Shared Data Grid Programming Improvements using Specialized Objects

Dacian Tudor, Georgiana Macariu, Vladimir Cretu
Computer Science Department
"Politehnica" University of Timisoara,
Timisoara, Romania
{dacian, georgiana, vcretu}@cs.upt.ro

Wolfgang Schreiner
Research Institute for Symbolic Computation,
Johannes Kepler University,
Linz, Austria
wolfgang.schreiner@risc.uni-linz.ac.at

Abstract - The shared data programming model is an attractive grid programming alternative to message passing solutions. This paper addresses type related improvements to a shared data grid programming model, by taking advantage of particular interactions patterns and object semantics which result in a broader set of specialized objects. We present analysis aspects of GUN, a grid service layer solution for shared data programming. We start by defining the evaluation criteria in terms of performance, resource and qualitative aspects. A series of experiments are used in order to highlight relevant behavioral and performance aspects of the model's implementation. Next, the analysis of the model is carried out at experimental level where a Java based prototype of the grid universe has been built. Starting with the generic objects and followed by every object type, the object particularities are introduced and the experimental results are described and discussed.

Keywords: grid, shared data programming.

I. INTRODUCTION

The grid programming concepts landscape is currently dominated by message passing solutions. Solutions based on shared data programming are almost not present. We have been working towards a model for a service layer for distributed shared data grid programming that aims to provide a more appealing programming solution based on an object oriented view and the combination of a relaxed memory consistency and type coherence model [1]. The model makes use of the universe concept which is an abstraction of networked machines in latency proximity. The model makes use of entry consistency specification at the object level as well as specialized objects that aim to provide additional information on data interactions. Decoupling data representation and replication from the operational execution opens new dimensions in programming grid applications whose exploration is one of the main focus points of this work. Some of the recent attempts to deliver shared data programming solution have been presented in [2],[3],[4],[5] together with a new proposal based on the concept of Grid Universe [6].

Distributed systems analysis is a very important, complex and sensitive topic. Evaluating a grid system in ideal conditions is straightforward, but it opens the question of reproducibility likelihood, meaning that if one wants to reproduce a given experiment, one must ensure a similar environment. Due to the complexity of grid systems, sometimes this requirement cannot be achieved. Worse, a real-life experimental scenario is almost impossible to reproduce in case of a large scale distributed application deployed on a wide area grid. As a result, we have considered large scale test environments made out of either physical machines or simulated environments where every machine is dedicated its own processor. In addition, we have considered basic application specific interactions as part of the evaluation use case opposite to evaluating a complete distributed application which always relates to a particular implementation. In our work we aim to perform system analysis on three different directions: theoretical analysis, prototype-based analysis and computer aided analysis. We distinguish three main analysis domains while analyzing a grid system: performance analysis, resource related analysis and quality related analysis. This paper focuses on prototype-based analysis.

II. GRID UNIVERSE EXPERIMENTS

GUN is the acronym for Grid UNiverse and represents a Java based implementation of the grid universe model defined in [1]. Remote interactions are expressed in GUN based on Java's remote object model. First, the Remote Method Invocation (RMI) solution was chosen for its simplicity and ease of use. Second, because the system model does not require multicasting support (like Jini [7] or ProActive [8] solutions do for example), the RMI model fits well to the abstract model and completely isolates remote data accesing.

GUN reflects the architecture of the abstract model and the abstract system architecture described in [6]. Similar to the abstract model, GUN considers a set of processes deployed over several networks called universe nodes. The universe nodes are homogeneous and each of them is able to accommodate a certain number of shared objects, until the available capacity of the universe node is consumed. Typically, universe nodes are grouped in network latency proximity and form a universe. The collection of all universes is called the grid universe. Each universe contains a dedicated node called "primary node" which manages the communication with other universes and indexes the information on available data items accommodated by each node within the same universe. All primary nodes can be seen as a distributed registry, each being responsible for managing certain number of data objects.

We have analyzed the GUN prototype with respect to three major criteria: performance, resource utilization (memory and number of indirections) and quality (success rate for acquire operations and operation throughput). In terms of performance analysis we have defined a series of measurements, like completion time (CT - execution time for a distributed application, from the time the first application process starts execution until the last process finishes) and acquire/acquire exclusive time (AQT/AQET – elapsed time from the moment of issuing an acquire request from a node until the acquire operation is granted on the referred data).

