Shared Variable Management in SOAr-DSGrid

Xinjun Chen, Wentong Cai, Stephen J. Turner, and Yong Wang
Parallel and Distributed Computing Centre
School of Computer Engineering
Nanyang Technological University
Singapore 639798
{chen0081, aswtcai, assjturner, wang0065}@ntu.edu.sg

Abstract

Simulation is a low cost alternative to experimentation on real-world physical systems. In light of the prevalence of web services, issues relating to distributed simulation using web services have come to the fore in recent years. We have proposed a service-oriented architecture for distributed simulations on the Grid (SOAr-DSGrid). SOAr-DSGrid facilitates the development and execution of distributed simulations in different views (i.e., a component-based view for development and a service-oriented view for execution). At run time, one component may require information from another component for execution or decision-making. In SOAr-DSGrid, this shared information from one component to another component is implemented through shared variables. Since two components may execute out of phase, without proper shared variable management, the value provided by one component may be invalid for another component due to the time difference. Since SOAr-DSGrid provides a service-oriented view for component execution, the development of shared variable support also follows the service-oriented methodology. Three different approaches, namely Pull without FutureList, Pull with FutureList, and Push, have been implemented as the internal mechanism to solve the shared variable issue. Experimental results of the performance of these three approaches are also presented in this paper.

1. Introduction

Simulation has permeated many areas such as production, business, education, and science and engineering. It is a low cost alternative to experimentation on real-world physical systems. With the advances of computer networks and the prevalence of low-cost commodity computers, distributed simulation becomes a favorable technology to reduce simulation execution time and to enable the execution of simulation components at different geographical locations.

In a loosely-coupled distributed architecture, there should be as little dependency between components as possible. However, in real simulation applications, a certain degree of dependency between simulation components is inevitable. Figure 1 depicts a typical dependency between components in SOAr-DSGrid [1].

In Figure 1, Components CA, CB, and CC are distributed on different computers. Component CA processes entity e1 and produces entity e2. Subsequently, Component CA routes entity e2 to either Component CB or Component CC based on the result of rule r1. The processing of entity e1 and/or evaluation of rule r1 requires the value of the public variable v1 of Component CB. Component CB is a *producer component* and Component CA is a *consumer component*. A producer component is the owner of the shared variable, and a consumer component is the reader of the shared variable. A component may also have private variables, which cannot be shared to other components.

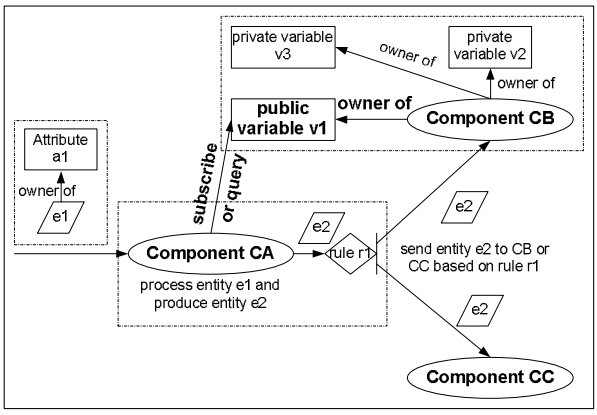

Figure 1. Shared variable scenario

There are two problems inherent in sharing variables in SOAr-DSGrid. First, because Component CA and Component CB are distributed as two stateful web

978-0-7695-3136-6/08 $25.00 © 2008 IEEE
DOI 10.1109/AMS.2008.161

services, a service interface must be provided for accessing shared variables. Second, since Components CA and CB are executed in a distributed manner, they may run out of phase (i.e., Components CA and CB may be at different simulation times). Consequently, if no proper shared variable mechanism is provided, the value of v1 returned from Component CB may not be at the simulation time as requested by Component CA.

In view of the above mentioned problems, the following tasks have been performed to enable shared variable support in SOAr-DSGrid:

1) At the component level, service interfaces are designed and implemented to enable runtime communication between two components to access shared variables.

2) Some underlying mechanisms, namely Pull without FutureList, Pull with FutureList, and Push, are proposed and implemented to ensure correct operations (i.e., query, update, and fossil-collection, of shared variables).

