly retrieved from the web services. To demonstrate these two extremes we compared the baseline scenario, in which caching has little effect, with an alternate scenario in which caching is highly effective. In the latter case one simulation executed before the others to populate the data store and the web service waited 3 s before responding to each request for a value set. We measured the performance of both scenarios with and without caching (simulations used either a common scenario

identifier so that they requested identical data or distinct identifiers so that they requested different data, respectively) in a 16-simulation 1-data manager configuration.

The average simulation runtime for the alternate scenario in the “no caching” case was approximately 2.5 times higher than in the baseline scenario due to the additional 3 s delay incurred by each of the 212 web service requests, as shown in Fig. 8. For the baseline scenario the average simulation runtime was similar in both the “caching” and “no caching” cases because the retrieval time (i.e. the time spent generating a request, calling a web service, and processing a response) was very low. For the alternate scenario, however, the average simulation runtime in the “caching” case was 59.8% lower than in the “no caching” case because the retrieval time was higher (approximately 3 s). In both scenarios the amount of data retrieved from the web service in the “caching” case was 93.8% lower than in the “no caching” case because the simulations requested identical input data and thus only 1/16th the amount of data was retrieved from the web service. In general, the amount by which the average simulation runtime can be reduced is the percentage of the runtime that is due to the retrieval of the data. The amount by which the data transfer can be reduced is a function of the size of the value set the number retrieved.

These two scenarios represent the extremes but in practice it would probably be the case that a portion of the value sets exist in the cache and a portion are retrieved from the web services. In such a case, the performance trend would be linear (provided that the value set sizes and web service retrieval times remain consistent). For example, if exactly half the value sets were in the cache, then the amount of data transferred and the duration of time spent retrieving data would be half that of the extreme case in which

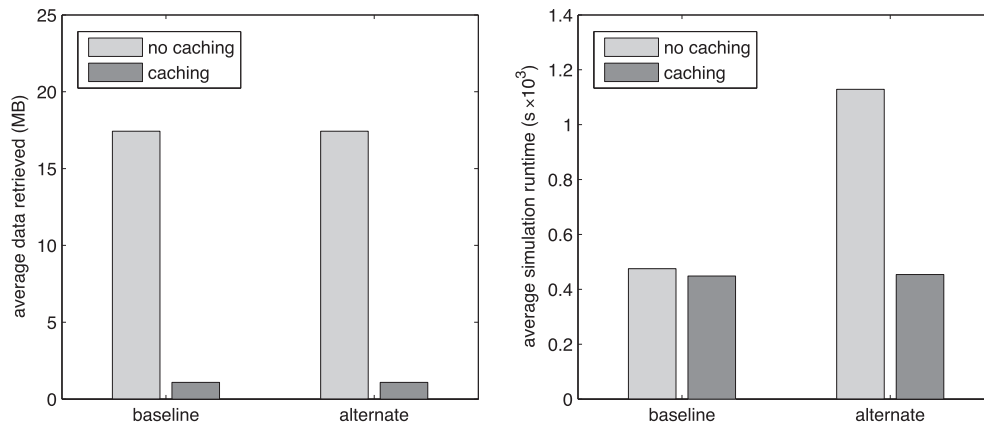


Fig. 8. Caching results.

there is no overlap and all data must be retrieved from the web services.

3.5. Prefetching

Prefetching is most effective when there is sufficient time during the components' time step calculations to retrieve data from the web services. It is least effective when time step calculation times are short compared to the retrieval time. To demonstrate this we compared the baseline scenario, for which prefetching has a small impact, to an alternate scenario in which prefetching has a large impact. In the latter case we configured the time step calculation time to be the same as the web service response time (i.e. the time between receiving a request and sending a response), allowing the retrieval of value sets to take place while the components (producer and consumer) are calculating their time steps. We measured the performance of both scenarios with and without prefetching enabled in 16- and 256-simulation configurations with a single data manager.