An experiment defines a series of operations which follows four steps: deployment, creating the shared data objects, reaching a steady state where objects are replicated and performing the concrete operations and measurements. Experiments might have variants. An experiment variant defines the exploration in the system parameters space where different parameter configurations are used.

In order to perform experiments on a high number of grid nodes, a simulated environment was used. The solution involved an SGI Altix 4700 machine located at the Research Institute for Symbolic Computation, which has 128 Intel Itanium 2 Montecito processors with hyper-threading technology, running at 1,6GHz and having 18 MB L3 cache which can execute up to 256 threads simultaneously.

Artificial latencies between remote calls were introduced in the GUN prototype in order to reflect a real deployment. In order to reduce the risk of uncontrolled thread scheduling, a spinning wait was used. A latency of 10ms was considered for calls within a universe and 50ms for calls from one universe to another. It is important to note that these values are highly dependent on the remote method signature as well as their values (e.g. in case of a list of various objects), as all method parameters are serialized and transferred over the network. This aspect was not addressed in the experiments running on Altix as it naturally happens in a real deployment, because in case of GUN, there is no significant data marshaling between remote machines. However, the remote execution penalty in the real wide scale distributed environment was not higher than 250ms, considering the parameters for all remote methods defined in GUN. In case of only the European clusters, the latency was between 40ms and 90ms. As a result, the fixed value of 50ms was considered in the Altix evaluation setup.

III. Generic Objects Performance

A. *Acquire*

During all the experiments we have noticed a 100% success rate for all acquire operations that were issued, considering a timeout value of 2000ms. In case of a deployment of 30 nodes per universe, but where a client is running on every node, depending on the delay between subsequent operations the results illustrated in Figure 1 were obtained. The first group of 30 nodes belongs to the universe where the token resides. As there is no acquire exclusive issued in this scenario, the token remains fixed, thus the nodes within the same universe experience a very low acquire time. The other nodes which belong to universes which do not hold the token experience increasing acquire time values as the delay between subsequent requests decreases. In case the acquire operations are issued with a delay of 2000ms respectively 3000ms, GUN shows a good and stable performance independent on the node location. There is only a slight increase in acquire time for the nodes belonging to the universes that do not have the token. This is quite normal since the acquire request has to pass the universe boundaries (remote call over large latency connection). In the other three cases, there is a performance degradation in the system when the acquire requests are issued more rapidly. This happens because all nodes are issuing request towards the primary node that holds the token and the requests are serialized in a queue. If the request frequency is higher than the processing frequency, the requests are accumulating in the queue and the waiting time increases. It is worth to note that the processing frequency depends on the inter-universe network latency because the responses are sent via a large latency connection.

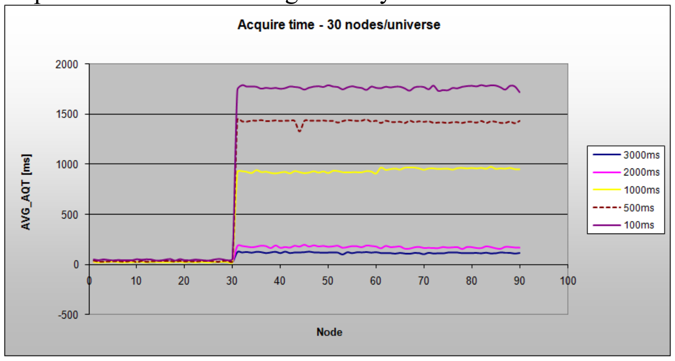

Figure 1. Acquire time

B. *Acquire Exclusive*

The correctness of the GUN implementation for acquire exclusive was assessed according to the specifications of the entry consistency model. In our experiment, each grid shared data contains a counter which is incremented each time the value of the object is changed. Since the order of acquire operations is the same from every node's point of view, the interleaved sequence of object counters must be ordered. Such an ordered sequence of object versions was witnessed for all the run experiments.