The rest of the paper is organized as follows. In section 2, we examine the background of SOAr-DSGrid and some related work pertaining to the shared variable problem in parallel and distributed simulation. In Section 3, we describe the detailed interface designed for runtime communication between components. In Section 4, we elaborate the underlying mechanisms to support shared variables in our service-oriented architecture. Then, we analyze the experimental results of the three approaches implemented in SOAr-DSGrid in Section 5. We conclude this paper in Section 6 with the outline of further work in this area.

2. Background and related work

In light of the prevalence of web services, issues relating to distributed simulation using web services have come to the fore in recent years. We have proposed a service-oriented architecture for distributed simulation on the Grid (i.e., SOAr-DSGrid) [1]. SOAr-DSGrid facilitates both development and execution of component-based distributed simulation, and views these two activities separately. At the development level, SOAr-DSGrid provides a component-based view. Simulation component developers define component interfaces according to a schema file predefined in SOAr-DSGrid. The simulation logic is separated from the component and defined in component operations. Component operations can be added into the component in a plug-and-play manner. As for modelers (i.e., simulation application developers), SOAr-DSGrid provides a schema for them to define simulation applications from the existing simulation components. At the execution level, SOAr-DSGrid provides a service-oriented view. Each simulation component is executed as a grid service in GT4 [3]. A simulation component is mapped to a web service in WSRF [8], and a simulation component instance is mapped to the combination of a web service and a resource instance. A web service is a collection of operations. A resource instance is a collection of state variables, on which the web service's operations perform. The web service's operations always get a resource instance first, and then operate on that resource. Shared variable management is part of the execution management of SOAr-DSGrid. The details of the execution management for shared variable management will be discussed in Sections 3 and 4.

The shared variable issue has been researched in the parallel and distributed simulation area for more than a decade. As early as 1993, Mehl and Hammes [6] proposed two general approaches to implement shared variables in a conservative synchronization algorithm, namely the *request-reply* and the *cached-copy*. In the request-reply, the owner maintains a history list for each shared variable. A history list contains the records which may be requested by the readers. The reader sends a request message to the owner for the value of the shared variable at simulation time t, and the owner will not send back the value until it is sure that the value of the shared variable will not be changed before t. The reader is time constrained by the owner, but not vice versa. The reader always issues a remote request when the variable is needed. In the cached-copy, the reader maintains a cached copy of the shared variable which is associated with the time-guarantee information which indicates the validity period of the cached-copy. The reader sends a request message to the owner only if the requested time goes beyond the validity period.

Most parallel discrete event simulation (PDES) protocols either forbid the use of shared variables or avoid the shared variable problem by certain partitioning techniques. In [7], Turner et al. proposed to partition simulation entities into separate conflict sets. In manufacturing simulation, if multiple machines share an operator, all these machines and the associated shared operators will be partitioned into the same conflict set. Each conflict set is then assigned to a logical process (LP). Hence, there are no shared variables between conflict sets (i.e., LPs).

In [2], Fujimoto proposed to allow zero-lookahead in the HLA time management specification to allow instantaneous access to attributes of a federate. Thus, the retrieval of a shared variable value can be implemented using zero-lookahead request and reply TSO messages. However, both the owner and the reader federates must be time constrained and must regulate the federation with a lookahead of zero. This is detri-

mental to the performance of the simulation. In order to improve the performance, Low et al. [5] revisited the shared variable problems, implemented, and conducted experiments on four Receive Order (RO) approaches, namely PullRO, PullROTG, PushRO, and PushROTG. Since RO messages are not used for computation of lookahead and time advancement constraint, the owner federate can regulate the federation with a non-zero lookahead. In the PullRO approach, the owner maintains a history list and the reader maintains a cache. The reader sends a pull request to the owner if the cache copy is no longer valid. In response, the owner will return the value with the time at which the value is updated. In PullROTG, the response from the owner also includes the time guarantee information to indicate the validity period of the cache. In the PushRO approach, the reader maintains a future list which contains the records that may be used by the reader in the future. The owner will push the value to the reader whenever the shared variable is updated. The reader will issue a pull request, if the required value cannot be found from the future list. In PushROTG, the owner pushes the value with time guarantee information to the reader.

3. Interface

In web services terminology, the methods in a web service exposed to other web services or applications are termed *operation*s. In stateful web services in GT4, we term the exposed methods *grid operation*s[1]. This section details the grid operations and other methods for runtime communication between components for the support of shared variables.