As shown in Fig. 9, prefetching reduced the average simulation runtime in all cases because some portion of the data retrieval could be performed during the time step calculation rather than waiting until each time step was calculated before requesting the data. In the baseline scenario with 16 simulations, prefetching resulted in a small decrease in the average simulation runtime (3.4%) because the retrieval time was small and hence only a small amount of time was saved by performing the retrieval during the time step calculation. In the alternate scenario with 16 simulations, prefetching resulted in a large decrease in the runtime (43.0%)

because the retrieval time was high (due to the increased web service response time) and thus a significant amount of time was saved by performing the retrieval during the time step calculation. In the case of 256 simulations, the reduction in the average simulation runtime was less in both scenarios due to the number of prefetch failures that were a result of the data component's prioritization of requests for data over requests for prefetching data.

The reduction in the average simulation runtime possible by prefetching is thus a function of the relative difference between the retrieval time and the length of time between subsequent requests made to the data component (i.e. the sum of the calculation times of all components for a time step). In cases in which the retrieval time is less than the total amount of time the components spend calculating a time step, the runtime of the simulation is not affected by the web service calls and is masked by the time step calculation time (assuming the data manager does not reject the prefetch requests). For example, in cases in which a model is performing an intensive and time-consuming computation on each time step, such as calculating an analytic solution, and data retrieval is relatively quick, such as when retrieving data of generally small element set sizes from a web service located within on-premise network. Prefetching is less effective when models compute time steps quickly, such as statistical regression model, or when retrieving data takes a relatively long time, such as when retrieving data from servers located overseas or when data consists of large element sets. In general, the amount by which the average simulation runtime can be reduced is the percentage of the runtime that is due to the retrieval of the data (i.e. the retrieval time). In cases in which the retrieval time is greater than the time spent calculating

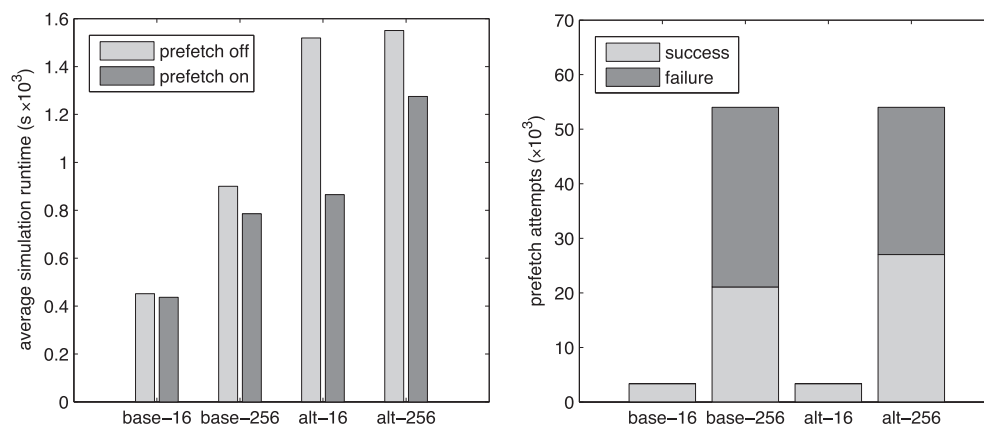


Fig. 9. Prefetching results.

time steps, the average simulation runtime will increase proportionally according to the relative difference between them.

4. Case study: a groundwater sustainability challenge

To demonstrate how the data component may be incorporated into a modeling study we present how we are utilizing it in an ongoing case study to provide input data from an online database to the components of a linked model executing on a desktop computer. Whereas the performance study demonstrated the efficiency of the data component, this section focuses on how it can be applied in practice.

4.1. Study area

The communities of western Kansas in the Central Plains of the United States have relied upon the availability of groundwater for irrigated agriculture for 50 years (Fig. 10). The rate at which water is extracted from the Ogallala Aquifer underlying the region has exceeded the rate at which it naturally recharges, resulting in a gradual decline in the volume of water stored in the aquifer. In some areas it is no longer possible to extract water from the aquifer due to the decreased saturated thickness, a trend that will continue to spread throughout the region unless agricultural practices transition to sustainable rates of water consumption. As this transition impacts the closely intertwined economy and ecology of the region it is essential that it be guided by multidisciplinary integrated assessment.