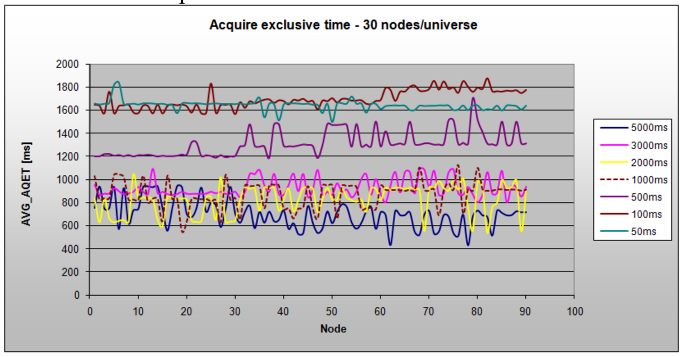

Figure 2. Acquire exclusive time

In case of 30 nodes per universe deployment where a client is running on every node and is issuing acquire exclusive requests as specified in the experiment description, the average acquire exclusive time for 7 different values of the delay parameter is represented in Figure 2. If the delay is higher than 1000ms, acquire exclusive time does not show any significant fluctuations between universe nodes and remains stable between 400 and 1000ms. If the delay is reduced below 1000ms, acquire exclusive time increases. The explanation is that smaller delays lead to a higher number of requests per time unit. As the processing capability of the primary nodes has an upper bound, requests accumulate in the primary node's queues and thus increasing the processing time.

C. *Release*

Figure 3 shows that contrary to the acquire time, the release time remains stable independent on the request frequency (acquire frequency is equal to the release frequency) since the release operation is implemented asynchronously and a response is not awaited from the primary nodes. Only in the extreme case where requests are issued within each 100ms, a variation of the release time can be noticed but this can be accounted to the global task scheduling mechanism.

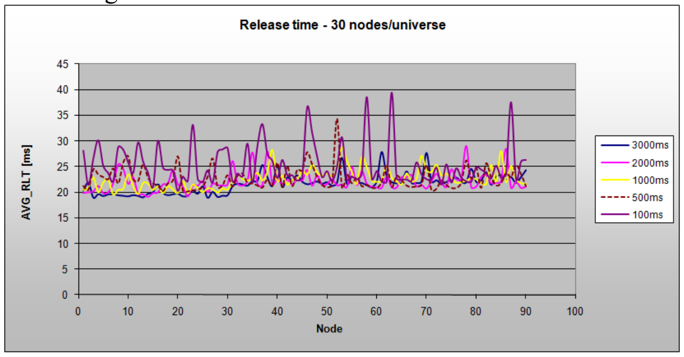

Figure 3. Release time

IV. SPECIALIZED OBJECTS PERFORMANCE

A. *Read-only Objects*

Read-Only objects are immutable grid objects that are created by one process and their value is bound to the value at the time of creation. These kinds of object do not require any synchronization mechanisms as the state is not changing after the object is added to the grid universe. Such kind of objects can benefit of a high replication rate.

Definition: A read-only object is a grid object accessible via its corresponding reference with the following properties: the value of the object is bound to the creation time value and the object value changes are not propagated among object replicas.

In case of read-only objects the acquire time is close to zero as no locking is necessary since the object's state is immutable. The same applies for the release operation. In this respect, the difference to the generic grid object is evident (assuming that the read only object is replicated to the caller node and thus no remote invocation cost is incurred). Since there is no logic behind the acquire operation there is no dependency to the number of nodes.

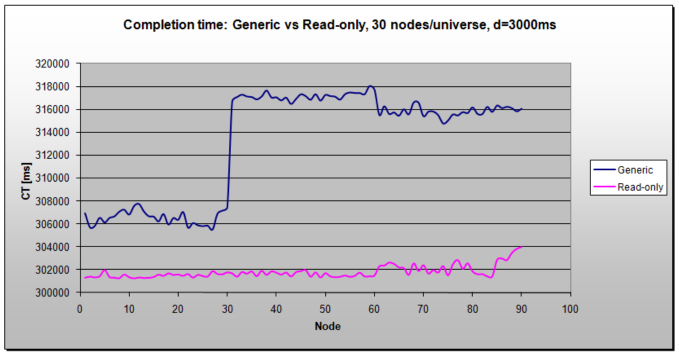

Figure 4. Completion time read-only vs. generics

The difference in the application completion time between read-only and generic objects for d=3000ms are shown in Figure 4. In case of the generic objects, the first group of object resides in the universe containing the token and is experiencing significantly lower values for the completion time. The other group of objects require to contact the primary node containing the token in order to get the acquire request granted and thus have higher completion time values. There is a significant advantage of using read-only objects where frequent object state must be read especially when the location of the token is uncertain (e.g. one client performs acquire exclusive operation and all other requests have to be redirected to a remote universe).