SOAr-DSGrid provides grid operations for retrieval of shared variables. Figure 2 shows the class diagram of the relevant classes involved in shared variable management. Unlike Object-Oriented Architecture (OOA), Service-Oriented Architecture (SOA) separates the data from the operations. In SOAr-DSGrid, all the grid operations are encapsulated in the web service implementation class PrimitiveComponent, and all the data are encapsulated in the PrimitiveComponentResource class which is an implementation class of ResourceProperties, that is, a collection of ResourceProperty [8]. Each ResourceProperty represents one state variable. The execution of each grid operation starts with the retrieval of a resource instance, followed by invocation of a corresponding method of the resource instance.

The following main data structures are used in the implementation (see Figure 2):

- **TimedRecord**, which is a tuple of *(*value, validFrom, validTill). value is the object representing the value of the shared variable. validFrom is the update time from which the value is valid, and validTill is the time from which the current value becomes invalid. When a shared variable is updated, the validTill field of the latest record will be updated with the current simulation time. A new record with both validFrom and validTill equal to the current simulation time will then be created.
- **HistoryList**, which is the implementation of the history list in the owner. The HistoryList maintains a list of TimedRecord objects and accessor methods to the list.
- **SharedVariable**, which is the class representation of a shared variable. It encapsulates a history list object. It also contains the subscriber list (i.e., a list of components that subscribe to the shared variable).
- **FutureList**, which is the implementation of future list in the reader. Similar to the history list, it contains a list of TimedRecord objects.
- **SubscribedVariable**, which is the class representation of a subscribed variable. A SubscribedVariable, the local shadow copy of the SharedVariable, is maintained by the reader. The SubscribedVariable encapsulates a future list object.

Each component instance has an instance of PrimitiveComponentResource. The owner component creates a SharedVariable object for each shared variable, and puts this object into the *sharedVariables* HashMap in the PrimitiveComponentResource instance. A SubscribedVariable object is used by the reader to maintain a local copy of each remote SharedVariable object. The created SubscribedVariable object is put in the reader's *subscribedVariables* HashMap. The HashMap sharedVariables and subscribedVariables are used to map the local variables defined in PrimtiveComponentResource to the SharedVariable object or SubscribedVariable object respectively.

SOAr-DSGrid defines an abstract class named Operation[2]. A subclass of Operation contains the application specific simulation logic. The execution of an Operation may use a local variable corresponding to the shared variable of another component instance. The Operation retrieves the value required by invoking the *getSubscribedVariable()* method of PrimitiveComponentResource. This method calls *getRecord()* method defined in the SubscribedVariable class, which in turn tries to get the value from the future list of the SubscribedVariable object. As for an owner, the Operation may modify the values of some shared variables.

[1] In this paper, all the grid operations are in bold, and all other class methods are highlighted in italic.

[2] The Operation class defined in SOAr-DSGrid is highlighted in Arial font to differentiate it from the grid operation concept in GT4.