We consider the relevant natural and human processes in this system to be (1) the movement and volume of groundwater, (2) the choice of crop planted, and (3) the growth of the plants. Building on previous experience in integrated modeling for irrigated agriculture (Bulatewicz et al., 2010, 2013) we have developed three new model components that simulate these processes and have created a prototype linked model integrating them.

4.2. Model components

The crop choice component is an iterative Positive Mathematical Programming (PMP) model (Howitt, 1995) that simulates farmers' allocation of arable land to different crops. The model operates on an annual time step, with each execution predicting farmers' choices in a single growing season. In addition to harvested crop prices and crop-specific costs of production, the model accepts as inputs the current depth to water and saturated thickness of the aquifer. Depth to water affects water extraction costs, while saturated thickness affects the pumping rate of wells, which in turn creates an upper bound on the annual extraction of irrigation water. The model simulates land allocations as the solution to a constrained optimization problem that represents farmers profit-

maximizing a mix of land uses, given price conditions, water extraction costs, and the constraints on water and land availability. It operates over a set of (independent) polygons of variable size (determined by the resolution of available calibration data). The component accepts inputs for saturated thickness and depth-to-water and provides outputs of acres planted for various combinations of crop (wheat, corn, sorghum, soybeans, alfalfa) and irrigation practice (irrigated or non-irrigated). Details on the model development, calibration, and data sources are in Clark (2008) and Garay et al. (July 2010). The model is implemented in MATLAB and interoperability with the OpenMI is provided by the Simple Script Wrapper (Bulatewicz et al., 2013).

The groundwater model provides the groundwater elevation (head) as a function of space and time. For this application, we have developed an OpenMI component for the Hydrologic Response Function (HRF) approach from Steward et al. (2009). Briefly, the aquifer is treated as a sloping base with rectangular cells used to gather pumped water-use within cells that contain uniform aquifer properties (Steward, 2007). Our OpenMI code fully implements the HRF equations and enables the drawdown associated with pumping to be communicated with neighboring cells. This approach was chosen as it has been shown to accurately reproduce the cones of depression formed by groups of wells in the study area (Steward et al., 2009), and the code executes much faster than other approaches based upon the Analytic Element Method (Steward et al., 2008) and finite gridded domain approaches (Steward and Allen, 2013). We also incorporated the groundwater added to the domain through leakage from surface water identified by Ahring and Steward (2012). This was accomplished by adding recharge to cells that coincide with rivers and adjusting the recharge rates until groundwater surfaces matched observations (see Steward et al. (2009) for a discussion of these recharge volumes). The component accepts inputs for irrigated water-use and provides outputs for saturated thickness and depth to water. The model is implemented in Scilab (INRIA) and interoperability with the OpenMI is provided by the Simple Script Wrapper.

The crop production component provides crop yield and irrigated water use data as simulated by the Erosion-Productivity Impact Calculator (EPIC) model (Williams, 1995). EPIC is a process-based generalized crop model that simulates daily crop growth by predicting plant biomass through the simulation of carbon fixation by photosynthesis, maintenance respiration, and growth respiration. In a previous effort we calibrated the model and created a wrapper component that executed the original executable program on-demand (Bulatewicz et al., 2009). In this work we took an alternative approach to model reuse in which we executed the original program for all combinations (2500) of the relevant model inputs (soil, crop, management, weather) and embedded the simulated output data in a component that provides them to other components. The component operates over a set of (independent) polygons of variable size and accepts inputs for soil type and precipitation and provides outputs for yield and water-use. It is implemented in C# using the Simple Model Wrapper (Castronova and Goodall, 2010).

4.3. Linked model design

We defined an element set consisting of 125 grid cells (10,000 m by 1000 m) overlying Seward County in southwestern Kansas (note that the crop choice model internally aggregated the cells to a single homogeneous unit). This set was used for all 13 links in the composition (Fig. 11). The components operate on an annual time step where each year of the simulation period begins when the crop production component requests the crop acreages from the crop choice component and the precipitation and soil type from the data

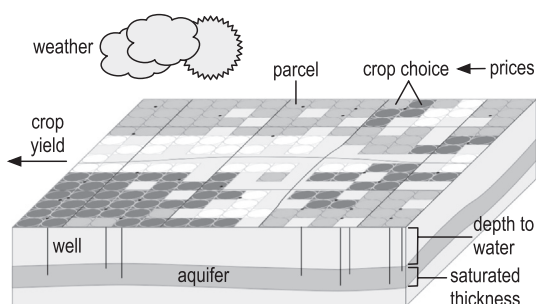


Fig. 10. Conceptualization of an irrigated agricultural system.