B. *Private Objects*

Private objects are objects that belong to a certain fixed location, namely a fixed node. During a computation, sometimes a grid object is only needed by the node where it belongs to and no other process from the same universe or other universes require that object. Their purpose is to apply local optimizations for frequent local interactions or to mark that the object is fixed. As private objects are accessed typically only locally, they do not have grid scope locking mechanisms as they are not replicated and are typically accessed by only one process. Such an object can be thought of as simply a local data carrier. The following definition applies:

Definition: A private object is a grid object accessible via its corresponding reference with the following properties: it is bound to a fixed node and only one copy of the object exists at any time in the Grid Universe.

Although they are shared objects, they are typically accessed by one or more local processes. Some algorithms can benefit out of this sharing pattern if they exhibit a waveform processing pattern. Such condition happens if they are processing elements of a structure and after the element is processed, it is only used by the local processes and no other external process. Thus, the reason of this object type is to reduce the synchronization overhead by reducing the scope of object monitoring and provide an optimized local lock mechanism.

The experiments have been conducted in respect to both acquire and acquire exclusive operations. In case only acquire requests have been issued, the performance of the private objects turned out to be superior to the generic objects. One of the obvious differences between private and generic objects is when the token does not belong to the same universe as the client node. The private object relies on the pure Java implementation of a multi-threaded monitor object. On the other side, GUN relies on a message queue which adds a queuing effect to all acquire requests. Particularly to this scenario is the fact that only acquire calls are being issued and no acquire exclusive calls are being made. As a result, the implementation of the private objects is more efficient. The saturation effect experienced when the request frequencies increase is expected to appear to the private objects too, but on a more moderate scale.

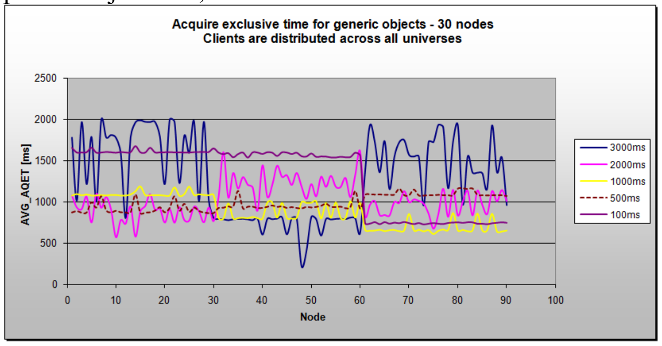

Figure 5. Acquire exclusive for generic objects

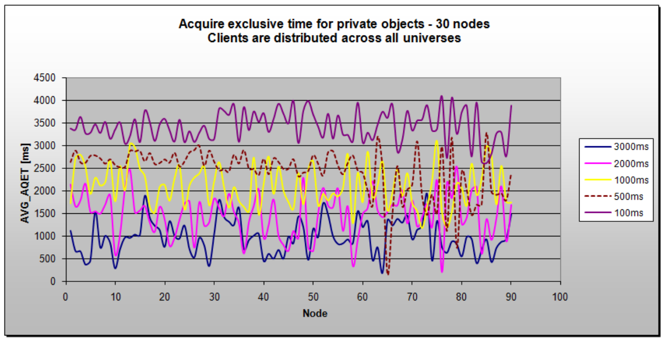

Figure 6. Acquire exclusive for private objects

With respect to the acquire exclusive operation, when the clients are distributed across all universes, the performance measurements are summarized in Figure 5 and Figure 6. Considering the more relaxed scenario where d=5000ms, the generic objects require about 600ms to satisfy an acquire exclusive request. Under the same conditions, the private objects require oscillating values around 1000ms. As the delay d decreases (more requests per time unit), generic objects require about 1800ms in the worst case scenario where d=100ms. This time private objects require about 3500ms to satisfy a request.

This experiment clearly shows the bottleneck of the global sequencer of the private object. The difference becomes evident when more than one client is used and accelerate when the number of client nodes increases. It is obvious that the GUN implementation is better in terms of response time and scalability than a plain sequencer. Although pure acquire operation appears to be more efficient, by extrapolating the results shown in this section, it is expected that seldom acquire exclusive requests can cause severe performance degradation. As a conclusion, as expected from the model design, private objects are very efficient when the callers are localized in the proximity of the shared data and only a limited number of calls are being issued. Breaking the locality constraints as well as the request limit leads to performance degradation and it would be better to use generic objects for these situations.