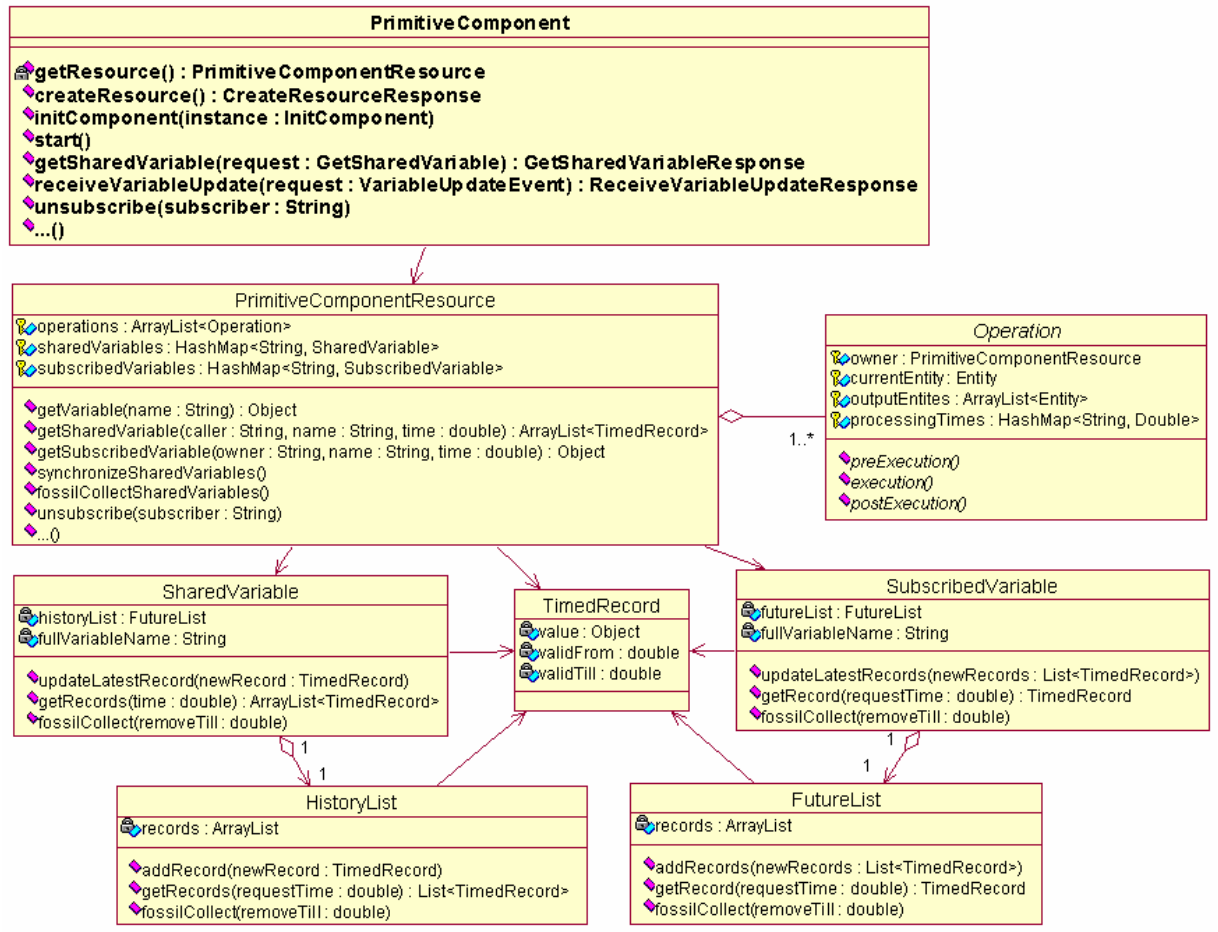

Figure 2. Data structures and interface for shared variables

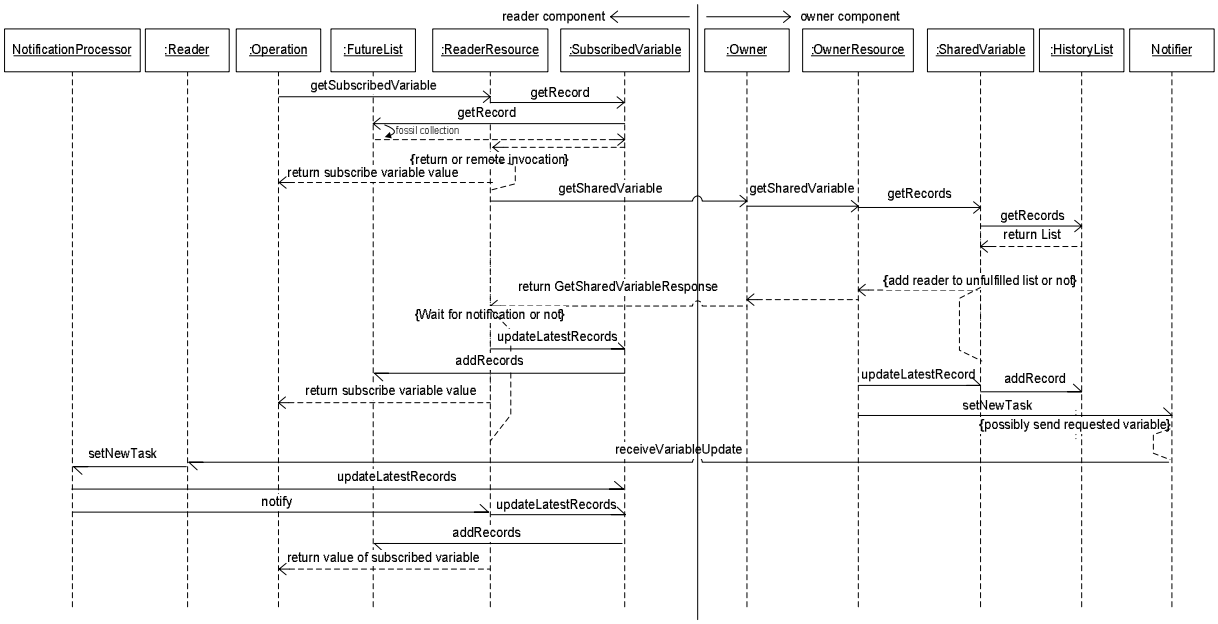

Figure 3. Sequence diagram for shared variable update

synchronizeSharedVariables() is used by the owner to add a record to the SharedVariable object using the value of the local variable which corresponds to the shared variable.

In the case that the reader's future list does not contain the required value, the reader will use the grid operation **getSharedVariable()** to get the updated values of the shared variable from the owner. When the owner updates its simulation time, the Notifier of the owner will invoke the corresponding reader's **receiveVariableUpdate()** grid operation: either push the latest record to the reader in an unsolicited manner or return the record to the reader as the response of an unfulfilled request. At the reader side, the NotificationProcessor will save the received updated record into the future list if there is any.

Figure 3 depicts the sequence of retrieval, update, and fossil-collection of shared variables. How these activities are used in the basic Pull without FutureList, Pull with FutureList, and Push mechanisms is described in Section 4.

4. Basic mechanisms for shared variable support

The underlying mechanisms described in this section are similar to the methods proposed in [5]. The Pull without FutureList is the same as PullROTG and the Push method is similar to PushROTG. Push is different from PushROTG when the reader runs ahead of the owner. For PushROTG, the reader will issue a pull request to the owner if the reader runs ahead of the owner. But for Push, the reader will not send a pull request to the owner and will wait till the requested value is pushed by the owner. The author introduced a new approach, Pull with FutureList to improve the performance of PullROTG.

4.1 Pull without FutureList

The Pull without FutureList is the same as the PullROTG in [5]. In this approach, the owner maintains a history list for each shared variable, and the reader maintains a local cache of the last received TimedRecord (not a FutureList) for each subscribed variable. When the reader requires the value of the shared variable at time *t*, it first checks whether the