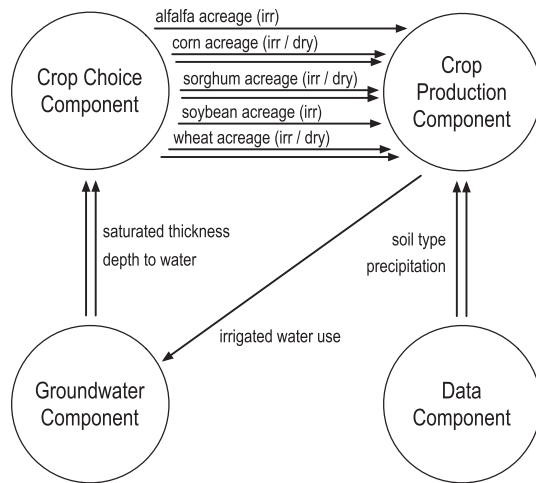


Fig. 11. Component linkages. Data is transferred in the direction of the arrows.

component. The data component retrieves the data from an online database and provides it to the crop production component. The crop choice component requests the saturated thickness and depth to water from the groundwater component (for the previous year), which in turn requests the water-use from the crop production component (also for the previous year). After receiving the response, the groundwater component calculates the saturated thickness and depth to water and provides them to the crop choice component, which in turn calculates the crop acreages and provides them to the crop production component. The crop production component then calculates the crop yield and irrigated water-use.

4.4. Using the data component

To create the linked model we began by adding the 3 models to a new composition using the OmiEd application and then added the appropriate links between them. We then configured the data component by (1) defining the necessary output exchange items, and (2) specifying the information about the web service from which they should be retrieved. The exchange items and web service information are defined within the data component's configuration file as shown in Fig. 12. The format of the configuration file is based on that of the Simple Model Wrapper and was extended to include web service information. The element set and variable of each exchange item (as well as the units information, not shown in the figure) is listed in the configuration file as well as the type, URL, and list of quantities provided and accepted by each web service. The variable ID specified in each output exchange item must appear in the list of RetrievableQuantities for one of the web services and each input item must appear in the DeliverableQuantities After creating the configuration file we added the data component to the composition and added links from each of its output exchange items to the appropriate input of the crop production component.

The URL specified in the configuration file is that of a CUAHSI HIS WaterOneFlow web service that was hosted on a virtualized server (running Windows) that we setup on the cluster network and was publicly accessible via the Internet. The web service (implemented in PHP) connected to a MySQL database that was also hosted on the server and used the Observations Data Model (Horsburgh et al., 2008), which is a relational data model for the storage and retrieval of time series hydrologic observations and associated metadata. The data component provides interoperability between the ODM/WaterOneFlow web service and the OpenMI by mapping their respective data models to one another in a similar way as Castronova et al. (2013b) (e.g. mapping quantities to variables and

```
<Configuration>
<ExchangeItems>
  <OutputExchangeItem>
    <ElementSetID>SewardCountyCells</ElementSetID>
    <Quantity><ID>ODM:Precipitation</ID></Quantity>
  </OutputExchangeItem>
</ExchangeItems>
<TimeHorizon>
  <StartDateTime>2012-01-01T00:00:00</StartDateTime>
  <EndDateTime>2040-08-01T00:00:00</EndDateTime>
  <TimeStepInSeconds>86400</TimeStepInSeconds>
</TimeHorizon>
<WebServices>
  <WebService>
    <Type>WaterOneFlow</Type>
    <Url>http://host/Baseline/Service_10.asmx</Url>
    <RetrievableQuantities>
      <QuantityID>ODM:Precipitation</QuantityID>
    </RetrievableQuantities>
  </WebService>
</WebServices>
</Configuration>
```

Fig. 12. The data component configuration file (partial).

sites to elements). Thus, the IDs of the elements within the element sets of the input and output exchange items specified in the configuration file must exist as sites in the database (mapped to SiteCode) and the variable IDs of the exchange items must exist as variables in the database (mapped to VariableName). The WaterOneFlow web service returns data as time series whereas exchanges between OpenMI components are space series, requiring the data component to make a separate web service call for each element on each time step.