C. *Migratory Objects*

Migratory objects represent grid objects that are accessed in phases by multiple processes. In every phase, a single process is taking exclusive ownership of the object. After the object is used by one process, another process takes its turn and applies another state modification. Migratory objects carry only the semantics of the object type and take advantage of the exclusive acquire operation to trigger a migration of the object to a new location for the new access.

Definition: A migratory object is a grid object accessible via its corresponding reference with the following properties: the object is accessed by multiple processes in phases, one process at a time and for any migratory object, there is no replicated object in the grid universe.

These kinds of object are never replicated and only one process is using them at any time, thus no concurrency issues can be exploited, but only locality Similar to private objects, they are supposed to be used only by one process at a time. The difference is that whereas private objects are fixed (they don’t change their location), migratory objects can move between nodes or between universes. Opposite to private objects, all locks are performed remotely, at the object scope, at the location where the object is located.

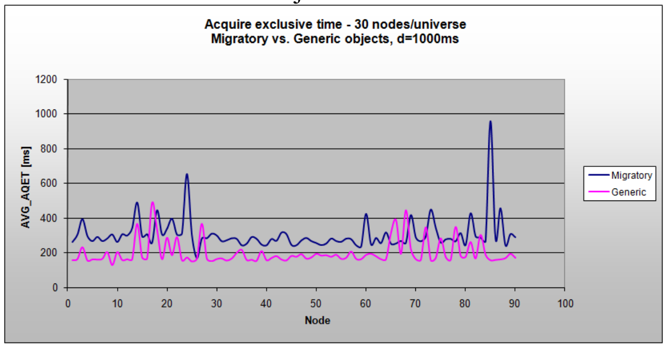

Figure 7. Migratory vs. generics

Figure 7 show the differences between migratory and generic grid objects with respect to the acquire exclusive time for different values of the parameter d. There is a difference of about 100ms in favor of the generic objects. We must mention that this experiment is a corner case which aims to answer the question whether there are situations where migratory objects are a real advantage compared to generic objects (assuming the same interaction orchestration is defined). It is self-evident that migratory objects would perform better than generic objects when the latter ones are not replicated. In this case, generic objects would experience the bottleneck effect.

The opposite case is the one described in the experiment, where generic objects are highly replicated. The experiment has proved that migratory objects do not improve the performance by the locality aspect since the locality is already offered by the generic object without the migration penalty. However the difference is not significant. Most probably an even point would be the situation where generic objects are replicated just on a certain number of nodes. Exceeding that threshold would bring the system in the state described above. Below that threshold it is expected that migratory objects perform better than the generic ones.

D. *Producer-Consumer Objects*

Producer-Consumer objects are grid shared objects written by only one process called producer process and read by multiple other processes called consumer processes.

Definition: A producer-consumer object is a grid object accessible via its corresponding reference with the following properties: the object's state is written by only one process and its state is read by multiple other processes.

The runtime system takes advantage of their semantics and can perform eager object synchronization. This implies that after a write operation that releases the object, the object's state might be synchronized in advance so that all other reader processes do not require another internal state synchronization. Thus, the possible optimizations in this case would be object replication for read operations and eager updates to all replicas at release time. In the best case scenario, consumers do not need to wait until the state is replicated across universes.

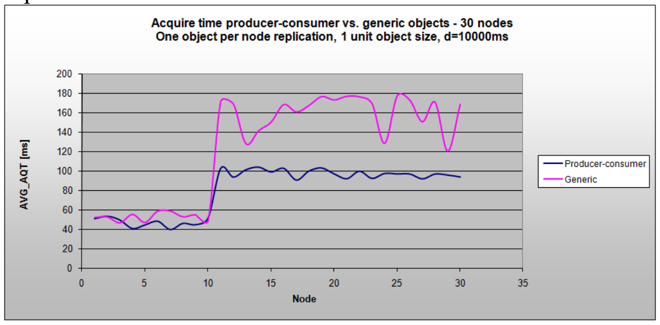

Figure 8. Acquire time producer-consumer vs generics

Figure 8 illustrate the differences between producer-consumer and generic objects in respect of acquire time, if one node per universe replication is used and the object size unit is equal to 1, for a delay of 10000ms. The overall graph shape is similar to the ones obtained for generic objects where the first group of nodes belongs to the universe holding the token. As shown in both graphs, producer-consumer objects exhibit lower acquire and completion time values than the generic objects.