requested time falls in the validity range of the local cache. If the local cache satisfies the condition $validFrom \le t < validTill$ *or* $validFrom = t = validTill$, then the *value* will be returned immediately to the main execution thread of the component. Otherwise, the local cache is considered as obsolete. A remote invocation of **getSharedVariable()** will be issued to the owner.

When the owner receives the request, it will try to get the requested record from the history list. The owner will return the single record which satisfies the condition $validFrom \le t < validTill$ *or* $validFrom = t = validTill$ as a response to the reader. If no such record is found, the owner will add the request to the unfulfilled list, and return a null record to the reader immediately without blocking the invocation. When the owner advances simulation time and the requested record becomes available, the Notifier of the owner will invoke the **receiveVariableUpdate()** grid operation of the reader.

When the reader receives the response, the NotificationProcessor of the reader will save the record as the fresh cache copy and return the value to the main execution thread if the response is not null. Otherwise, the reader's main execution thread will keep waiting until it is notified by the NotificationProcessor after the NotificationProcessor receives the record from the owner.

4.2 Pull with FutureList

The Pull with FutureList is similar to the Pull without FutureList. The differences are:

- When the owner receives a **getSharedVariable()** invocation, instead of returning a single record, it will check the history list to find all the records which satisfy the condition $t < validTill$ *or* $validFrom = t = validTill$.
- Since the reader maintains a future list, it will only send a pull request to the owner if it cannot find the requested value in the future list. The reader also needs to fossil-collect the future list properly.

4.3 Push

In the Push approach, the reader maintains a future list for each subscribed variable. This approach is based on the always-update-by-writer update policy, similar to the one used by Lim et al. [4]. The Push is similar to PushROTG, but it is different from PushROTG when the reader runs ahead of the owner.