The output of the linked model simulation for 3 indicators is shown in Fig. 13. The county-wide total crop yield and irrigated water use varied from year to year according to the weather while the saturated thickness of the aquifer decreased at a constant rate. The data component was executed on a desktop computer (along with a data manager) and communicated with the web service via the Internet over a distance of 2266 km. The first time the simulation was executed, the data component retrieved 111.9 MB of data and spent 14.8 m retrieving data (which was 42.2% of the total simulation time) over the course of the 100 time steps. On subsequent executions, the data component retrieved all data from the data manager and the web service was not called.

5. Conclusions

We have presented the design of a general-purpose data component for the OpenMI, evaluated its performance, and demonstrated its application in a modeling study. The data component can mitigate data management challenges in modeling and simulation by serving as a bridge between model components and online services minimizing the reliance on data files and ad-hoc scripting. We adapted three techniques to the unique design of the OpenMI to enable efficient operation: caching, prefetching, and buffering, making it suitable for use on both desktop computers and high-performance compute clusters.

The data component is added to a composition and linked to model components in same way that model components are linked to one another. The scientist configures, and re-configures the data component for the input and output exchange items necessary for any given set of model components based on the data available via web services. It relies on a data manager program that communicates with web services and manages a distributed data store

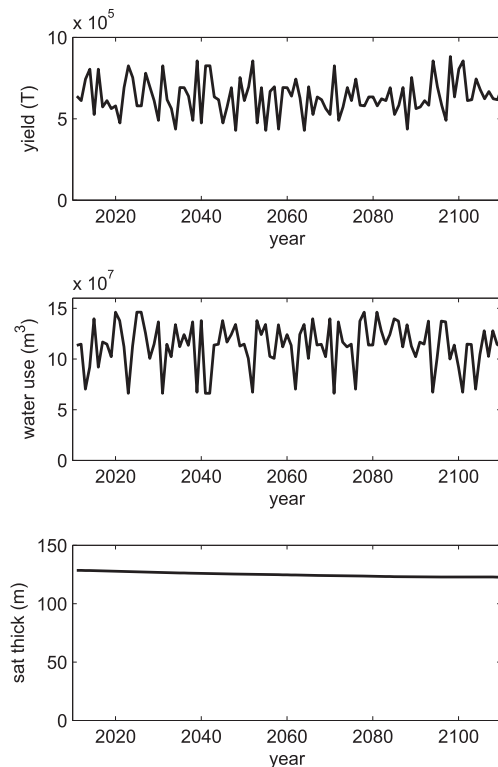


Fig. 13. Output from the linked model for 3 indicators.

shared across all the data managers executing on a compute cluster. The data retrieved from web services is cached in the data store and the data collected from model components is buffered in the data store before being delivered to web services.

We evaluated the performance of the data component in terms of scalability and the effectiveness of caching and prefetching in minimizing the simulation runtime. The results are summarized in Table 1. The increase in simulation runtime incurred by the data component (as compared to using local data files) ranged from 0.14 s for 16 simulations to 3.78 s for 1000 simulations for each time step. The data transferred to and from the web service was 131 KB per time step for a value set of 1000 values.

Caching can have a significant impact on the runtime of simulation in some cases and little or no impact in other cases. We demonstrated this via two configurations which resulted in a 1%–29% reduction in the average simulation runtime. This range only serves as an example of possible performance, as the actual impact is a direct result of the retrieval time and the number of times model components request identical data.

Prefetching can also have a significant impact on the runtime of a simulation, but through different means than caching. Prefetching is only effective when the time step processing time of a model component is comparable to the retrieval time thus making it possible to overlap the model execution with the retrieval of data. We demonstrated this via two configurations in which the runtime was reduced by only 3% when there was no overlap and 43% when

there was full overlap. In addition, prefetching is less effective when a data manager is under high utilization.