In terms of differences between producer-consumer and the generic object with respect to AQET and release time (RLT), the experiments showed smaller values in case of producer-consumer type (281ms vs. 1052ms) since the chances for object synchronization prior to granting the access are totally reduced. The opposite situation can be noticed for the release time (1783ms vs. 25ms) because the release operation is synchronizing all object replicas. It is important to note that for the entire application execution, the distribution in time of these operations is important since a snapshot to the system's status does not reflect the entire behavior of the system.

E. *Read-Mostly Objects*

Read-mostly objects are those grid objects that are mostly read than written, leading to a high read/write ratio. In case of these objects, it is desirable to have more replicas that can be updated after each write operation requested by a process. This kind of objects is similar to producer-consumer objects, with the difference that there might be more than one writer as in the producer-consumer case.

Definition: A read-mostly object is a grid object accessible via its corresponding reference with the following properties: the object's state is written by at least one process, the object's state is read by multiple processes and the object's read/write ratio is higher than a given threshold, thus the object is mostly read.

The runtime system takes advantage of their semantics and can perform a proactive and eager replication protocol to achieve shorter synchronization timings.

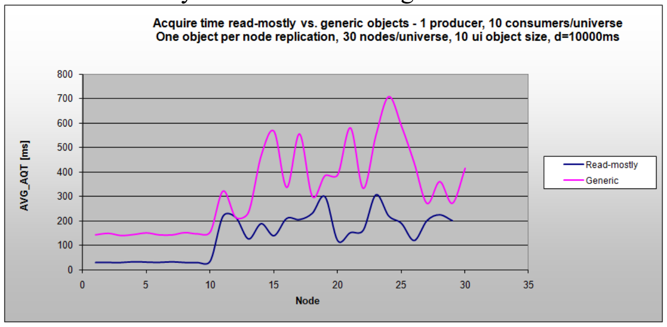

Figure 9. Acquire time: read-mostly vs. generics

Considering a stable system where requests are being issued every d=10000ms where there are 10 consumers per universe and only one producer in the entire grid universe and one object per node replication rule is chosen, as shown in Figure 9, read-mostly objects show a much lower acquire time. Acquire exclusive time as well as release times are higher due to the write-update protocol which is used for read-mostly objects. The graphic shape is similar to the acquire time for generic objects where the first group of nodes belong to the universe holding the token. Since there is only one producer there is no token movement.

F. *Result Objects*

Result objects are objects that are constructed through a builder process, where many processes are writing separate and non-conflicting parts and one process is reading the final result upon completion. Once written, they are only used by one process that collects the result.

Definition: A result object is a grid object accessible via its corresponding reference with the property that object's state can be decomposed into distinct, non-conflicting parts.

The benefit is that such objects can relax the synchronization constraints when object's state is updated, if the object can be decomposed in disjoint parts. Writing any of these parts does not require any specific synchronization and can run in parallel. When the object state is collected by the "reader" process, the state is synchronized by following a global merge procedure. The acquire operation is used to indicate the state when inconsistencies are not tolerated. The implications of the above definition are no locking mechanism is used when processes are writing non-conflicting parts of the result object.

Considering a relatively stable system where requests are being issued every d=5000ms where there are 10 producers and only one consumer, as represented in Figure 10, acquire exclusive time is much smaller for result objects. This is a natural consequence of the fact that there are virtually no locks for the acquire exclusive implementation. On the opposite side, the generic objects show the same behavior presented in previous sections. The acquire time is almost 6 times higher for the result objects as an effect of the object composition procedure out of the disjoint parts. As there is no token movement in this scenario, the token time (TT) is zero for result objects. The penalty of the higher object synchronization time is generated by the higher complexity of the object composition procedure and probably on the higher simultaneous updates which causes a longer execution time of the synchronization procedure.