When the reader requires a variable, it always tries to retrieve the value from the corresponding future list first. If in the future list there is a record that satisfies the condition $validFrom \le t < validTill$ *or* $validFrom = t = validTill$, then the *value* will be returned immediately to the main execution thread of the component, and the future list will be fossil-collected accordingly. Otherwise, the reader's main execution thread will wait till the requested record is received from the owner through the reader's NotificationProcessor after it received the required record from the owner. This is different in PushROTG. In PushROTG, the reader will

send a pull request to the owner if the reader runs ahead of the owner.

At the owner side, if a variable update happens, the Notifier thread will push the updated record to each reader through the **receiveVariableUpdate()** grid operation of the reader. As described above, when the reader receives the notification, the NotificationProcessor will add the updated record to the future list, and subsequently notify the main execution thread of the component to read from the future list.

4.4 Fossil-collection

In all approaches proposed and implemented, a history list and/or a future list are involved. If the sizes of these lists keep on increasing, the cost of searching and the memory usage will also increase, and it will finally reach a point where the solution becomes infeasible. Thus, an effective and proper fossil-collection is required for both the history list and the future list.

Fossil-collection happens whenever there are some obsolete records in the lists. As for the history list, when the global minimal simulation time of all the subscribers increases, the history list should be fossil-collected. As for the future list, whenever an Operation requests for a subscribed variable at a certain time, all the records whose *validTill* is smaller than the request time are considered as obsolete. However, the latest record in the history list should never be fossil-collected[3]. The record whose *validTill* is equal to the request time should not be fossil collected as this record may be requested again with the same request time. Figures 4 and 5 show the algorithms used for fossil-collection of the history list and future list respectively.

```
1 removeTill // remove all records with validTill <
removeTill (except for the last record)
2  while(there is a record) {
3     if(record is the latest record) break;
4     else if(record.validTill < removeTill){
5        remove this record;
6     } else { break; }
7  }
```

Figure 4. Fossil-collection of the history list

```
1 removeTill // remove all records with validTill <
removeTill
2  while(there is a record) {
3     if(record.validTill < removeTill)
4        remove this record;
5     } else { break; }
6  }
```

Figure 5. Fossil-collection of the future list

[3] *validFrom* and *validTill* fields of this record are equal and the *validTill* field will be set to the updated simulation time when the next time the shared variable is updated.

5. Experiments

Experiments were carried out to compare the performance of the three solutions: Pull without FutureList, Pull with FutureList, and Push. The same simulation model is used in the experiments with different configurations. The simulation terminates at a specific simulation time for all experiments. In the simulation model, there are two components. One is the owner component with OperationA. The other one is the reader component with OperationB. After OperationA is triggered, the owner will do the following:

1) Executes a spin-loop for a period, defined by OwnerUnitProcessingTime;

2) Updates the value of the shared variable and synchronizes the shared variable;

3) Advances its simulation time by an amount defined by UpdatePattern.

After OperationB is triggered, the reader will do the following:

1) Advances its simulation time by an amount defined by RequestPattern;

2) Retrieves the shared variable;

3) Executes a spin-loop for a period defined by ReaderUnitProcessingTime.

The following table summarizes the factors that affect the simulation performance.

Table 1. Simulation factors

Factor	Description
UpdatePattern	The update interval (in simulation time) of the owner
RequestPattern	The request interval (in simulation time) of the reader
OwnerUnit ProcessingTime	The emulated processing time (in wall-clock time) between two consecutive simulation time advancements of the owner
ReaderUnit ProcessingTime	The emulated processing time (in wall-clock time) between two consecutive simulation time advancements of the reader

In the first experiment, we vary the unit processing time of the reader. In the second experiment, we vary the request pattern of the reader.

5.1 Varying unit processing time

In this experiment, the configuration of the simulation is summarized in Table 2.

From Figure 6, it can be seen that when the reader unit processing time is smaller than the owner unit processing time, the Push gives the best performance. Since the reader unit processing time is smaller, the reader runs ahead of the owner. In this case, for both Pull without FutureList and Pull with FutureList, the

reader will issue a pull request through the **getSharedVariable()** grid invocation to the owner. The owner will return a null record immediately because the requested record is not in the HistoryList of the owner. This will cause the reader to wait until the owner's simulation time advances to the requested time. Then, the owner's Notifier will return the requested record to the reader through the **receiveVariableUpdate()** grid invocation. Thus, either in Pull with or without FutureList, these **getSharedVariable()** grid invocations are essentially redundant. In the Push approach, these redundant invocations are avoided and consequently the performance of Push is better than the other two.