Buffering always reduces the runtime of a simulation where the reduction is directly proportional to the web service response time. Although the impact of buffering on the simulation runtime cannot be measured empirically (because buffering is inherent in the design of the data manager) its impact can be estimated by adding the time spent sending the data on each time step. For the experimental configuration, in which the model components spend 2 s processing each time step, if the time spent sending data on each time step was 0.1 s then the reduction in runtime due to buffering would be 5% whereas if the time spent sending data was 1.0 s then the reduction would be 33%.

Based on the results of the performance study, it can be expected that the simulation runtime will increase as the number of simulations is increased, and that buffering always results in improved runtimes while caching and prefetching may result in improvements depending upon the situation. Overall, the runtime overhead of the data component is primarily determined by the web service response time and to a lesser degree the time step processing time of the model components and the value set size (as the data transfer size and parsing time are influenced by it). As the web service response time increases, the runtime increase incurred by the data component becomes larger while at the same time the benefit of buffering and the potential benefit of caching and prefetching increase as well. In general, the percentage of the runtime that is due to the web service calls is equivalent to the reduction that would be achieved in cases in which caching and prefetching are effective.

We therefore conclude that the design of the data component meets the three requirements identified in Section 2. Standards for web services make it possible for the component to be configured and reconfigured as necessary to meet the needs of different linked model configurations and different web services. The increase in simulation runtime incurred by the data component (as compared to using local data files) is reasonable and in some cases can be eliminated by caching and prefetching data. The overall performance of the data component is reasonable for large numbers of simultaneous simulations.

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

Acknowledgments

This work was supported by the National Science Foundation (grants GEO0909515, EPS0919443, EPS1006860, CNS1126709) and the Ogallala Aquifer Project of the USDA/ARS. Access to the Beocat compute cluster at the Dept. of Computing and Information Sciences at Kansas State University was appreciated. Any findings, opinions, conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of any funding units.

References

- Ahring, T.S., Steward, D.R., 2012. Groundwater surface water interactions and the role of phreatophytes in identifying recharge zones. *Hydrol. Earth Syst. Sci.* 16, 4133–4142.
- Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., Mock, S., 2004. Kepler: an extensible system for design and execution of scientific workflows. In: *Scientific and Statistical Database Management*, 2004. Proceedings. 16th International Conference on, pp. 423–424.
- Bulatewicz, T., Andresen, D., 2011. Efficient data access for Open Modeling Interface (OpenMI) components. In: Arabnia, H.R. (Ed.), *Proceedings of the International*

Table 1
Summary of performance study results.

Technique	Improvement	Dimension	Cost per time step
Caching	1%–29%	Time	0.14 s–3.78 s
Prefetching	3%–43%	Data	131 KB
Buffering ^a	5%–33%		

^a Estimated.

- Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), vol. 1. CSREA Press, Las Vegas, Nevada, USA, pp. 822–828. July 18–21.
- Bulatewicz, T., Andresen, D., 2012. Efficient data collection from Open Modeling Interface (OpenMI) components. In: Arabnia, H.R. (Ed.), Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), vol. 1. CSREA Press, Las Vegas, Nevada, USA, pp. 53–59. July 16–19.
- Bulatewicz, T., Jin, W., Staggenborg, S., Lauwo, S.Y., Miller, M., Das, S., Andresen, D., Peterson, J., Steward, D.R., Welch, S.M., 2009. Calibration of a crop model to irrigated water use using a genetic algorithm. *Hydrol. Earth Syst. Sci.* 13, 1467–1483.
- Bulatewicz, T., Yang, X., Peterson, J.M., Staggenborg, S., Welch, S.M., Steward, D.R., 2010. Accessible integration of agriculture, groundwater, and economic models using the Open Modeling Interface (OpenMI): methodology and initial results. *Hydrol. Earth Syst. Sci.* 14 (3), 521–534.
- Bulatewicz, T., Allen, A., Peterson, J.M., Staggenborg, S., Welch, S.M., Steward, D.R., 2013. The simple script wrapper for OpenMI: enabling interdisciplinary modeling studies. *Environ. Model. Softw.* 39, 283–294.
- Callahan, S.P., Freire, J., Santos, E., Scheidegger, C.E., Silva, C.T., Vo, H.T., 2006. Managing the evolution of dataflows with VisTrails. In: Proceedings of the 22nd International Conference on Data Engineering Workshops. IEEE Computer Society, Washington, DC, USA, p. 71.
- Cassandra, 2013. <http://cassandra.apache.org> (accessed 20.01.14.).
- Castronova, A.M., Goodall, J.L., 2010. A generic approach for developing process-level hydrologic modeling components. *Environ. Model. Softw.* 25 (7), 819–825.
- Castronova, A.M., Goodall, J.L., Elag, M.M., 2013a. Models as web services using the Open Geospatial Consortium (OGC) Web Processing Service (WPS) standard. *Environ. Model. Softw.* 41 (0), 72–83.
- Castronova, A.M., Goodall, J.L., Ercan, M.B., 2013b. Integrated modeling within a hydrologic information system: an OpenMI based approach. *Environ. Model. Softw.* 39 (0), 263–273 (Thematic Issue on the Future of Integrated Modeling Science and Technology).
- Chandrasekaran, S., Silver, G., Miller, J., Cardoso, J., Sheth, A., 2002. Web service technologies and their synergy with simulation. *Winter Simul. Conf.* 1, 606–615.
- Clark, M.K., 2008. Effects of High Commodity Prices on Western Kansas Crop Patterns and the Ogallala Aquifer (Unpublished M.S. thesis). Department of Agricultural Economics, Kansas State University.
- DataONE. <http://www.dataone.org> (accessed 20.01.14.).
- Deelman, E., Gannon, D., Shields, M., Taylor, I., 2009. Workflows and e-science: an overview of workflow system features and capabilities. *Future Gener. Comput. Syst.* 25 (5), 528–540.
- Demir, I., Krajewski, W.F., 2013. Towards an integrated flood information system: centralized data access, analysis, and visualization. *Environ. Model. Softw.* 50, 77–84.
- Garay, P.V., Peterson, J.M., Golden, B.B., Smith, C.M., July 2010. Disaggregated spatial modeling of irrigated land and water use. In: Selected Presentation at the Agricultural and Applied Economics Association Annual Meeting, Denver.
- Globus Online, 2013a. <http://www.globusonline.org> (accessed 20.01.14.).
- Globus Toolkit, 2013b. GridFTP User's Guide. <http://www.globus.org> (accessed 20.01.14.).
- Goodall, J.L., Robinson, B.F., Castronova, A.M., 2011. Modeling water resource systems using a service-oriented computing paradigm. *Environ. Model. Softw.* 26 (5), 573–582.
- Goodall, J.L., Saint, K.D., Ercan, M.B., Briley, L.J., Murphy, S., You, H., DeLuca, C., Rood, R.B., 2013. Coupling climate and hydrological models: interoperability through web services. *Environ. Model. Softw.* 46, 250–259.
- Gregersen, J.B., Gijssbers, P.J.A., Westen, S.J.P., 2007. OpenMI: Open Modeling Interface. *J. Hydroinform.* 9 (3), 175–191.
- Horak, J., Orlik, A., Stromsky, J., 2008. Web services for distributed and interoperable hydro-information systems. *Hydrol. Earth Syst. Sci.* 12, 635–644.
- Horsburgh, J.S., Tarboton, D.G., Maidment, D.R., Zaslavsky, I., 2008. A relational model for environmental and water resources data. *Water Resour. Res.* 44, W05406.