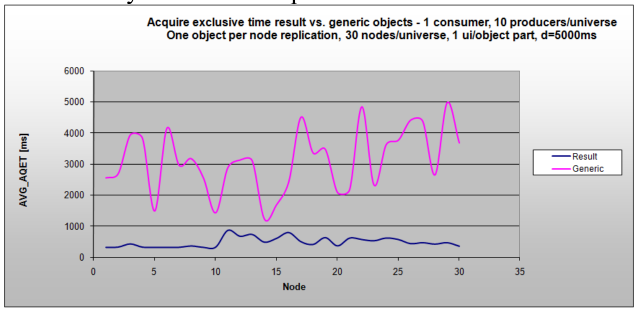

Figure 10. Acquire exclusive time: result vs. generics for 1 consumer

While increasing the number of consumers from one to three and keeping the same experiment parameters as in the previous case, we noticed that the system made out of result objects is still in a relatively stable state where AQET is below 1000 ms, while the system made out of generic objects is moving towards the unstable state (increasing AQT and TT). Increasing the producer-consumer ratio from 10:3 to 5:3 and keeping the same experiment parameters, we noticed that the result objects have an increasing acquire exclusive time, but lower values for the other parameters since the number of consumers has been reduced and thus the overall update penalty.

V. Conclusions

In this paper we summarized the performance analysis of GUN, a prototype of a grid service layer for shared data programming. The experiments covered multiple aspects of the system's dynamics, from a stable and efficient system to a saturated and performance degraded system. The very first experimental results [9] have confirmed the initial hypothesis of the feasibility of shared data programming approaches for large scale distributed systems. One of the first observations during the experiments was that it was very easy and straightforward to program a distributed application using GUN. GUN does not require any knowledge of MPI-like programming concepts, and it requires only the algorithmic representation of the problem. It was more easy and convenient to express interactions via shared data rather than messages. Additionally, it was very straightforward to orchestrate data exchange via workers since there is a built-in data representation at the grid shared object. Even if this is a subjective remark, it appeared quite easy to define distributed data abstractions and let their life cycle be managed by GUN automatically. While this paper focused more on the performance aspects, considering the conducted experiments, GUN showed a good performance as well as scalability. A bottleneck of the primary node's queue could be observed in most of the experiments (except the case of read-only objects) which did not turn to be a problem for modest deployments.

GUN represents an attractive alternative to program large scale grid applications following the shared data paradigm. Base on the grid universe model, it supplies a flexible deployment model based on the network's characteristics as well as a customizable object replication configuration. Having a good scalability and built-in object life-cycle management, it provides an easy to use concept for shared data programming on the grid.

References

[1] Dacian Tudor, Vladimir Cretu and Wolfgang Schreiner, "Designing an Architecture for Distributed Shared Data on the Grid", Proceedings of the International Conference on Algorithms and Architectures, Cyprus, June 9-11, 2008, A. Bourgeois and S.Q. Zheng (Eds.), LNCS 5022, pp. 261–264.

[2] Cheung, B.W.L. Cho-Li Wang Lau, F.C.M. LOTS: a software DSM supporting large object space, IEEE International Conference on Cluster Computing, Pp. 225-234, ISBN: 0-7803-8694-9, 2004.

[3] J.P. Ryan, B.A. Coghlan, SMG: Shared memory for Grids, In: Proceedings of 6th IASTED International Conference on Parallel and Distributed Computing and Systems. 2004, pp. 439-451.

[4] Gabriel Antoniu, Luc Bouge, and Mathieu Jan. JuxMem: An adaptive supportive platform for data sharing on the grid. Scalable Computing: Practice and Experience, 6(3):45-55, November 2005.

[5] J. Osrael, L. Froihofer, and K.M. Goeschka. A Replication Model for Trading Data Integrity against Availabilit, The 12th Int. Symp. on Pacific Rim Dependable Computing, IEEE CS Press, 2006.

[6] Tudor, D., Macariu, G., Schreiner, W. and Cretu, V-I. (2009) 'Experiences on grid shared data programming', International Journal of Grid and Utility Computing, Vol. 1, No. 4, pp.296–307, 2009.

[7] Baker, M. Smith, G, Jini meets the Grid, Proceedings of the International Conference on Parallel Processing Workshops, pp. 193-198, ISBN: 0-7695-1260-7, Spain 2001.

[8] Laurent Baduel, Francoise Baude, Denis Caromel. Efficient, Flexible and Typed Group Communications for Java. Joint ACM Java Grande - ISCOPE 2002 Conference, Seattle, Washington, 2002.

[9] Dacian Tudor, Georgiana Macariu, Wolfgang Schreiner, Vladimir Cretu "Experiments on a Grid Layer Prototype for Shared Data Programming Model", Proceedings 5th International Symposium on Applied Computational Intelligence and Informatics, Timisoara, Romania, May 28-29, 2009. IEEE Catalog: CFP0945C- CDR, ISBN: 978-1-4244-4478-6.