Table 2. Simulation configuration (varying reader unit processing time)

	Value
UpdatePattern (simulation time)	10
RequestPatten (simulation time)	10
OwnerUnitProcessinTime (wall-clock time in millisecond)	50
ReaderUnitProcessingTime (wall-clock time in millisecond)	3, 5, 10, 20, 40, 50, 60, 80, 100, 200

The experiment results are summarized in Figure 6.

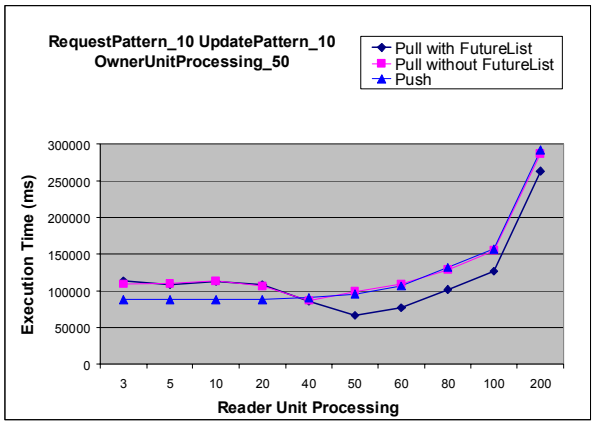

Figure 6. Experiment results (varying reader unit processing time)

The figure also shows that the Pull without FutureList and Pull with FutureList have similar performance when the reader runs ahead of the owner. It is clear that the Pull with FutureList will degrade to the Pull without FutureList as there is always zero record returned to the reader for every **getSharedVariable()** grid invocation.

When the reader unit processing time is greater than the owner unit processing time, the owner runs ahead of the reader. In this case, the Pull with FutureList gives the best performance. Since the owner runs ahead of the reader, multiple records may be available to the reader when the reader issues the **getSharedVariable()** grid invocation. So at the reader side, most of the requests of the values of the shared variable can be fulfilled by the FutureList. As for Push, most of the requests can also be fulfilled from the FutureList. However, since the owner only pushes one record to the reader each time, the number of **receiveVariableUpdate()** grid invocations will be similar to the number of grid invocations in Pull without FutureList. That is why Push has similar performance as Pull without FutureList when the owner runs ahead of the reader.

5.2 Varying request pattern

In this experiment, the configuration of the simulation is summarized in Table 3. The experiment results are summarized in Figure 7.

Table 3. Simulation configuration (varying request pattern)

	Value
UpdatePattern (simulation time)	10
RequestPatten (simulation time)	2, 4, 8, 16, 32, 64, and 128
OwnerUnitProcessinTime (wall-clock time in millisecond)	50
ReaderUnitProcessingTime (wall-clock time in millisecond)	50

In this experiment, the owner and the reader have the same unit processing time. The relative relationship between the update pattern and the request pattern will determine whether the owner runs ahead of the reader or the other way.

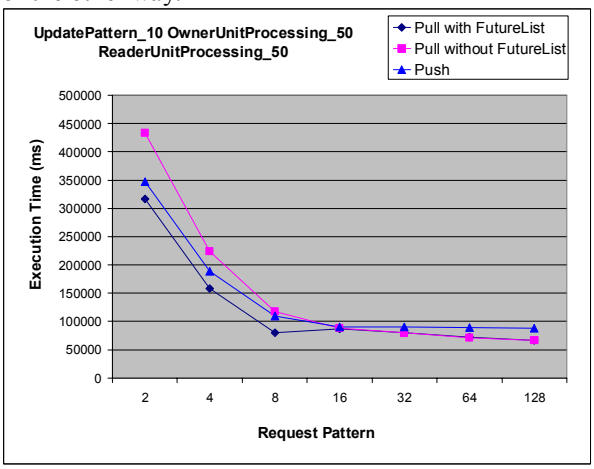

Figure 7. Experiment results (varying request pattern)