- Horsburgh, J.S., Tarboton, D.G., Piasecki, M., Maidment, D.R., Zaslavsky, I., Valentine, D., Whitenack, T., 2009. An integrated system for publishing environmental observations data. *Environ. Model. Softw.* 24 (8), 879–888.
- Howitt, R.E., 1995. Positive mathematical programming. *Am. J. Agric. Econ.* 77 (2), 329–342.
- Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M., Li, P., Oinn, T., 2006. Taverna: a tool for building and running workflows of services. *Nucleic Acids Res.* 34 (Web Server issue), 729–732.
- INRIA, Scilab. <http://www.scilab.org> (accessed 20.01.14.).
- Jagers, H., 2010. Linking data, models and tools: an overview. In: International Congress on Environmental Modelling and Software Ottawa Canada.
- Maidment, D.R., 2008. Bringing water data together. *J. Water Resour. Plan. Manag.* 134 (2), 95–96.
- Mason, S.J., Cleveland, S.B., Llovet, P., Izurieta, C., Poole, G.C., 2014. A centralized tool for managing, archiving, and serving point-in-time data in ecological research laboratories. *Environ. Model. Softw.* 51, 59–69.
- Memcached, 2013. <http://memcached.org> (accessed 20.01.14.).
- Nativi, S., Mazzetti, P., Geller, G.N., 2013. Environmental model access and interoperability: the GEO model web initiative. *Environ. Model. Softw.* 39, 214–228 (Thematic Issue on the Future of Integrated Modeling Science and Technology).
- OpenMI Association, 2010. News and Announcements. <http://www.openmi.org/archives/archived-news-and-announcements/2010> (accessed 20.01.14.).
- Ozturk, T., 2010. Scalable Data Structures for Java. Devoox, Metropolis Antwerp Belgium.
- Pettit, C., Williams, S., Bishop, I., Aurambout, J.P., Russel, A.B.M., Michael, A., Sharma, S., Hunter, D., Chan, P.C., Enticott, C., Borda, A., Abramson, D., 2013. Building an ecoinformatics platform to support climate change adaptation in Victoria. *Future Gener. Comput. Syst.* 29 (2), 624–640.
- Portele, C., 2007. OpenGIS Geography Markup Language (GML) Encoding Standard, Open Geospatial Consortium. OGC 07-036.
- Pullen, J.M., Brunton, R., Brutzman, D., Drake, D., Hieb, M., Morse, K.L., Tolk, A., 2005. Using web services to integrate heterogeneous simulations in a grid environment. *Future Gener. Comput. Syst.* 21, 97–106.
- Rajasekar, A., Wan, M., Moore, R., Schroeder, W., 2006. A prototype rule-based distributed data management system. In: HPDC Workshop on Next Generation Distributed Data Management, Paris, France.
- Steward, D.R., 2007. Groundwater response to changing water-use practices in sloping aquifers. *Water Resour. Res.* 43 (W05408), 1–12.
- Steward, D.R., Allen, A.J., 2013. The Analytic Element Method for rectangular gridded domains, benchmark comparisons and application to the High Plains Aquifer. *Adv. Water Resour.* 60, 89–99.
- Steward, D.R., Le Grand, P., Janković, I., Strack, O.D.L., 2008. Analytic formulation of Cauchy integrals for boundaries with curvilinear geometry. *Proc. Royal Soc. Lond. Ser. A Math. Phys. Eng. Sci.* 464, 223–248.
- Steward, D.R., Yang, X., Chacon, S., 2009. Groundwater response to changing water-use practices in sloping aquifers using convolution of transient response functions. *Water Resour. Res.* 45 (W02412), 1–13.
- Tarboton, D.G., Horsburgh, J.S., Maidment, D.R., Whiteaker, T., Zaslavsky, I., Piasecki, M., Goodall, J., Valentine, D., Whitenack, T., 2009. Development of a Community Hydrologic Information System. In: Anderssen, R.S., Braddock, R.D., Newham, L.T.H. (Eds.), 18th World IMACS Congress and MODSIM09 International Congress on Modelling and Simulation. Modelling and Simulation Society of Australia and New Zealand and International Association for Mathematics and Computers in Simulation, pp. 988–994.
- Vretanos, P.A., 2010. OpenGIS Web Feature Service 2.0 Interface Standard – OGC 09-025r1 and ISO/DIS 19142. Open Geospatial Consortium Inc.
- Williams, J.R., 1995. The Epic Model. In: Computer Models of Watershed Hydrology, pp. 909–1000. Chapter 25.
- Zaslavsky, I., Valentine, D., Whiteaker, T., 2007. CUAHSI WaterML, OGC 07–041r1. Open Geospatial Consortium Inc.