When the request pattern is less than update pattern, the owner essentially runs ahead of the reader since the owner's simulation time advances more than

the reader's in the same wall-clock time. In this case, the Pull with FutureList gives the best performance. As the update pattern is larger, when the reader requests for the value, a single record's validity period will be greater than the request pattern. The same record will be used to fulfill several consecutive requests. During this period, the owner advances its simulation time further. Therefore, the next **getSharedVariable()** grid invocation will fetch more records, and more requests will be fulfilled by the FutureList. In total, the number of **getSharedVariable()** grid invocations in Pull with FutureList is much smaller than that of the Pull without FutureList. The number of **receiveVariableUpdate()** grid invocations in Push is fixed since each update will lead to one **receiveVariableUpdate()** grid invocation.

When the request pattern is greater than the update pattern, the reader advances simulation time faster. As the request pattern is greater, in both Pull without FutureList and Pull with FutureList, the reader needs to issue a **getSharedVariable()** grid invocation and wait until the owner's Notifier issues a **receiveVariableUpdate()** grid invocation. The total numbers of grid invocations for Pull without FutureList and Pull with Future list are the same. However, for Push, the total number of **receiveVariableUpdate()** grid invocations is larger than the total number of grid invocations in Pull with FutureList, and most of the pushed records are not used by the reader since the request pattern is greater than the update pattern. Therefore, the Pull with FutureList and Pull without FutureList have similar performance, and the Push has the worst performance.

6. Conclusions

Simulation is a low cost alternative to experimentation on real-world physical systems. SOAr-DSGrid is a service-oriented architecture for distributed simulation on the Grid. In order to support shared variables in SOAr-DSGrid, interfaces for components to communicate at execution time are developed. The underlying mechanisms are also implemented to provide runtime support for shared variables. Three approaches namely, Pull without FutureList, Pull with FutureList, and Push, are implemented. Pull with FutureList is a greedy version of Pull without FutureList. The Pull with FutureList generally has better or the same performance as the Pull without FutureList. The speed of the simulation is affected by four factors in two dimensions. The first dimension is the simulation time advancement amount (i.e., update pattern and request pattern), and the second is the unit processing time (i.e., the owner unit processing time and the reader unit processing time). If the reader runs ahead of the owner because the reader unit processing time is smaller, then the Push has better performance than the Pull with FutureList. For all the other three combinations, the Pull with FutureList has better performance over the Push.

As for future work, an adaptive algorithm which allows the reader and/or the owner to switch between Pull with FutureList and Push dynamically will be developed to improve the performance.

References

[1] X. Chen, W. Cai, S. J. Turner, and Y. Wang, "SOAr-DSGrid: Service-Oriented Architecture for Distributed Simulation on the Grid", *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation (PADS'06)*, Singapore 2006, pp 65-73.

[2] R. M. Fujimoto, "Zero Lookahead and Repeatability in the High Level Architecture", *Proceedings of 1997 Spring Simulation Interoperability Workshop*, Orlando, Florida, United States 1997, 97S-SIW-046.

[3] Globus, Globus Toolkit Version 4.0, http://www.globus.org/toolkit/.

[4] C. C. Lim, Y. H. Low, B. P. Gan, and S. Jain, "Implementation of Dispatch Rules in Parallel Manufacturing Simulation", *Proceedings of 1998 Winter Simulation Conference*, Washington DC, United States 1998, pp 1591-1597.

[5] Y. H. Low, B. P. Gan, J. Wei, X. Wang, S. J. Turner, and W. Cai, "Shared State Synchronization for HLA-Based Distributed Simulation", *Simulation: Transactions of the Society for Modeling and Simulation International*, Vol. 82, No. 8, pp. 511-521, August 2006.

[6] H. Mehl, and S. Hammes, "Shared Variables in Distributed Simulation", *Proceedings of 7th Workshop on Parallel and Distributed Simulation (PADS'93)*, San Diego, California, United States 1993, pp 68-75.

[7] S. J. Turner, W. Cai, C. C. Lin, Y. H. Low, W. J. Hsu, and S. Y. Huang, "A Methodology for Automating the Parallelization of Manufacturing Simulation", *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, Banff, Alberta, Canada 1998, pp 126-133.

[8] WSRF, the Web-Services Resource Framework Version 1.0, http://www.globus.org/wsrf/specs/ws- wsrf.pdf